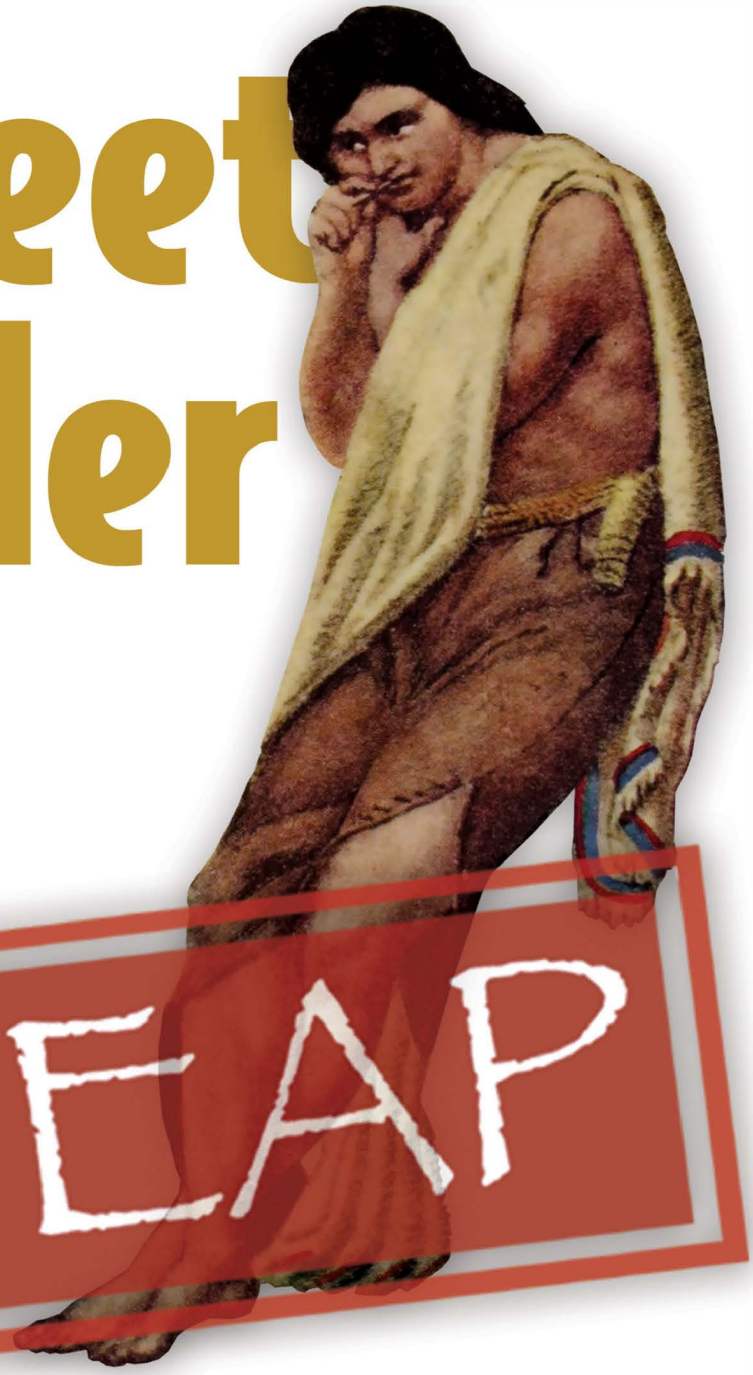
With examples in C#

Street Coder

The rules to break and how to break them



Sedat Kapanoğlu

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Street Coder
With examples in C#
The rules to break and how to break them
Version 5

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP edition of *Street Coder*.

This book is for beginner and medium-level programmers. Whichever learning path that you took, be it a university degree, an online course, a bootcamp, or your own self-teaching adventure, this book will stitch together some gaps that you might have between what you've learned and what you'll experience in the world of professional software development: the streets.

A professional career in software development is demanding and competitive. With this book, you'll understand how working extra for better quality code can save you time in total and let you create great products faster.

Rather than being a comprehensive guide about each and every programming topic, the book tries to bring back some forgotten or forsaken wisdom to the surface and bury some of the wellknown best practices too. The ultimate goal is to cause a perspective change in you about programming wisdom. Yes, you will learn some practical tips and tricks, but more importantly, you will learn how to approach the next trick you'll read in a blog post, or your team lead who is insisting on using it.

To get the most out of this book, it's good to have basic programming skills with C#, understand basic concepts of object-oriented programming. If you already have some tricks in your hat, that's also good, because this book might destroy some of them too.

This book is cultivation of my 25 years of professional software development experience, some notes that I've been taking to share with my colleagues, and some of the writings on the walls of my office written in blood.

I'm extremely excited to go through this journey with you to create this guide for software developers all around the world. Looking forward to your feedback and seeing you become street ready! Please be sure to post any questions, comments, or suggestions you have about the book in the liveBook Discussion Forum.

-Sedat Kapanoglu

brief contents

1 To the streets

2 Practical theory

3 Useful anti-patterns

4 Tasty testing

5 Rewarding refactoring

6 Security by scrutiny

7 Opinionated optimization

8 Palatable scalability

9 Living with bugs

1 To the streets

This chapter covers

- The realities of streets
- Who is a street coder?
- The problems of modern software development
- How to solve your problems with street lore

I am lucky. I wrote my first program in the 80's. It only took me turning on the computer, which took less than a second, writing two lines of code, typing "RUN" and voila! The screen was suddenly filled with my name. I was immediately awestruck with the possibilities ahead of me. If I could do this with three lines, imagine what I could do with six lines, or even 20 lines? My 9-year-old brain got flooded with so much dopamine that I was addicted to programming right there at that instant.

Today, software development is immensely more complex. It's nowhere close to the simplicity of the 80's where user interactions only consisted of "press any key to continue," albeit users occasionally struggled to find an "any" key on their keyboard. There were no windows, no mice, no web pages, no UI elements, no libraries, no frameworks, no runtimes, no mobile devices. All you had was a set of commands and a static hardware configuration.

There is a reason for every level of abstraction we have today and it's not that we are masochists, with the small exception of Haskell[1] programmers. Those abstractions are in place because it's the only way to catch up with the software standards of today. Programming isn't about filling the screen with your name anymore. Your name must have the correct font, it must be in a window so we can drag it around and resize it. It must look good. It should support copy and paste. It must support different names for configurability too. Perhaps it should store the names in a database, in the cloud even. Filling the screen with your name isn't so much fun anymore.

[1] Haskell is an esoteric language that was created as a challenge to fit as many academical papers into a single programming language as possible.

Fortunately, we have resources to contend with the complexity: universities, hackathons, bootcamps, online courses, and *rubber ducks*.

> **TIP** Rubber duck debugging is an esoteric method to find solutions to programming problems that involves to talking to a yellow plastic bird. I'll tell you more about it in the debugging chapter.

We should be well-equipped now with all the resources we have, but the foundation we build for ourselves may not always be sufficient in a high-competition, high-demanding career of software development, **the streets**.

1.1 What matters in the streets

The world of professional software development is quite mysterious. There are customers who swear that they will pay you in a couple of days every day you call them over months. There are employers who don't pay you any salary at all but will pay "once they make money." The chaotic randomness of the universe decides who gets the window office. There are bugs that disappear when you use a debugger. There are teams that don't use any source control at all. Yes, frightening, I know. But you must face the realities.

One thing is clear in the streets though: your throughput is what matters the most. Nobody cares about your elegant design, knowledge of algorithms, your high-quality code. They all care about how much you can deliver in a given time. Counterintuitively, a good design, good use of algorithms, and good quality code can impact your throughput significantly and that's what many programmers miss. They are usually thought of as hindrances, frictions between a programmer and the deadline. That kind of thinking steers you toward being a zombie with a chain and ball attached to your foot.

In fact, there are some people who care about the quality of your code: your colleagues. They don't want to babysit your code. They want your code to work and be easily understandable and maintainable. That is something you owe them. Because, once you commit a code to the source code, it's everybody's code. Because in a team, the team's throughput is more important than that of each member. If you are writing bad code, you are slowing down your colleagues. Your code's lack of quality hurts the team, and a slowed down team hurts the product, and an unreleased product hurts your career.

The easiest thing you can write from scratch is an idea. That's why good design matters. Good design isn't a design that looks good on paper. You can have a design in your mind too and that works. You will encounter people who don't believe in designing and just improvise the code. Those people don't value their time.

Similarly, a good design pattern or a good algorithm can increase your throughput too. If it doesn't help your throughput, it's not useful. As almost everything can be deemed a monetary value, everything you do can be measured in terms of throughput.

You can have high throughput with bad code too, but only in the first iteration. The moment customer requests a change, you are stuck with terrible code to maintain. Throughout the book, I'll be talking about cases where you can identify that you are digging yourself into a hole and can get yourself out before losing your mind.

1.2 Who's a street coder?

Microsoft considers two distinct categories of candidates when hiring: new grads who graduated from a computer science department recently and industry experts, who have substantial experience in software development.

Be it a self-taught programmer or someone who studied computer science, they have a common missing piece at the beginning of their career: *street lore*, which is the expertise to know what matters most. A self-taught programmer has many trials and errors under their belt but can lack formal theory and how it applies to everyday programming. A university graduate, on the other hand, knows a lot about theory but lacks practicality and sometimes a questioning attitude toward what they learned.

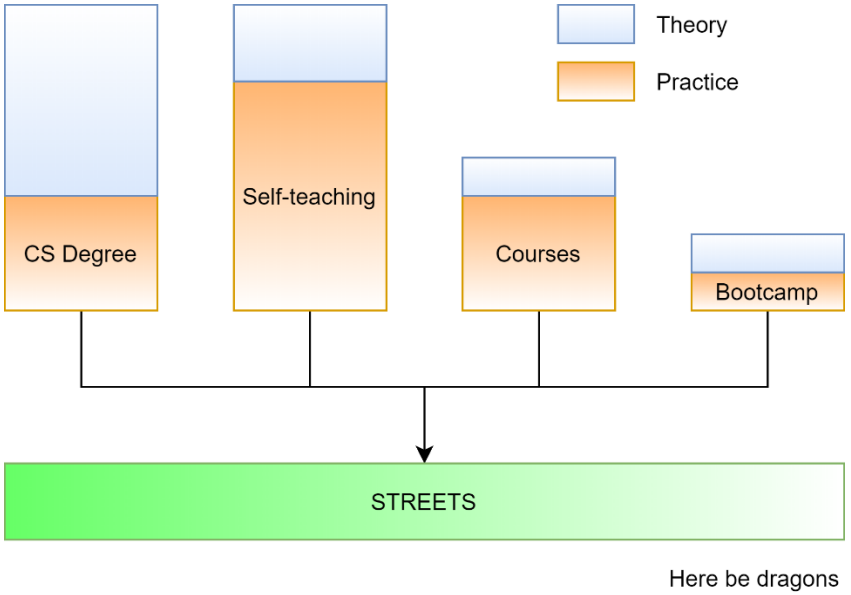


Figure 1.1 Starting a career through different paths

The corpus you learn at school doesn't have priority associated with it. You learn in the order of learning path, not in the order of importance. You have no idea how much certain subjects might be useful in the streets where competition is relentless. Timelines are unrealistic. Coffee is cold. The best framework in the world has that single bug that renders a week of your work worthless. Your perfectly designed abstraction crumbles under the customer who constantly changes their requirements. You manage to quickly refactor your code with some copy/paste, but now you must edit 15 separate places just to change one configuration value.

Over the years, you develop new skills to tackle ambiguity and complexity. Self-taught programmers get to learn some algorithms that help them on their way and university graduates eventually understand that the best theory isn't always the most practical.

A street coder is anyone with software development experience in the industry, who had their beliefs and theories shaped by the realities of an unreasonable boss who wanted a week's worth of work done in the morning. They have learned to back up everything on multiple media after losing thousands of lines of code and have rewritten all of it from scratch. They have seen C-beams glitter in the server room from burning hard drives, and have fought with the systems administrator at the doors of the server room just to get access to production because someone has just deployed an untested piece of code. They have tested their software-compression code on its own source code, only to discover that it's compressed everything into one byte and the value of that byte is 255. The decompression algorithm is yet to be invented.

You've just graduated and looking for a job or you've been fascinated by programming but have no idea what awaits you ahead? You've got out that bootcamp and looking for job opportunities but you're not sure about the knowledge gap you have? You've taught yourself a programming language but you're not sure what else is missing in your skills toolbelt? Welcome to the streets.

1.3 Great street coders

A street coder ideally has these qualities about software development, besides street cred, honor, and loyalty of course:

- Questioning
- Results-driven (aka "results-oriented" in HR-speak)
- High-throughput
- Embraces complexity and ambiguity

Great Software Developers Are Not Just Great Coders

Being a great colleague to work with involves many more skills than just putting bits and bytes into computer. You need to be good at communication, providing constructive feedback and take criticism like a champion. Even Linus Torvalds[2] admitted that he needed to work on his communication skills. However, those are out of the scope of this book. You will have to make friends.

1.3.1 Questioning

Usually, someone talking to themselves is considered unusual at best. Especially if they don't have answers for the questions they ask themselves. However, being a questioning person, asking questions to yourself, asking questions about the most accepted notions, deconstructing them can clarify your vision.

Many books, software experts, and Slavoj Žižek[3] emphasize the importance of being critical, inquisitive, however few of them give you something to work with. In this book,

[2] Linus Torvalds created the Linux operating system, Git source control software, and the culture that swearing at your project's volunteers is okay if they are technically wrong.

[3] Slavoj Žižek is a modern philosopher who suffers from a condition that forces him to criticize everything in the world, without exceptions.

you'll see examples about very well-known techniques, best-practices and how they can be less efficient than they claim to be.

A critique of a technique doesn't mean it's useless. However, it will expand your horizon so you can identify some use cases where an alternative technique might actually be better.

The goal of this book isn't to cover every programming technique from end-to-end but to give you a perspective on how to treat best practices, how to prioritize them based on merit and how you can weigh pros and cons of alternative approaches.

1.3.2 Results-driven

You can be the best programmer in the world, with the best understanding of the intricacies of software development and someone who comes up with the best design for your own code, but those will mean nothing if you are not shipping, if you are not getting the product out.

According to Zeno's paradox[4], to reach an end-goal, you must first reach the halfway through. It's a paradox because to reach the halfway through, you have to reach the quarter way through, and so on, which makes you unable to reach anywhere. Zeno had a point; having an end product requires you to meet deadlines and milestones in-between, too. Otherwise it's impossible to reach your end-goal. Being results-driven also means being milestones-driven, being progress-driven.

> "How does a project get to be a year late? ... One day at a time."
>
> **Fred Brooks, The Mythical Man Month**

Getting results can mean sacrificing code quality, elegance, and technical excellence. It's important to have that perspective at hand and keep yourself in check for what you're doing, and for what's sake.

Sacrificing code quality doesn't mean sacrificing the product quality. If you have good tests in place, if there is a good set of written requirements, you can even go ahead and write everything in PHP[5]. It could mean, however, to bear some pain in the unforeseeable future because code with bad quality will eventually bite you back. It's called code karma.

Some of the techniques you'll learn in the book will help you in making decisions to get results.

1.3.3 High-throughput

The greatest factors affecting your speed of development are experience, good and clear specifications, and mechanical keyboards. Just kidding, contrary to the popular belief, mechanical keyboards don't help your speed at all. They just look cool and are great at annoying your significant other. In fact, I don't think typing speed is helpful in development speed at all. Your confidence in your typing speed might even encourage you to write more elaborate code than necessary.

[4] Zeno was a Greek guy who lived thousands of years ago who couldn't stop asking frustrating questions. Naturally, none of his written works survived.
[5] From what I've heard, PHP has come a long way since it was the butt of the programming jokes and is a fantastic programming language now. It still has some brand image issues to address though.

Some of the expertise can be gained by learning from the mistakes and despair of others. In this book, you'll find examples of such cases. The techniques and knowledge you gained will make you write less code, make decisions faster, and will allow you to have as little technical debt as possible so you won't be spending days untangling your code you wrote only 6 months ago.

1.3.4 Embracing complexity and ambiguity

Complexity is scary, and ambiguity more so, because you don't even know how much you should be scared and that makes you scared even more.

Dealing with ambiguity is one of the core skills Microsoft asks questions about in their interviews. That usually entails hypothetical questions like "How many violin repair shops are there in New York?", "How many gas stations are there in Los Angeles?", "How many secret service agents does the president have, and what's their shift schedule? List their names, and preferably show their walking paths on this blueprint of The White House" and so forth.

The trick to solving these questions comes down to clarifying everything you know about the problem as much as possible and get to an approximation based on those. For example, you can start with New York's population and how many people might be playing violin in the population. That would give you an idea about size of the market and how much competition the market can hold.

Similarly, when encountered with a problem which you don't know all the parameters of, such as estimating the time it would take to develop of a feature, you can always narrow down the window of approximation based on what you can know. You can use what you know to your advantage and leverage it as much as possible, which can leave the ambiguous part miniscule.

Interestingly, dealing with complexity is quite the same. Something that looks extremely complex can be divided in parts that are much more manageable, less complex, and in the end, simple.

The more you clarify, the more you can tackle with the unknown. The techniques you'll learn in the book will hopefully, clarify some of the things and will make you more confident in tackling ambiguity and complexity.

1.4 The problems of modern software development

Besides the increased complexity, countless layers of abstractions, and StackOverflow moderation, modern software development has other issues:

- There are too many technologies: too many programming languages, too many frameworks, and certainly too many libraries, considering NPM (the package manager for NodeJS framework) had a library called "left-pad" solely to add space characters to the end of a string.
- It's paradigm-driven, hence conservative. Many programmers consider programming languages, best practices, design patterns, algorithms, and data structures as relics of an ancient alien race, and we have no idea how they work.
- Technology is getting opaquer, like cars. People used to be able to repair their own cars, replace the carburetor filter themselves. Now with the engines getting

increasingly advanced, all we see under the hood is a metal cover, like on a pharaoh's tomb which will release cursed spirits onto whoever opens it. Software development technologies are no different. Despite that almost everything is open source today, I think they make less sense than a reverse engineered code from a binary from the 90's, because of the changing nature of technologies.

- People don't care about overhead of their code, because we have orders of magnitude of more resources at our leisure today. Did you write a new simple chat application? Why not bundle it with an entire package of a full-blown web browser, because you know it just saves you time and nobody bats an eye when you use gigabytes of memory anyway.
- Programmers are focused on their stack and disregard how the rest works, and rightfully so, they need to bring food to the table and there is no time to learn. I call this "The Dining Developers Problem". Many things that influence the quality of their product go unnoticed because of the constraints they have. A web developer usually has no idea how networking protocols underneath work. They accept the delay when loading a page as is, and learn to live with it, because they don't know a minor technical detail like how an unnecessarily long certificate chain can slow down a web page's loading speed.
- Thanks to the paradigms that has been taught, there is a stigma against menial work, like repeating yourself or copy and paste. You are expected to find a DRY[6] solution. That kind of culture makes you doubt yourself and think of yourself as not good enough, hurting your productivity.

The story of NPM and left-pad

NPM became the de facto JavaScript library package ecosystem in the last decade. People could contribute their own packages to the ecosystem and other packages could use them. That way, it became easier to develop large projects. Azer Koçulu was one of those developers. Left-pad was only one of the packages among 250 that he contributed to NPM ecosystem. It had only one function: append spaces to a string to make sure that it's always a fixed size; quite trivial.

One day, he received an email from NPM saying that they removed one of his packages called "Kik", because a company with the same name complained and they decided to remove Azer's package and give the name to the company. That made Azer so angry that he removed all the packages that he contributed, including Left-pad. The thing is, you see, there were hundreds of large scale projects in the world directly or indirectly using the package. It caused all the projects fail to build and stop in their tracks. It was quite a catastrophe and quite a good lesson about the trust we have on the platforms.

The moral of the story is, the life in the streets are full of unwelcome surprises like this.

In this book, I propose some solutions to these problems, such as going over some core concepts that you might have found boring, prioritizing practicality, simplicity, rejecting some

[6] DRY. Do Not Repeat Yourself. A superstition that says if someone repeats a line of code instead of wrapping it in a function, they will instantly be transformed into a frog.

long-held unquestionable beliefs, and most importantly questioning everything we do. There is value in asking questions first.

1.4.1 Too many technologies

Our constant search for the best technology comes from the fallacy of a silver bullet most of the time. We think that there is a technology out there that can increase our productivity in orders of magnitude. There isn't. For example, Python[7] is an interpreted language. You don't need to compile Python code; it just runs right away. Even better, you don't even need to specify types for the variables you declare which makes you even faster. So, Python must be a better technology than C#, right? Not really.

Because you don't spend time annotating your code with types and compiling it, you miss the mistakes you make. That means you can only discover them during testing or in the production, which are much more expensive than simply compiling code. Most technologies are tradeoffs, rather than productivity boosters. What boosts your productivity is how adept you are in that technology and your techniques, rather than which technologies you're using. Yes, there are better technologies, but they rarely make an order of magnitude difference.

When I wanted to develop my first interactive web site back in 1999, I had absolutely no idea about how to go about writing a web application. Had I tried to search for the best technology first, it would have meant teaching myself VBScript or Perl. Instead, I used what I knew best then: Pascal[8]. It was one of the least suitable languages for that purpose, but it worked. Of course, there were problems with it. Whenever it hanged, the process stayed active in memory in a random server in Canada and it required a support call every time to the service provider so they would restart the physical server. Yet, Pascal let me reach to a prototype the quickest because I was comfortable with it. Instead of launching the web site I imagined after months of development and learning, I wrote and released the code in three hours.

I'm looking forward to showing you ways that you can be more efficient in using the existing toolset you have under your belt.

1.4.2 Paragliding on paradigms

The earliest *programming paradigm* I encountered was Structured Programming back in the 80's. Structured programming is basically writing your code in structured blocks like functions and loops instead of line numbers, `GOTO` statements, blood and tears. It made your code easier to read and easier to maintain without sacrificing performance. Structured Programming sparked my interest in programming languages like Pascal and C.

The next paradigm I encountered came at least half a decade after I learned about structured programming: Object-Oriented Programming, or OOP for short. I remember computer magazines of the time couldn't get enough of it. It was the next big thing and would allow us to write even better programs than we did with structured programming.

[7] Python is a collective effort to promote whitespace, disguised as a practical programming language.
[8] The early source code of Ekşi Sözlük is available on GitHub: https://github.com/ssg/sozluk-cgi

After OOP, I thought I would encounter a new paradigm every five years or so. Conversely, they started to appear more frequently. The 90's introduced us JIT-compiled[9] managed programming languages with the advent of Java, web scripting with JavaScript, and functional programming slowly creeped into mainstream towards the end of the 90's.

Then came 2000's. In the following decades, we saw increased use of the term N-Tier Applications. Fat clients. Thin clients. Generics. MVC, MVVM, and MVP. Asynchronous programming started to proliferate with Promises, Futures, and finally Reactive Programming. Microservices. More functional programming concepts like LINQ, pattern matching, immutability have made it into the mainstream languages. It's like a tornado of buzzwords.

I haven't even gone into design patterns or best practices. We have countless best practices, tips and tricks about almost every subject. There are *manifestos* written about whether we should use tabs or space characters for indenting the source code despite that the obvious answer is spaces, of course[10].

We assume our problems can be solved by employing a paradigm, a pattern, a framework, or a library. Considering the complexity of the problems we face today, it's not unfounded either. However, the blind adoption of those tools can cause more problems in the future: they can render you slower than if you hadn't adopted them by introducing new domain knowledge to learn, their own sets of bugs. They can even force you to change your design. Hopefully, this book will let you have more confidence in using correct patterns, approach them more inquisitively and acquire good comebacks in a code review.

1.4.3 The black boxes of technology

A framework or a library is a package today. Software developers install it, read its documentation, and use it. But usually they don't know how it works. They approach algorithms and data structures the same way too. They use a dictionary datatype because it's handy to keep keys and values. They don't know the consequences.

The unconditional trust in package ecosystems and frameworks is prone to significant mistakes. It can cost us days of debugging because we just didn't know that adding items to a dictionary with the same key would be no different than a list in lookup performance. They could use C# generators when a simple array would suffice and suffer the significant degradation in performance without knowing why.

One day in 1993, a friend handed me a sound card and asked me to install it to my PC. Yes, we used to need external cards to get decent sound from a PC, otherwise all we heard was just a beep. Anyway, I had never opened my PC case before, and I was afraid to damage it. I told him "can't you do this for me?". My friend told me:

"You have to open it to see how it works"

That resonated with me, because I understood that my anxiety was caused by my ignorance rather than my incapability. Opening the case and seeing the insides of my own PC calmed me down. It was couple of boards. Sound card went into one of the slots. It wasn't a

[9] JIT, or Just-In-Time Compilation. A myth created by Sun Microsystems, creator of Java, that if you compile a code while it's running, it will become faster, because the optimizer will have collected more data during runtime. It's still a myth.

[10] I had written about Tabs vs Spaces debate from a pragmatic point of view: https://medium.com/@ssg/tabs-vs-spaces-towards-a-better-bike-shed-686e111a5cce

mystery box to me anymore. I later used the same technique when teaching Art School students about basics of computers. I opened a mouse and showed them its ball. Mice had balls back then. Welp, this was unfortunately ambiguous. I opened the PC case. "You see, it's not scary, it's a board, and some slots".

That later became my motto in dealing with anything new and complex. I stopped being afraid to open the box and usually did it the first thing too so I could face with the whole extent of the complexity, and it was always less than what I feared it to be.

Similarly, the details of how a library, a framework, or a computer works can have tremendous effect on your understanding of what's built on top of it. Opening the box and looking at the parts can help you using the box correctly. You don't really have to read a code from scratch or go through a thousand-page theory book, but you should at least be able to see which part goes where and how it can affect your use cases.

That's why some of the topics I'll be talking about are some unusually fundamental or low-level subjects. It's about opening the box and seeing how things work, so we can have better decision mechanisms for high-level programming.

1.4.4 Underestimating overhead

I'm glad that we are seeing more cloud-based apps every day. Not only are they cost effective, they are also a reality check for us to understand the actual cost of our code. When you start paying an extra cent for every wrong decision you make in your code, overhead suddenly becomes something to watch out for.

Frameworks and libraries usually help us to avoid overhead. They are useful abstractions in that sense. However, we can't delegate all our decision-making process to frameworks. Sometimes we'll have to make decisions for ourselves and we have to take overhead into account. At-scale applications make overhead even more crucial. Every millisecond you save can help you recover precious resources.

A software developer's priority shouldn't be eliminating overhead. However, knowing how overhead can be avoided in certain situations and having the perspective as a tool under your belt will help you save time. Both for yourself, and for the user who is waiting on that spinner[11] on your web page.

Throughout the book, you'll see scenarios and examples where overhead can be avoided easily without making it your utmost priority.

1.4.5 Not my job

One of the ways to deal with complexity is to focus solely on our responsibilities: the component we own, the code we write, the bugs we have caused, and occasionally the exploded lasagna in the office kitchen microwave. It may sound like the most time-efficient way to do our work but like all beings in existence, all code is also interconnected.

Like learning how a specific technology ticks, how a library does its job, learning about dependencies, how they work, and how they are connected can allow us to make better

[11] Spinners are modern hourglasses of computing. In the ancient times, computers used hourglasses to make you wait for an indefinite time. A spinner is the modern equivalent of that animation and it's usually a circular arc rotating around indefinitely. It's just a distraction to keep user's frustration in check.

decisions when writing code. The examples in the book will provide you a perspective to focus not only your area, but its dependencies and outside your comfort zone too, as you'll discover that those prescribe the fate of your code.

1.4.6 Menial is genial

All the principles taught about software development comes down to a single advice: spend less time doing your work. Avoid repetitive, brainless tasks like copying and pasting, writing the same code with little changes from scratch. Because, first, they take longer, and second, it's extremely hard to maintain them.

Not all menial tasks are bad though. Not even all copy & paste is bad. There is a strong stigma against those but there are ways that they can be more efficient than some of the best practices you've been taught.

Besides, not all the code you write works as a code for the actual product. Some of the code you write will be to develop a prototype, some will be for tests, some will be for warming you up to the actual task you have in hand. I'll be discussing some of those scenarios and how you can use those tasks to your advantage.

1.5 What this book isn't

This book is not a comprehensive guide on programming, algorithms, or any subject matter whatsoever. I do not deem myself expert on specific topics, but I deem myself someone with enough expertise on software development. It mostly consists of pieces of information that is not apparent from well-known, popular, and many great books out there. It's definitely not a guide to learn programming.

Experienced programmers might find little or no benefit in the book as they have already acquired sufficient knowledge and already become street coders themselves.

This book is also an experiment where programming books can be fun to read, as I'd like to introduce programming primarily as a fun practice. It doesn't take itself seriously, you shouldn't either. If you feel like a better developer after reading the book and have fun reading it, I deem myself successful.

1.6 Themes

There are certain themes that will be repeating throughout the book:

- Minimal foundational knowledge that is enough for you to get by in the streets. Those subjects will not be exhaustive, but it might spark your interest in them if you saw them before as boring topics. They are usually core knowledge that helps you in making decisions.
- Well-known or well-accepted best practices or techniques that I propose an anti-pattern against that could be more effective in certain cases. The more you read about these, the more it will amplify your sixth sense for critical thinking about programming practices.
- Some seemingly irrelevant looking programming techniques, such as some CPU-level optimization tricks, that might influence your decision-making and code-writing at the

higher level. I find great value in knowing the internals, "opening the box", even if you don't use that piece of information directly.

- Some techniques that I find useful in my day-to-day programming activities which might help you increase your productivity, including eating your nails and being invisible to your boss.

These themes will emphasize a new perspective when looking at programming topics, will change your understanding of certain "boring" subjects and perhaps will change your attitude on certain dogmas. They will make you enjoy your work more.

1.7 Summary

- The harsh reality of "streets", the world of professional software development, requires a set of skills that are not taught or prioritized in formal education or sometimes completely missed in self-teaching.
- New software developers either tend to care about theory or completely ignore them, each on the extreme ends. You'll find a middle point eventually in your life, but it can be accelerated with a certain perspective.
- Modern software development is vastly more complex than how it was a couple of decades ago. It requires tremendous amount of knowledge on many layers just to develop a simple running application.
- Today's programmers face a dilemma between creating software and learning. This can be overcome with reframing topics in a more pragmatic way.
- Lack of clarity about what you work on makes programming a mundane and boring task, reducing your actual productivity. A better understanding about what you do will bring you more joy.

2

Practical theory

This chapter covers

- Why computer science theory is relevant to your survival
- Making types work for you
- Understanding the characteristics of algorithms
- Data structures and their weird qualities that your parents didn't tell you about

Contrary to the popular belief, programmers are human. They carry the same cognitive biases humans have over the practice of software development. They widely overestimate the benefits of not having to use types, not caring about correct data structures, or assuming that algorithms are only important for library authors.

You're no exception. You're expected to deliver a product on time, with good quality and with a smile on your face. As the saying goes, a programmer is effectively an organism that receives coffee as input and creates software as output. You might as well write everything the worst way possible; use copy and paste, use the code you found on StackOverflow, use plain text files for data storage, or make a deal with a demon if your soul isn't already under NDA[1]. Nobody other than your peers really care about how you do things, everybody else wants a good, working product.

Theory can be overwhelming and unrelatable. Algorithms, data structures, type theory, big O notation, polynomial complexity; these can look complicated and irrelevant to software development. Existing libraries and frameworks already handle this stuff, in an optimized and a well-tested way. You're encouraged to never implement an algorithm from scratch anyway, especially in the context of information security or tight deadlines.

Why should you care about theory then? Because, not only does knowledge of computer science theory make you develop algorithms and data structures from scratch, but it also lets

[1] Non-Disclosure Agreement, an agreement that prevents employees from talking about their work unless they start the conversation with "you didn't hear this from me but...".

you make the right decision when you need to use one. It makes you understand the costs of tradeoff decisions. It makes you understand the scalability characteristics of the code you're writing. It makes you see ahead. You will probably never implement a data structure or an algorithm from scratch but knowing how one works will make you an efficient developer. It will improve your chances of survival in the streets.

This book will only go over certain critical parts about theory that you might have missed when learning about them. Some less-known aspects of data types, understanding complexities of algorithms and certain data structures that you need to be aware how they work internally. If you haven't learned about types, algorithms or data structures before, this chapter will provide you cues to get you interested in the subject.

2.1 Crash course on algorithms

An algorithm is a set of rules and steps to solve a problem. Thank you for attending my TED talk. You were expecting a more complicated definition, weren't you? For example, going over the elements of an array to find out if it contains a number is an algorithm, a simple one at that:

```
public static bool Contains(int[] array, int lookFor) {
    for (int n = 0; n < array.Length; n++) {
        if (array[n] == lookFor) {
            return true;
        }
    }
    return false;
}
```

We could have called this *Sedat's Algorithm* if I were the first person to invent it, but it was probably one of the first algorithms ever emerged. It's not clever in any way but it works and it makes sense. That's one of the important points about algorithms, they only need to work for your needs, they don't necessarily have to create miracles. When you put dishes in the dishwasher and run it, you follow an algorithm. Existence of an algorithm doesn't mean it's clever.

There can be smarter algorithms though, depending on your needs. In the code example above, if you know that the list only contains positive integers, you can add special handling for non-positive numbers:

```
public static bool Contains(int[] array, int lookFor) {
    if (lookFor < 1) {
        return false;
    }
    for (int n = 0; n < array.Length; n++) {
        if (array[n] == lookFor) {
            return true;
        }
    }
    return false;
}
```

This could make your algorithm much faster depending on how many times you call it with a negative number. At best, your function would be called always with negative numbers or

zeros, and it would return immediately, even if the array had billions of integers. At the worst case, your function would always be called with positive numbers and you'd be incurring just an extra unnecessary check. Types can help you there as there are unsigned versions of integers called `uint` in C#. So, you can always receive positive numbers and compiler will check for it if you violate that rule, incurring zero performance issues:

```
public static bool Contains(uint[] array, uint lookFor) {
    for (int n = 0; n < array.Length; n++) {
        if (array[n] == lookFor) {
            return true;
        }
    }
    return false;
}
```

We fixed the positive number requirement with type restrictions, rather than changing our algorithm, but it can still be faster based on the shape of data. Do we have more information about the data provided? Is the array sorted? If it is sorted, we can make more assumptions about where our number might be. If we compare our number with any item in the array, we can eliminate a huge chunk of items easily:

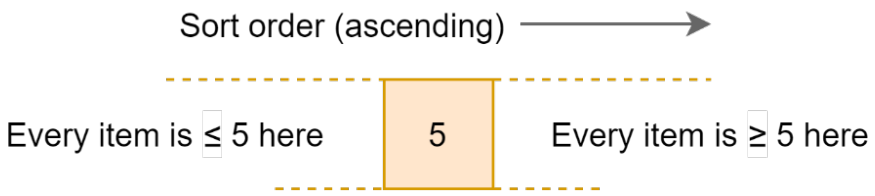


Figure 2.1 We can eliminate one side of the element with one comparison on a sorted list

If our number is, say, three, and if we compare it with five, we can make sure that our number won't be anywhere right of five. That means we can eliminate all the elements right of the list immediately.

Because of that, if we pick the element from the middle of the list, it will be guaranteed that we can eliminate the at least half of the list after the comparison. We can apply the same logic to the half remaining, and pick a half point there, and go on. That means, we only need to make three comparisons at most for a sorted array with eight items to determine if an item exists in it. More importantly, it will only take about ten lookups at most to find if an item exists in an array with a thousand items. That's the power you get by going over in halves. Your implementation could look like listing 2.1. We basically continuously find a middle spot and eliminate the remaining half depending on how the value that we're looking for would fall into. We write the formula in a longer more elaborate form although it corresponds to `(start + end) / 2`. That's because `start + end` can overflow for large values of `start` and `end` and would find an incorrect middle spot. If you write the expression as in listing 2.1, you avoid that overflow case.

Listing 2.1 Searching a sorted array with binary search

```
public static bool Contains(uint[] array, uint lookFor) {
  int start = 0;
  int end = array.Length - 1;
  while (start <= end) {
    int middle = start + ((end - start) / 2);    #A
    uint value = array[middle];
    if (lookFor == value) {
      return true;
    }
    if (lookFor > value) {
      start = middle + 1;    #B
    } else {
      end = middle - 1;    #C
    }
  }
  return false;
}
```

#A Find the middle spot and avoid overflows.
#B Eliminate the left half of the range.
#C Eliminate the right half of the range.

Here we implemented binary search, a much faster algorithm compared to *Sedat's Algorithm*. Since we can now imagine how binary search can be faster than a plain iteration, we can start thinking about the revered Big O notation.

2.1.1 Big O better be good

Understanding growth is a great skill for a developer to have. Be it in size or numbers, when you know how fast something grows, you can see the future, you can see what kind of trouble you're getting into before spending so much time on it. It's especially useful when the light at the end of the tunnel is growing despite that you're not moving.

Big O notation, as the name suggests, is just a notation to explain growth and it's subject to misconceptions too. When I first saw O(N), I thought it was a regular function which is supposed to return a number, I guess? It isn't. It's a way how mathematicians explain growth. It gives us a basic idea about how scalable an algorithm is. Going over every element sequentially (aka Sedat's Algorithm) has a number of operations linearly proportional to the number of elements in the array. We denote that by writing O(N), N denoting the number of the elements. We still can't know how many steps the algorithm will take just by looking at "O(N)" but we know that it grows linearly. That allows us to make assumptions of the performance characteristics of an algorithm depending on the data size. We can foresee at which point it can turn bad by looking at it.

The binary search we implemented has a complexity of $O(\log_2 N)$. If you're not familiar with logarithms, it's the opposite of exponential, so having a logarithmic complexity is actually a great thing unless money's involved. In our example above, if our sorting algorithm magically had logarithmic complexity, it would take only 18 comparisons to sort an array with 500,000 items. Our binary search implementation is that great because of it.

Big O notation isn't only used for measuring increase in computational steps, aka *time complexity*, but it's also used for measuring increase in memory usage too, which is called

space complexity. An algorithm might be fast but could have polynomial growth in memory like our sorting example above. We should know the distinction.

> **KNOW IT RIGHT** Contrary to the popular belief $O(N^x)$ doesn't mean exponential complexity. It denotes *polynomial complexity* which is, although quite bad, not as terrible as exponential complexity which is denoted by $O(x^N)$ instead. With a mere 100 items, $O(N^2)$ would iterate 10,000 times while $O(2^N)$ would iterate some mind-boggling number of times with 30 digits; I can't even pronounce it. There is also factorial complexity, which is even worse than exponential, but I haven't seen any algorithms apart from calculating permutations or combinations using it, probably because nobody was able to finish inventing it.

Since Big-O is about growth, the largest growth function in the notation is the most important part. So, practically O(N) and O(4N) are equivalent as far as Big-O cares. O(N.M) on the other hand, dot being the multiplication operator, may not be so when both N and M are growing. It can even be $O(N^2)$ effectively. O(N.logN) is slightly worse than O(N), but not as bad as $O(N^2)$. O(1), on the other hand, is amazing. It means that the performance characteristics aren't related to number of elements in the given data structure for an algorithm, also known as *constant time*.

Imagine that you implemented a search feature which finds a record in the database by iterating all over them. That means your algorithm would grow linearly proportional to the number of items in the database. Assume that accessing every record takes a second because I guess we're using leaves on water for data storage now. That means searching for an item in a database of sixty items would take a minute. That's O(N) complexity. Other developers in your team can come up with different algorithms as shown in table 2.1.

Table 2.1 Impact of complexity on performance

Search algorithm	Complexity	Time to find a record among 60 rows
The DIY quantum computer Lisa's uncle has in his garage	O(1)	1 second
Binary search	O(log N)	6 seconds
Linear search (because your boss asked you to do it an hour before the presentation)	O(N)	60 seconds
The intern accidently put two for loops nested.	$O(N^2)$	1 hour
Some randomly pasted code from StackOverflow that also finds a solution to some chess problem while searching but the developer didn't bother to remove that part.	$O(2^N)$	36.5 billion years
Instead of finding the actual record, the algorithm tries to find the arrangement of records that spell out the record you're looking for when sorted in a certain way. Good news is that the developer doesn't work here anymore.	O(N!)	End of the universe, but still before those monkeys finish their so-called Shakespeare.

You need to be familiar with how Big-O notation explains the growth in an algorithm's execution speed and memory usage so you can make informed decisions when choosing which data structure and algorithm to use. Be familiar with Big-O, even though you may not need to implement an algorithm. Beware of complexity.

2.2 Inside data structures

> **In the beginning, there was void. When the first electrical signals hit the first bit in the memory, there became data. Data was only free-floating bytes. Those bytes got together and created structure.**
>
> **-Init 0:1**

Data structures are about how data is laid out. People discovered that when data is laid out in a certain way, it can become more useful. A shopping list on a piece of paper is easier to read if every item is on a separate line. A multiplication table is more useful if it's arranged in a grid. Understanding how a certain data structure works is essential for you to be a better programmer. That understanding begins with popping the hood and looking at how it works.

Let's look at arrays for example. An array in programming is one of the simplest data structures, and it's laid out like contiguous elements in memory. Let's say you have this array:

```
var values = new int[] { 1, 2, 3, 4, 5, 6, 7, 8 };
```

You can imagine that it would look like this in memory:

Figure 2.2 A symbolic representation of an array

Actually, it wouldn't look like that in memory, as every object in .NET has a certain header, a pointer to the virtual method table pointer, and length information contained within:

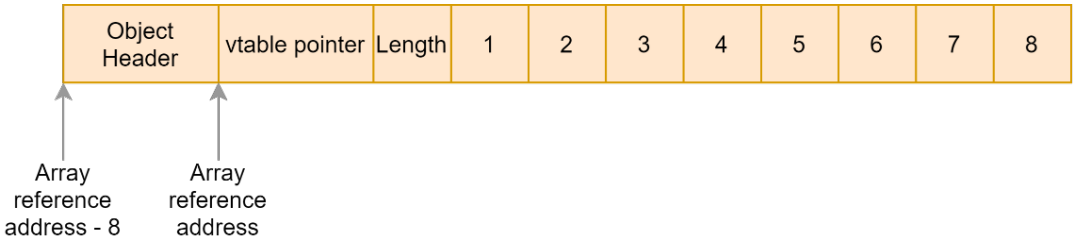


Figure 2.3 Actual layout of an array in memory

It becomes even more realistic if you look at it how it's placed in RAM, as RAM isn't built in integers. I'm sharing these because I want you to be unafraid of these low-level concepts. Your understanding of these will help you at all levels of programming:

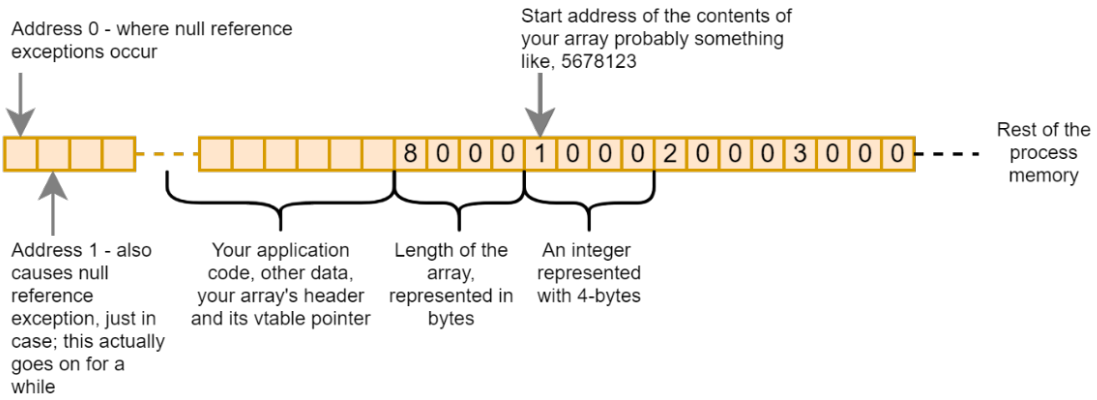


Figure 2.4 Memory space of a process and an array

This isn't how your actual RAM looks like, as every process has their own slice of memory dedicated to them, related to how modern operating systems work. But this is the layout that you'll always be dealing with unless you start developing your own operating system or your own device drivers.

All in all, how data is laid out can make things faster or more efficient, or the opposite. It's crucial to know some basic data structures and how they work internally:

2.2.1 String

Strings could be the most humane data type in the world of programming. They represent text, and are usually human readable. You're not supposed use strings when there is a more suited type but they are inevitable and convenient. When you use strings, you have to know some basic facts about them that are not apparent from the get-go.

Although they resemble arrays in usage and structure; Strings in .NET are *immutable*. Immutability means that the contents of a data structure cannot be changed after it's initialized. Assume that we'd like to join names of people to produce a single, comma separated string, and that we travelled two decades back in time so there is no better way of doing this:

```
public static string JoinNames(string[] names) {
    string result = String.Empty;    #A
    int lastIndex = names.Length - 1;    #B
    for (int i = 0; i < lastIndex; i++) {
        result += names[i] + ", ";
    }
    result += names[lastIndex];    #C
    return result;
}
```

#A If we didn't initialize the string, it would have a default value of null, which would have been caught by nullability checks if you had used them.
#B Index of the last element.
#C This way, we avoid finishing the string with a comma.

It might seem like we have a string called `result` and we are modifying the same string over the course of execution at first look, but it's not the case. Every time we assign `result` a new value, we are creating a new string in memory. .NET needs to determine the length of the new string, allocate new memory for it, and copy the contents of other strings into the newly built memory and return it to you. That is quite an expensive operation and the cost increases as the string and the trail of garbage to collect gets longer.

There are tools in the framework to avoid this problem. Even if you don't care about performance, these tools are free so you don't really need to change your logic or jump hoops in order to get better performance. One of those is `StringBuilder` with which you can work to build your final string and retrieve it with a `ToString` call in one shot.

```
public static string JoinNames(string[] names) {
    var builder = new StringBuilder();
    int lastIndex = names.Length - 1;
    for (int i = 0; i < lastIndex; i++) {
        builder.Append(names[i]);
        builder.Append(", ");
    }
    builder.Append(names[lastIndex]);
    return builder.ToString();
}
```

`StringBuilder` uses consecutive memory blocks internally, instead of reallocating and copying every time it needs to grow the string, therefore it's usually more efficient than building a string from scratch.

Obviously, an idiomatic and a much shorter solution has been available for a long time, but your use cases may not always overlap with these:

```
String.Join(“, “, names);
```

Concatenating a string is usually okay when initializing the string, because that involves only a single buffer allocation after calculating the total length required. For example, if you have a function that joins first name and last name with a space inbetween using addition operator, you're only creating a single new string in one shot, not multiple:

```
public string ConcatName(string firstName, string middleName,
  string lastName) {
  return firstName + " " + middleName + " " + lastName;
}
```

This might seem like a no-no, as if `firstName + " "` would create a new string first, then it would create new string with `middleName` and so on, but the compiler actually turns into a single call to a `String.Concat()` function, which allocates a new buffer with the length of the sum of the lengths of all strings and returns it in one shot. So, it's still fast. But when you concatenate strings in multiple shots, with `if` clauses inbetween, or loops, compiler can't optimize that. Know when it's okay to concatenate strings and when it's not.

Immutability isn't a holy seal that cannot be broken either. There are ways around modifying strings in place, or other immutable structures for that matter, which mostly involves unsafe code and astral beings, but it's not usually recommended as strings are deduplicated by .NET runtime, and some of their properties such as hash codes are cached. The internal implementation relies heavily on the immutability characteristic.

String functions work with the current *culture* by default and that might be painful to experience when your app stops working in another country.

> **HINT** A culture, also known as locale in some programming languages, is a set of rules for performing region specific operations, like sorting strings, displaying date/time in correct format, how to place utensils on the table, and so forth. Current culture is usually what operating system thinks it's using.

Understanding cultures can make your string operations safer and faster. For instance, consider a code where we detect if the given file name has ".gif" extension:

```
public bool isGif(string fileName) {
    return fileName.ToLower().EndsWith(".gif");
}
```

We are smart, you see; we turn the string to lower-case so we handle the case where the extension could be ".GIF" or ".Gif" or any other combination of cases. The thing is, not all languages have the same lower-case semantics. In Turkish language for instance, the lower-case of "I" is not "i" but "ı" also known as dotless-I. The code above would fail in Turkey, and maybe in some other countries like Azerbaijan too. And by lower-casing the string, we are in fact, creating a new string as we learned, which is inefficient.

.NET supplies culture-invariant versions of some string methods, like `ToLowerInvariant`. It also provides some overloads of the same method that receives a `StringComparison` value which has invariant and ordinal alternatives. Therefore, you can write the same method in a safer and faster way:

```
public bool isGif(string fileName) {
    return fileName.EndsWith(".gif",
        StringComparison.OrdinalIgnoreCase);
}
```

This way, we avoid creating a new string and we're using a culture-safe and faster string comparison method which doesn't involve our current culture and its intricate rules. We could have used `StringComparison.InvariantCultureIgnoreCase` too, but unlike ordinal comparison, it adds a couple of more translation rules such as treating German umlauts or graphemes with their Latin counterparts ("ß" vs "ss") which might cause problems with file names or other resource identifiers. Ordinal comparison compares character values directly without involving any translation.

2.2.2 Array

We looked at how an array looks like in memory. Arrays are practical to keep a number of items whose numbers won't be growing beyond the array's size. They are static structures. They cannot grow or change size. If you want a larger array, you have to create a new one

and copy the contents of the old one over. There are a couple of things to know about arrays though.

Arrays, unlike strings, are mutable. That's what they are about. You can freely play with their contents. Actually, it's really hard to make them immutable, which makes them poor candidates for interfaces. Consider this property:

```
public string[] Usernames { get; }
```

Even though the property has no setter, the type is still an array, which makes it mutable. There is nothing that prevents you from doing:

```
Usernames[0] = "root";
```

Which can complicate things, even when it's only you who's using the class. You shouldn't allow yourself to make changes to the state unless it's absolutely needed. State is the root of all evil, not null. The fewer states your app has, the fewer problems you'll have.

Try to stick to the type that has the smallest functionality for your purpose. If you only need to go over the items sequentially, stick to `IEnumerable<T>`. If you also need a repetitively accessible count, use `ICollection<T>`. Note that, the LINQ extension method `.Count()` has special handling code for types that support `IReadOnlyCollection<T>`, so even if you use it on an IEnumerable, there is a chance that it might return a cached value instead.

Arrays are best suited for using inside the local scope of a function. For any other purpose, there is a better suited type or interface to expose in addition to `IEnumerable<T>`, like `IReadOnlyCollection<T>`, `IReadOnlyList<T>`, or `ISet<T>`.

2.2.3 List

A list behaves like an array that can grow slightly similar to how `StringBuilder` works. It's possible to use lists over arrays almost everywhere, but that will incur an unnecessary performance penalty due to indexed accesses being *virtual calls* in a list while an array uses direct access.

You see, object-oriented programming comes with a nice feature called *polymorphism* which means an object can behave according to the underlying implementation without its interface changing. If you have, say, a variable `a` with a type of `IOpenable` interface, `a.Open()` might open a file or a network connection depending on the type of the object assigned to it. This is achieved by keeping a reference to a table that maps virtual functions to be called to the type at the beginning of the object, called virtual method table, or vtable for short. This way, although `Open` maps to the same entry in the table in every object with the same type, you wouldn't know where it's going to lead until you look up the actual value in the table.

Because we don't know what exactly we're calling, these are named virtual calls. A virtual call involves an extra lookup from the virtual method table, so it's slightly slower than regular function calls. That may not be a problem with a couple of function calls, but when it's done inside an algorithm, its overhead can grow polynomially too. Because of that, if

your list won't grow in size after initialization, you might want to use an array over a list in a local scope.

Normally, you should almost never think about these details. But when you know the difference, there are cases where an array might be preferable to a list.

Lists are similar to `StringBuilder`, both are dynamically growing data structures, but lists are less efficient in growth mechanics. Whenever a list decides that it needs to grow, it allocates a new array with a larger size and copies the existing contents to it. StringBuilder, on the other hand, keeps chunks of memory chained together instead which doesn't require a copy operation. The buffer area for lists grows whenever the buffer limit is hit, but the size of the new buffer gets doubled every time which means the need for growth gets reduced over time. Still, this is an example where using specific class for the task at hand is more efficient than using a generic one.

You can get great performance from lists too, by specifying a capacity. If you don't specify a capacity to a list, it will start with an empty array. It will then increase its capacity to a few items. It will double its capacity after it's full. If you set a capacity while creating the list, you avoid unnecessary growth and copying operations altogether. Keep this in mind when you already know the maximum number of items the list will have beforehand.

Don't make a habit of specifying list capacity without knowing the reason though. That might cause unnecessary memory overhead and those can accumulate. Make a habit of making conscious decisions.

2.2.4 Linked list

You can consider linked list as lists where elements aren't consecutive in memory, but each element points to the address of the following item. They are useful for their O(1) insertion and removal performance. You can't access individual items by index because they can be stored anywhere in memory, and it's not possible to calculate it, but if you mostly access to the beginning or to the end of the list, or you just need to enumerate the items, it can be as fast too. Otherwise, say, checking if an item exists in a linked list is an O(N) operation.

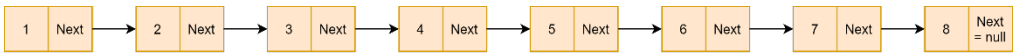


Figure 2.5 Layout of a linked list

That doesn't mean a linked list is always faster than a regular list though. Individual memory allocations for each element instead of allocating a whole block of memory in one shot and additional reference lookups can also hurt performance.

You might have needed a linked list whenever you need a queue or stack structure, but .NET covers that. So, ideally, unless you're into systems programming, you shouldn't need to use a linked list in your daily work except for job interviews. Interviewers love their puzzle questions with linked lists unfortunately, so it's still important for you to get familiar with them.

No, you won't reverse a linked list

Answering coding questions in interviews is a rite of passage for software development positions. Most of the coding questions also cover some data structures and algorithms. Linked lists are part of the corpus, so there is a chance that you might encounter someone asking you to reverse a linked list or invert a binary tree.

You will probably never perform those tasks in your actual job, but to give the credit to the interviewer, they are testing your knowledge of data structures and algorithms to simply assess that you know what you're doing. They are trying to make sure that you're capable of making the right decision when there comes a need to use the right data structure at the right place. They are also testing your analytical thinking and problem-solving ability, so it's important for you to think aloud and present your process of thought to the interviewer.

You don't always need to solve the given question. Interviewer usually looks for someone who is passionate and knowledgeable about certain basic concepts and can find their way around, even though they might get lost.

I, for example, usually followed up my coding questions to the candidates at Microsoft with an extra step for them to find bugs in their code. That actually made them feel better because it felt like bugs were expected and they were not assessed based on the how bug-free the code is, but how they can identify bugs.

Interviews aren't only about finding the right person, but also about finding someone you'd enjoy working with. It's important for you to be a curious, passionate, persistent, easygoing person that can really help them in their tasks.

Linked lists were more popular in the ancient times of programming because memory efficiency took precedence. We just couldn't afford to allocate kilobytes of memory just because our list needed to grow. We had to keep tight storage. Linked list was the perfect data structure for that.

They are also still used frequently in operating system kernels because of their irresistible O(1) characteristic for insertion and removal operations.

2.2.5 Queue

A queue is a data structure that represents the most basic form of civilization. It allows you to read items from a list in the order of insertion. A queue can simply be an array, as long as you keep separate spots for reading the next item and inserting the new one. If we added ascending numbers from a queue it would resemble this:

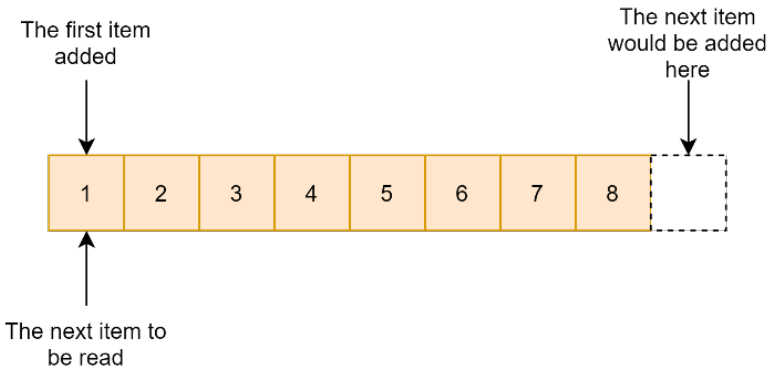


Figure 2.6 A high-level layout of a queue

The keyboard buffer on PC's in MS-DOS era used a simple array of bytes to store key presses. The buffer prevented keystrokes from getting missed because of the slow or unresponsive software. When the buffer was full, the BIOS would beep so we would know that our keystrokes weren't being recorded anymore.

Fortunately, .NET comes with an existing `Queue<T>` class where we can use without worrying about implementation details and performance.

2.2.6 Dictionary

Dictionaries, also known as *hashmaps* or sometimes *key/value things*, are one of the most useful and the most used data structures. We take their capabilities for granted though. A dictionary is a list that can store a key and a value. It can later retrieve a value with a key in constant, aka O(1), time. That means they are extremely fast for data retrieval. How are they so fast? What's the magic?

The magic lies in the word "hash". Hashing is the term for generating a single number from arbitrary data. The number generated must be deterministic which means that it must generate the same number for the same data, but it doesn't have to generate a unique value. There are many different ways to calculate a hash value. The hashing logic of an object resides in the GetHashCode implementation.

The nice thing about hashes is that because you get the same value every time, you can use the hash values for lookups. Imagine, if you have an array of all possible hash values, you can look them up with an array index. But such an array would take about eight gigabytes for each dictionary created.

Dictionaries allocate a much smaller array instead and rely on their distribution. Instead of looking up the hash value they look up "hash value mod array length". Let's say that a dictionary with the key of integers allocates an array of six items to keep an index for them, and the `GetHashCode()` method for an integer would just return its value.

That means our formula to find out where an item would map to would simply be `value % 6`, since array indices start at zero. An array of numbers from one to six would be distributed like this in our array:

Figure 2.7 The distribution of items in a dictionary

What happens when we have more than the capacity of the dictionary? There will be overlaps, that's for sure. So, dictionaries keep the overlapping items in a linked list. Let's say we store items with keys from one to seven, the key array would look like this:

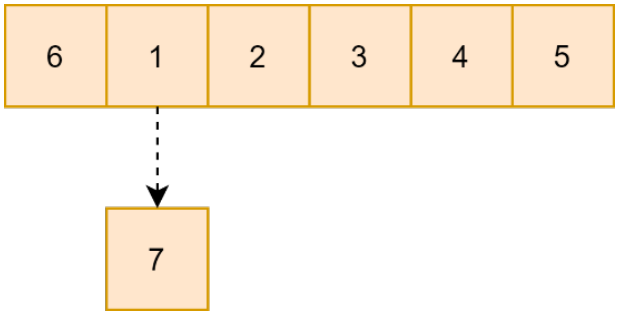


Figure 2.8 Storage of overlapping items in a dictionary

Why am I talking about this stuff? Because key lookup performance of a dictionary is O(1), normally; but lookup overhead of a linked list is O(N). That means as the number of overlaps increases, lookup performance will slow down too. If you had a GetHashCode function that always returned, say 4, for instance[2]:

```
public override int GetHashCode() {
    return 4; // chosen by fair dice roll
}
```

That means the internal structure of the dictionary would look like this when you add items to it:

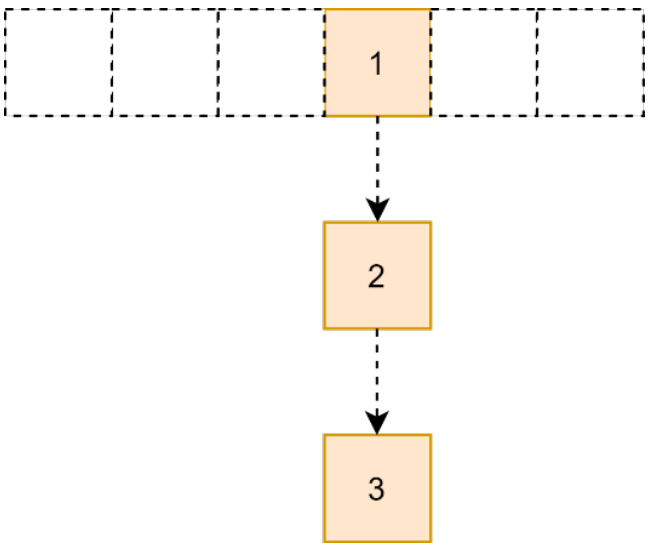


Figure 2.9 A dictionary when you screw up your GetHashCode()

[2]Inspired by this excellent XKCD about random numbers: https://xkcd.com/221

A dictionary is no better than a linked list if you have bad hash values. It can even have worse performance due to extra plumbing the dictionary uses to juggle these items. That brings us to the most important point: Your GetHashCode function needs to be as unique as possible. If you're having many overlaps, your dictionaries will suffer, for a suffering dictionary your application will suffer, for a suffering application an entire company will suffer. In the end, you will suffer. For want of a nail, a nation was lost.

Sometimes, you had to combine values of multiple properties in a class to calculate a unique hash value. For instance, repository names are unique per user on GitHub. That means any user can have a repository with the same name and the repository name itself isn't enough to make it unique. Had you used name only, it would cause more collisions. That meant you had to combine hash values. Similarly, if our web site had unique values per topic, we would have the same problem.

To combine hash values efficiently you had to know their ranges and deal with their bitwise representation. Had you simply used an operator like addition, or simple OR/XOR operations, you might have still ended up with many more collisions than you anticipated. You had to involve bit shifts too. A proper GetHashCode function would use bitwise operations in order to get a good spread over the full 32-bits of an integer.

The code for such an operation might look like a hacking scene from a cheesy hacker movie. It's cryptic and hard to understand even for someone who is familiar with the concept. We're basically rotating one of the 32-bit integers by 16 bits so their lowest bytes are moved towards the middle and XOR'ing ("^") that value together with the other 32-bit integer hence lowering the chances of collisions a lot. It looks like this, scary:

```
public override int GetHashCode() {
    return (int)(((TopicId & 0xFFFF)<< 16)
        ^ (TopicId & 0xFFFF0000 >> 16)
        ^ PostId);
}
```

Luckily, with the advent of .NET Core and .NET 5, combining hash values in a way that gives the least collisions has been abstracted away behind HashCode class. In order to combine two values, all you have to do know is:

```
public override int GetHashCode() {
    return HashCode.Combine(TopicId, PostId);
}
```

Hash codes are not only used in dictionary keys. They are also used in other data structures like sets too. Since it's far easier to write a proper GetHashCode with helper functions, you have no excuse to skip on it. Keep an eye on it.

When not to use `Dictionary`? If you only need to go over key/value pairs sequentially, a dictionary provides no benefits. It can, in fact, harm performance. Consider using a `List<KeyValuePair<K,V>>` instead, so you'd avoid unnecessary overhead.

2.2.7 HashSet

A set is like an array or a list except that it can only contain unique values. Its advantage over arrays or lists is that it has O(1) lookup performance like dictionary keys, thanks to

hash based maps we just looked into. That means, if you need to perform a lot of checks to see if a given array or list contains an item, using a set might be way faster. It's called HashSet in.NET, and it's free.

Because HashSet is fast for lookups and insertion, it's also suitable for intersection and union operations. It even comes with methods that provide the functionality. To get the benefits you need to pay attention to your GetHashCode() implementations again.

2.2.8 Stack

Stack is a LIFO (Last In First Out) queue. They are useful when you want to save state and restore it in the reverse order it's saved. When you visit a DMV (Department of Motor Vehicles) office in real life, you sometimes need to use a stack. You first approach counter 5, and the employee at the counter checks your documents and sees that you're missing a payment, so they send you to counter 13. The employee at counter 13 sees that you're missing a photo in your documents and sends you to another counter, this time counter 47, to get your photo taken. Then you have to trace your steps back to the counter 13 where you take the payment receipt and go the first counter 5 to get your driving permit. The list of counters and how you process them in order (LIFO) is a stack-like operation, and they are usually more efficient than DMV.

Stack can be represented with an array. What's different is where you put the new items and where you read the next item from. If we had built a stack by adding numbers in an ascending order, it would look like this:

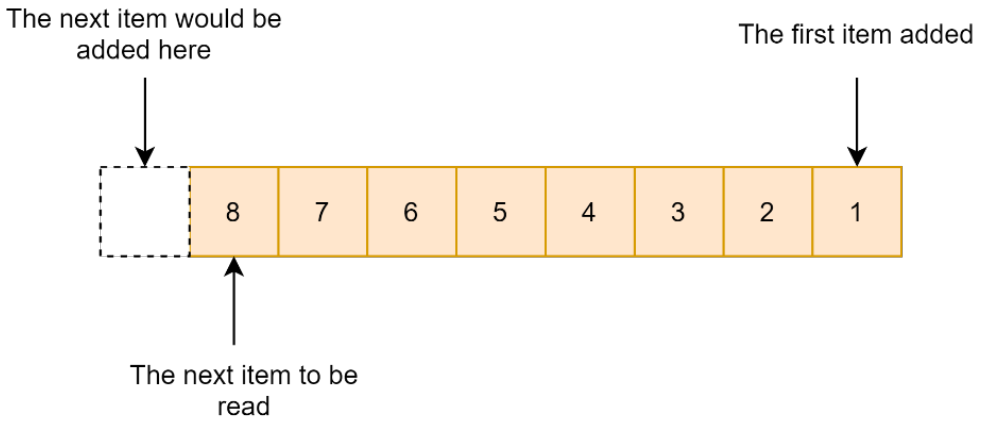


Figure 2.10 A high-level overview of a stack

Adding to a stack is usually called *pushing* and reading the next value from a stack is called *popping*. Stacks are useful for backtracking your steps. You might already be familiar with the *call stack* because it shows you where an exception occurred; not only where, but which execution path it followed too. Functions know where to return after they are done executing

by using a stack. Before calling a function, the return address is added to the stack. When the function wants to return to its caller, the last address pushed onto the stack is read and CPU continues execution at that address.

2.2.9 Call stack

Call stack is the data structure where functions store the return addresses so the called functions know where to go back when they are done executing. There is one call stack per *thread*.

Every application runs in one or more separate processes. Processes allow memory and resource isolation. Every process has one or more threads. Threads are the unit of execution. All threads run parallel to each other on an operating system, hence the term *multi-threading*. Even though you might only have a four core CPU, the operating system can run thousands of threads in parallel. This can happen because most threads are waiting for something to complete most of the time, so it's possible to fill their slot with some other thread and have a sense of all threads running in parallel. That makes multitasking possible even on a single CPU.

There was a time where a process was both the container for application resources and units of execution in older UNIX system. Although the approach was simple and elegant, it caused problems like *zombie processes*. Threads are more lightweight and have no such problem as they are bound to the execution lifetime.

Every thread has their own call stack: a fixed amount of memory. By tradition, stack grows from top to bottom in the process memory space, top meaning the end of the memory space and bottom meaning our famous null pointer: address zero. Pushing an item onto the call stack means putting the item there and decrementing the *stack pointer*.

Like every good thing, stack has an end. It has a fixed size, so when it grows beyond the fixed size, CPU raises a `StackOverflowException`, something you'll encounter in your career whenever you accidentally call a function from itself. The stack is quite large, so you don't usually worry about hitting it in a normal case.

Call stack doesn't only hold return addresses. It also holds function parameters and local variables. Because local variables occupy too little memory, it's very efficient to use stack for them as it doesn't require extra steps of memory management like allocation and deallocation. The stack is fast, but it has a fixed size, and it has the same lifetime as the function using it. When you return from a function, the stack space is given back. That's why it's only ideal to store small amount of local data in it. Because of that, managed runtimes like C# or Java don't store class data in stack and just store their references instead.

That's another reason why value types can have better performance over reference types in certain cases. Value types only exist on stack when locally declared, although they are passed around with copying.

2.3 What's the hype on types?

Programmers take data types for granted. Some even argue that programmers are faster in *dynamically typed* languages like JavaScript or Python because they don't have to deal with intricate details like deciding the type of each variable.

HINT *Dynamically typed* means that data types of variables or class members in a programming language can change during runtime. You can assign a string to a variable then assign an integer to the same variable in JavaScript because it's a dynamically typed language. A statically typed language like C# or Swift wouldn't allow that. We'll go into the details of these.

Yes, specifying types for every variable, every parameter, every member in the code is a chore, but you need to adopt a holistic approach to being faster. Being fast isn't solely about writing code, but maintaining it too. There could be few cases where you may not really need to care about maintenance because you learned that you just got fired and you couldn't care less. Apart from that, software development is a marathon, not a sprint.

Failing early is one of the best practices in development. Data types are one of the earliest defenses against development friction in coding. Types let you fail early and fix your mistakes before they become a burden. Aside from the obvious benefit of not confusing a string with an integer accidentally, you can make types work for you in other ways.

2.3.1 Being strong on the type

Most programming languages have types. Even the simplest programming languages like BASIC had types: strings and integers; some of its dialects even had real numbers. There are a few languages called "typeless" like Tcl, REXX, Forth, and so forth. Those languages only operate on a single type: usually a string or an integer. Not having to think about types makes programming convenient, but it makes written programs slower and more prone to bugs.

Types are basically free checks for correctness, so understanding the underlying type system can help you tremendously to make yourself a productive programmer. How programming languages implement types is strongly correlated with whether they are interpreted or compiled:

- *Interpreted programming languages* like Python or JavaScript let you to run code in a text file immediately without a need for a compilation step. Because of their immediate nature, variables tend to have flexible types: you can assign a string to a previously integer variable, you can even add strings and numbers together. These are usually called *dynamically typed languages* because of how they implement types. You can write code much faster in interpreted languages because you don't really need to declare types.
- *Compiled programming languages* tend to be stricter. How strict they are depends on how much pain the language designer wants to inflict upon you. For example, Rust language can be considered the *German engineering* of programming languages, extremely strict, perfectionist, and therefore error-free. C language can also be considered German engineering but like Volkswagen: it lets you to break the rules and pay the price later. Both languages are statically typed, once a variable is declared its type cannot change, but Rust is called *strongly typed* like C# while C is considered *weakly typed*.

Strongly and weakly typed means how relaxed a language is in terms of assigning different types of variables to each other. C is more relaxed in that sense; you can assign a pointer to

an integer or vice versa without issues, while C# is stricter; pointers/references and integers are incompatible types.

Table 2.1 Flavors of type strictness in programming languages

	Statically typed Variable type *cannot* change in runtime.	**Dynamically typed** Variable type *can* change in runtime.
Strongly typed Different types *cannot* be substituted for each other.	C#, Java, Rust, Swift, Kotlin, TypeScript, C++	Python, Ruby, Lisp
Weakly typed Different types *can* be substituted for each other.	Visual Basic, C	JavaScript, VBScript

Strict programming languages can be frustrating. They can even make you question life and why we exist in the universe when it comes to languages like Rust. Declaring types and converting them explicitly when needed may look like a lot of bureaucracy. You don't need to declare types of every variable, argument, and member in JavaScript for example. Why do we burden ourselves with explicit types if many programming languages can work without them?

The answer is simple: types can help us write code that is safer, faster, and easier to maintain. We can reclaim the time we lost while declaring types of variables, annotating our classes with the time we gained by having to debug fewer bugs, and having to solve fewer issues with performance. Apart from obvious benefits of types, they have some subtle benefits too. Let's go over them.

2.3.2 Proof of validity

Proof of validity is one of the less-known benefits of having predefined types. Suppose that you're developing a microblogging platform that only allows certain amount of characters in every post, and in return, you're not judged for being too lazy to write something longer than a sentence. In this hypothetical microblogging platform, you can mention other users in a post with "@" prefix and mention other posts with "#" prefix followed by the post's identifier. You can even retrieve a post by typing its identifier in the search box. If you type in a username with "@" prefix in the search box, that user's profile will be shown.

User input brings a new set of problems with validation. What happens if user provides letters after "#" prefix? What if they input a longer number than allowed? It might seem like those scenarios work themselves out, but usually your app crashes because somewhere in the code path, something that doesn't expect an invalid input will throw an exception. It's the worst possible experience for the user: they don't know what's gone wrong and they don't even know what to do next. It can even become a security problem if you display that given input without sanitizing it.

Data validation doesn't provide a proof of validity throughout the code. You can validate the input in client, but somebody, a third-party app for example, can send a request without validation. You can validate at the code that handles web requests, but another app of yours,

such as your API code, can call your service code without necessary validation. Similarly, your database code can receive requests from multiple sources, like the service layer and a maintenance task, so you need to make sure that you're inserting the right records in the database.

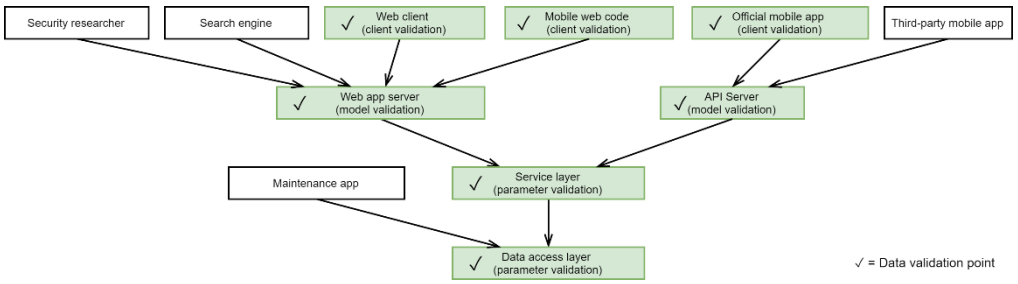


Figure 2.11 Unvalidated data sources and places where you need to validate data repetitively

That might eventually make you validate the input at multiple places around the code and you need to make sure that you're consistent in validation too. You don't want to end up with a post with an identifier of "-1", or a user profile named "' OR 1=1--" (which is a basic SQL injection attack that we will visit in the chapter about security).

Types can carry over proof of validity. Instead of passing an integer for blog post identifiers or strings for usernames, you can have classes or structs that validate their input on construction which makes them impossible to contain an invalid value. It is simple, yet powerful. Any function that receives a post identifier as a parameter asks for a `PostId` class instead of an integer. This allows you to carry over proof of validity after the first validation in the constructor. If it's an integer, it needs to be validated, if it's a `PostId`, it has already been validated; there's no need to check its contents, because there is no way to create it without validation, as you can see in the following snippet. The only way to construct a `PostId` in the code snippet is to call its constructor, which validates its value and throws an exception if it fails. That means it's impossible to have an invalid `PostId` instance:

```
public class PostId
{
    public int Value { get; private set; }    #A
    public PostId(int id) {   #B
        if (id <= 0)  {
            throw new ArgumentOutOfRangeException(nameof(id));
        }
        Value = id;
    }
}
```

#A Our value is impossible to be changed by external code.
#B Constructor is the only way to create this object.

The style of code examples

Placement of curly braces is the second most debated topic in programming that hasn't been settled in a consensus yet right after tabs vs spaces. I prefer Allman style for most C-like languages, especially C# and Swift. Allman style is where every curly brace character resides on its own line. Swift officially recommends using 1TBS (One True Brace Style), aka improved K&R style, where an opening brace is on the same line with the declaration. People, however, still feel the need to leave extra blank lines after every block declaration because 1TBS is too cramped. When you add blank lines, it effectively becomes Allman style, but people can't bring themselves to admit it.

Allman style is the default for C# where every brace is on its own line. I find it much more readable than 1TBS or K&R. Java uses 1TBS by the way.

I've had to format the code in 1TBS style because of the publisher's typesetting restrictions, but I suggest you consider Allman-style when using C# not only because is it more readable, but because it's the most common style for C#.

When you decide to go that path, it's not as easy as the example I've shown though. For example, comparing two different `PostId` objects with the same value wouldn't work as you expected, as by default, comparison only compares references, not the contents of the classes (I'll be talking about references versus values later in this chapter). You have to add whole scaffolding around it to make it work without issues. Here is a quick checklist:

- You have to at least implement an override for `Equals` method as some framework functions and some libraries can depend on it to compare two instances of your class.
- If you plan on comparing values yourself using equality operators ("`==`" and "`!=`") you have to implement their *operator overloads* in the class.
- If you plan to use it in a `Dictionary<K,V>` as a key, you have to override `GetHashCode` method. I will be describing how hashing and dictionaries are related later in this chapter.
- String formatting functions such as `String.Format` uses `ToString` method to get a string representation of the class suitable for printing.

Don't use operator overloading unless necessary

Operator overloading is a way to change how operators like "`==`", "`!=`", "`+`", and "`-`" in a programming language behave. Developers who learn about operator overloading might go overboard and tend to create their own language with weird behavior for irrelevant classes like overloading "`+=`" operator to insert a record to a table with a syntax such as `db += record`. It's almost impossible to understand the intent of such code. It's also impossible to discover too unless you read the documentation. There is no IDE function to discover which operators a type is overloading. Don't be the person who uses operator overloading needlessly. Even you will forget what it does and will beat yourself up over this. Use operator overloading only to provide alternatives to equality and typecasting operators, and only when needed. Don't waste time implementing them if they won't be needed.

We'll be using operator overloading in some of the examples because it's required to make classes semantically equivalent to the values they represent. You'd expect a class to work with `==` operator the same way the number it represents would.

A PostId class with all necessary plumbing to make sure it works in all equality scenarios is shown in listing 2.2. We overrode `ToString()` so our class becomes compatible with string formatting and easier to inspect its value while debugging. We overrode `GetHashCode()` so it returns Value directly because the value itself can fit perfectly into an `int`. We overrode `Equals()` method so equality checks in collections of this class work correctly in case we need unique values, or we'd like to search against this value. We finally overrode "`==`" and "`!=`" operators so we can directly compare to PostId values without accessing its value.

> **MINI-HINT** An immutable class solely to represent values is called a *value type* in the streets. It's good to know colloquial names but don't focus on them. Focus on their utility.

Listing 2.2 Full implementation of a class encompassing a value

```
public class PostId
{
    public int Value { get; private set; }
    public PostId(int id) {
        if (id <= 0) {
            throw new ArgumentOutOfRangeException(nameof(id));
        }
        Value = id;
    }
    public override string ToString() => Value.ToString();    #A
    public override int GetHashCode() => Value;    #A
    public override bool Equals(object obj) {    #A
        return obj is PostId other && other.Value == Value;
    }
    public static bool operator ==(PostId a, PostId b) {    #B
        return a.Equals(b);
    }
    public static bool operator !=(PostId a, PostId b) {    #B
        return !a.Equals(b);
    }
}
```

#A System.Object overrides, using arrow syntax notation
#B Overloading code for equality operators

The arrow syntax

The arrow syntax is introduced to C# in 6.0 and is equivalent to normal method syntax with a single return statement. You can opt for arrow syntax if the code is easier to read that way. It is not right or wrong to use arrow syntax, readable code is right, unreadable code is wrong.

The method

```
public int Sum(int a, int b) {
    return a + b;
}
```

is equivalent to:

```
public int Sum(int a, int b) => a + b;
```

It's not usually needed but in case your class needs to be in a container that is sorted or compared, you have to implement these two additional features too:

1. You need to provide ordering by implementing `IComparable<T>` because equality itself isn't sufficient to determine the order. We didn't use it in the listing 2.1 because identifiers are not ranked.
2. If you plan on comparing values using less than or greater than operators, you have to implement related operator overloads ("<", ">", "<=", ">=") for them too.

This can look like a lot of work when you can simply pass an integer around, but it pays off in large projects, especially when working in a team. You'll see more of the benefits in following sections.

You don't always need to create new types in order to leverage a validation context. You can use inheritance to create base types which contain certain primitive types with common rules. For example, you can have a generic identifier type that can be adapted to other classes. You can simply rename `PostId` class in listing 2.1 to `DbId` and derive all types from it.

Whenever you need a new type like `PostId`, `UserId`, or `TopicId` you can inherit it from `DbId` and extend as needed. Here we can have fully functional varieties of same type of identifier to be able to distinguish them better from other types. You can also add more code in the classes to specialize them in their own way:

```
public class PostId: DbId {    #A
    public PostId(int id): base(id) { }
}
public class TopicId: DbId {    #A
    public TopicId(int id) : base(id) { }
}
public class UserId: DbId {    #A
    public UserId(int id): base(id) { }
}
```

#A We use inheritance to create new flavors of the same type.

Having separate types for your design elements makes it easier to semantically categorize different uses of our DbId type if you're using them together and frequently. It also protects you from passing incorrect type of identifier to a function.

> **RULE OF THUMB** Whenever you see a solution to a problem, make sure that you also know when not to use it. This is no exception. You may not need such elaborate work for your simple prototype, you may not even need a custom class. When you see that you're passing the same kind of value to functions frequently, and you seem to be forgetting if that needed validation or not, it might be beneficial to encompass it in a class and pass it around instead.

Custom data types are powerful as they can explain your design better than primitive types, can help you avoiding repetitive validation therefore bugs. They can be worth the hassle to implement. Moreover, the framework you're using might already be providing the types you need.

2.3.3 Don't framework hard, framework smart

.NET, like many other frameworks, comes with a set of useful abstractions for certain data types which are usually unknown or ignored. Custom text-based values like URLs, IP addresses, file names, or even dates are stored as strings. We'll look at some of those ready-made types and how we can leverage them

Some of you may already know about .NET-based classes regarding those data types, but might still prefer to use a string, because it's simpler to handle. The issue with strings is that they lack proof of validation; your functions don't know if given string is already validated or not, causing either inadvertent failures or unnecessary re-validation code, slowing you down. Using a ready-made class for a specific data type is a better choice in those cases.

When only tool you have is a hammer, every problem looks like a nail. The same applies to strings. Strings are great generalized storage for content, and they are so easy to parse, split, merge, or play around. They are so tempting. But this confidence in the strings makes you inclined to re-invent the wheel occasionally. When you start handling things with a string, you tend to do everything with string processing functions although that can be entirely unnecessary.

Consider this example: you're tasked to write a lookup service for a URL shortening company called Supercalifragilisticexpialidocious which is in financial trouble for unknown reasons, and you're Obi-wan, their only hope. Their service works like this:

- User provides a long URL such as:

```
https://llanfair.com/pwllgw/yngyll/gogerych/wyrndrobwll/llan/tysilio/gogo/goch.html
```

- The service creates a short code for the URL and creates a new short URL such as:

```
https://su.pa/mK61
```

- Whenever user navigates to the shortened URL from their web browser, they get redirected to the address in the long URL they provided.

The function you need to implement must extract the short code from a shortened URL. A string-based approach would look like this:

```
public string GetShortCode(string url)
{
    const string urlValidationPattern =
        @"^https?://([\w-]+.)+[\w-]+(/[\w- ./?%&=])?$";    #A
    if (!Regex.IsMatch(url, urlValidationPattern)) {
        return null;    #C
    }
    // take the part after the last slash
    string[] parts = url.Split('/');
    string lastPart = parts[^1];    #B
    return lastPart;
}
```

#A Regular expression. It's used in string parsing and occult invocation rituals.
#B This is a new syntax introduced in C# 8 which refers to the second last item in a range.
#C Not a valid URL.

This code might look okay at first, but it contains bugs already, based on our hypothetical specification. The validation pattern for URL is incomplete, it allows invalid URLs. It doesn't take the possibility of multiple slashes in the URL path into account. It even unnecessarily creates an array of strings just to get the final portion of URL.

> **NOTE** A bug can only exist against a specification. If you don't have any specification, you cannot claim anything to be a bug. This lets companies to avoid PR scandals by dismissing bugs like "oh, that's a feature". You don't need a written document for a specification either, it can exist in your mind, as long as you can answer the question "is this how this feature is supposed to work?".

More importantly, the logic isn't apparent from the code. A better code might leverage the `Uri` class from .NET framework and look like the example below:

```
public string GetShortCode(Uri url)    #A
{
    string path = url.AbsolutePath;    #B
    if (path.Contains('/')) {
        return null;    #C
    }
    return path;
}
```

#A It's clear what we're expecting.
#B Look ma, no regular expressions!
#C Not a valid URL.

This time, we don't deal with string parsing ourselves. It's been handled already by the time our function gets called. Our code is more descriptive, it is easier to write, only because we just wrote `Uri` instead of `string`. Because parsing and validation happens earlier in the code, it becomes easier to debug too. There is a whole chapter about debugging but the best debugging is not having to debug in the first place.

In addition to primitive data types like `int`, `string`, `float`, and so forth, .NET provides many other useful data types available to use in our code.

`IPAddress` is a better alternative to `string` for storing IP addresses. Not just because it has validation in it, but because it also supports IPv6 which is in use today, unbelievable I know. The class also has shortcut members for defining a local address:

```
var testAddress = IPAddress.Loopback;
```

This way, you avoid writing `127.0.0.1` whenever you need a loopback address, you become faster. In case you make a mistake with the IP address, you catch it earlier than you would with a string.

Another such type is `TimeSpan`. It represents a duration as the name implies. Durations are used almost everywhere in a software project especially when talking about caching or expiration mechanics. We tend to define durations as compile time constants. The worst possible way is this:

```
const int cacheExpiration = 5; // minutes
```

It's not immediately clear that the unit of cache expiration is in minutes. It's impossible to know the unit without looking at the source code. It's a better idea to incorporate it in the name at least, so your colleague, or even yourself in the future, would know its type without looking at the source code:

```
public const int cacheExpirationMinutes = 5;
```

It's better this way but when you need to use the same duration for a different function that receives a different unit, you'll have to convert it, like:

```
cache.Add(key, value, cacheExpirationMinutes * 60);
```

This is extra work. You have to remember to do this. It's prone to errors too. You can mistype `60` and have a wrong value in the end and maybe spend days debugging it or try to optimize performance needlessly because of such a simple miscalculation.

TimeSpan is amazing in that sense. There is no reason for you to represent any duration anything other than in TimeSpan, even when the function you're calling doesn't accept TimeSpan as a parameter.

```
public static readonly TimeSpan cacheExpiration = TimeSpan.FromMinutes(5);
```

Look at that beauty! You already know it's a duration and its unit where it's declared. What's better is that you don't have to know its unit anywhere else. For any function that receives a TimeSpan, you just pass it along. If a function receives a specific unit, say, minutes, as an integer, you can call it like this instead:

```
cache.Add(key, value, cacheExpiration.TotalMinutes);
```

And it gets converted to minutes. Brilliant.

There are many more types that are useful in a similar sense like `DateTimeOffset`, which represents a specific date and time like `DateTime` but along with the time zone information, so you don't lose data when suddenly your computer's or server's time zone information changes. In fact, you should always try to use DateTimeOffset over DateTime as it's also

convertible to/from DateTime easily. You can even use arithmetic operators with TimeSpan and DateTimeOffset, thanks to operator overloading:

```
var now = DateTimeOffset.Now;
var birthDate =
    new DateTimeOffset(1976, 12, 21, 02, 00, 00,
        TimeSpan.FromHours(2));
TimeSpan timePassed = now - birthDate;
Console.WriteLine($"It's been {timePassed.TotalSeconds} seconds since I was born!");
```

> **NOTE** Date and time handling is such a delicate concept and easy to break, especially in global projects. That's why there are separate third-party libraries that cover the missing use cases, such as Noda Time by Jon Skeet.

.NET is like that gold pile that Uncle Scrooge jumps and swims in. It's full of great utilities that make our lives easier. Learning about them might seem wasteful or boring, but it's much faster than trying to use strings or to come up with your own makeshift implementations.

2.3.4 Types over typos

Writing code comments can be a chore, and I argue against doing it later in the book, although you should wait until you read that part before throwing keyboards at me. Even without the code comments, your code doesn't have to lack descriptiveness. Types can help you to explain your code.

Consider you encounter this snippet in the vast dungeons of your project's code base:

```
public int Move(int from, int to) {
    // ... quite a code here
    return 0;
}
```

What is this function doing? What is it moving? What kind of parameters is it taking? What kind of result is it returning? These are all vague without types. You can try to understand the code or try to look up the encompassing class, but they all would take time. Your experience could be much better had the naming been better:

```
public int MoveContents(int fromTopicId, int toTopicId) {
    // ... quite a code here
    return 0;
}
```

It's much better now, but you still have no way to know what kind of result it is returning. Is it an error code, is it number of items moved, or is it the new topic identifier resulting from conflicts in the move operation what is it? How can you convey this without relying on code comments? With types, of course. Consider this code snippet instead:

```
public MoveResult MoveContents(int fromTopicId, int toTopicId) {
    // ... still quite a code here
    return MoveResult.Success;
}
```

It's slightly clearer. I mean it doesn't add much because we already knew that the int was the result of the move function. But there is a difference: we now can explore what's in `MoveResult` type to see what it is actually doing by simply pressing a key, F12 on Visual Studio and VS Code.

```
public enum MoveResult
{
    Success,
    Unauthorized,
    AlreadyMoved
}
```

We've got a much better idea now. Not only does it improve the understanding of the method's API but it also improves the actual code itself in the function too as instead of some constants or worse, hardcoded integer values, you see a clear `MoveResult.Success`. Unlike constants in a class, enums constrain the possible values that can be passed around and they come with own type name so you have a better chance of describing the intent.

Because the function receives integers as parameters, it needs to incorporate some validation since it's a publicly facing API. You can tell that it might even be needed in internal or private code because how validation got pervasive. This would look better if there was a validation logic in the original code:

```
public MoveResult MoveContents(TopicId from, TopicId to) {
    // ... still quite a code here
    return MoveResult.Success;
}
```

As you can see, types can work for you by moving code to their relevant place, and making it easier to understand. Since the compiler checks if you wrote a type's name correctly, they prevent you from having typos too.

2.3.5 To be nullable or non-nullable

In the long run, all developers will encounter `NullReferenceException`. Although Tony Hoare, colloquially known as the inventor of *null*, calls it *the billion dollar mistake* to have created it in the first place, it's not all hopeless.

The brief story of null

Null, or *nil* in some languages, is a value that symbolizes the absence of a value or the apathy of the programmer. It's usually synonymous with the value zero. Since a memory address with the value zero means an invalid region in memory, modern CPU's can catch this invalid access and convert it to a friendly exception message. In the medieval era of computing when null accesses weren't checked, computers used to freeze, get corrupt, or just rebooted.

The problem isn't exactly null itself, we need to describe a missing value in our code anyway. It exists for a purpose. The problem is that all variables can be assigned null *by default*, and never checked if they get assigned to a null value unexpectedly, causing them to be assigned to null at the most unexpected places and crashes in the end. JavaScript, as if it doesn't have enough issues with its type system, has two different nulls: *null* and *undefined*. Null symbolizes missing value while undefined symbolizes missing assignment. I know, it hurts. You must accept JavaScript as it is.

C# 8 introduced a new feature called nullable references. It's a seemingly simple change: references can't be assigned `null` by default. That's it. Nullable references is probably the most significant change in C# language since the introduction of generics. Every other feature about nullable references is related to this core change.

The confusing part about that name is that references were already nullable before C# 8. It should have been called non-nullable references to give programmers a better idea. I understand their logic in naming it, because how they introduced *nullable value types*, but many developers might feel like it doesn't bring anything new to the table.

When all references were nullable, all functions that accepted references could receive two distinct values a valid reference and null. Any function that didn't expect null value would cause a crash when it tried to reference the value.

Making references non-nullable by default changed this. Functions can never receive null anymore as long as calling code also exists in the same project. Consider the following code:

```
public MoveResult MoveContents(TopicId from, TopicId to) {
    if (from is null) {
        throw new ArgumentNullException(nameof(from));
    }
    if (to is null) {
        throw new ArgumentNullException(nameof(to));
    }
    // .. actual code here
    return MoveResult.Success;
}
```

HINT The `is null` syntax in the preceding code might look alien to you. I recently started use it over `x == null` after I read about it in a Twitter discussion by senior Microsoft engineers. Apparently, `is` operator cannot be overloaded, so it's always guaranteed to return the correct result. You can similarly use `x is object` syntax instead of `x != null`. Non-nullable checks eliminate the need for null checks in your code but external code can still call your code with nulls, say if you're publishing a library. In that case, you might still need to perform null checks explicitly.

Why do we check for nulls if the code will crash either way?

If you don't check your arguments for null at the beginning of your function, the function continues to run until it references that null value. That means it can halt in an undesired state, like a half-written record, or may not halt but perform an invalid operation without you noticing. Failing as early as possible and avoiding unhandled state is always a good idea. Crashing isn't something you need to be afraid of, it's an opportunity for you to find bugs.

If you fail early, your stack trace for the exception will look cleaner. You'll know exactly which parameter caused the function to fail.

Not all null values need to be checked either. You might be receiving an optional value, and null is the simplest way to express that intent.

The chapter about error handling will have more details about this.

You can enable null checks project-wide or per file. I always recommend enabling it project-wide for new projects because it encourages you to write correct code from the beginning, so

you spend less time fixing your bugs. To enable it per file you add a line saying `#nullable enable` at the beginning of the file.

> **PRO-TIP** Always end an enable/disable compiler directive with a `restore` counterpart rather than opposite of enable/disable. This way you will not affect the global setting. This helps when you're fiddling with global project settings. You might miss valuable feedback otherwise.

With nullable checks enabled, your code looks like the following:

```
#nullable enable
public MoveResult MoveContents (TopicId from, TopicId to) {
    // .. actual code here
    return MoveResult.Success;
}
#nullable restore
```

When you try to call that function with a null value or a nullable value, you will get a compiler warning right away instead of an error waiting to happen at a random time in production. You'll have identified the error even before trying the code out. You can choose to ignore the warnings and continue but you should never.

Nullable references can be annoying at first though. You cannot easily declare classes like you used to. Consider that we are developing a registration web page for a conference that receives name and email of the recipient and records the results to the DB. Our class has a campaign source field that is a free-form string passed from the advertising network. If the string has no value, it means the page is accessed directly, not referred from an ad. Let's have a class like the following:

```
#nullable enable
class ConferenceRegistration
{
    public string CampaignSource { get; set; }
    public string FirstName { get; set; }
    public string? MiddleName { get; set; }    #A
    public string LastName { get; set; }
    public string Email { get; set; }
    public DateTimeOffset CreatedOn { get; set; }    #B
}
#nullable restore
```

#A Middle name is optional.
#B Having record creation dates in the database is good for auditing.

When you try to compile the class in snippet, you'll receive a compiler warning for all the strings declared non-nullable, that is all the properties except `MiddleName` and `CreatedOn`:

> **Non-nullable property '...' is uninitialized. Consider declaring the property as nullable.**

Middle name is optional, so we declared `MiddleName` as nullable. That's why it didn't get a compiler error.

RULE OF THUMB Never use empty strings to signify optionality. Use null for that purpose. It's impossible for your colleague to understand your intention with an empty string. Are empty strings valid values, or do they indicate optionality? Impossible to tell. Null is unambiguous.

About empty strings

Throughout your career, you will have to declare empty strings for other purposes than optionality. When you need to do that, avoid using the notation "" to signify empty strings. Because of many different environments a code can be viewed, like your text editor, test runner output window, or your continuous integration web page, it's easy to confuse it with a string with a single space in it (" "). Explicitly declare empty strings with `String.Empty`, leverage existing types. You can also use it with lowercase class name `string.Empty`, whichever your code conventions let you do. Let the code convey your intent.

`CreatedOn`, on the other hand, is a struct, so compiler just fills it with zeros, that's why it doesn't throw a compiler error, but still it can be something we want to avoid.

A developer's first reaction to fix a compiler error is to apply whatever suggestion the compiler comes up with. In the example above, that would be to declare the properties as nullable, but that changes our understanding. We suddenly make the properties for first name and last name optional too, which we don't want. We shouldn't be doing that. But we need to think about how we want to apply the optionality semantics.

If you want a property not to be null, you need to ask yourself these questions first:

"Does the property have a default value?"

If it does, you can assign the default value during the construction. That will give you a better idea about the behavior of the class when examining the code. If the field for campaign source has a default value, it can be expressed in code like this:

```
public string CampaignSource { get; set; } = "organic";
public DateTimeOffset CreatedOn { get; set; } = DateTimeOffset.Now;
```

That will remove the compiler warning, and it will convey your intent to whom reads your code.

First name and last name cannot be optional though, and they cannot have default values. No, don't try to put "`John`" and "`Doe`" for default values. Ask yourself this:

"How do I want this class to be initialized?"

If you want your class to be initialized with a custom constructor, so it won't allow invalid values *ever*, you can assign the property values in the constructor and declare them as "`private set`", so they are impossible to change. We will discuss this more in the relevant sections about immutability. You can signify optionality in the constructor with an optional parameter with a default value of null too, as in listing 2.3.

Listing 2.3 A sample immutable class

```
class ConferenceRegistration
{
    public string CampaignSource { get; private set; }    #A
    public string FirstName { get; private set; }    #A
    public string? MiddleName { get; private set; }    #A
    public string LastName { get; private set; }    #A
    public string Email { get; private set; }    #A
    public DateTimeOffset CreatedOn { get; private set; } = DateTime.Now;    #A

    public ConferenceRegistration(
        string firstName,
        string? middleName,
        string lastName,
        string email,
        string? campaignSource = null) {    #B
        FirstName = firstName;
        MiddleName = middleName;
        LastName = lastName;
        Email = email;
        CampaignSource = campaignSource ?? "organic";
    }
}
```

#A All properties are `private set`.
#B Signify optionality with null.

I can hear you whining, "but that's too much work" and I agree. Creating an immutable class shouldn't be this hard. C# team has been working on a new construct called *record types* to make this much easier than it's been, but until then, we have to make a decision: do we want less bugs, or do we want to be done with it as quickly as possible?

Record types to the rescue

C# 9.0 will bring in record types, which makes creating immutable classes extremely easy. The class in listing 2.2 can simply be expressed with a code like this:

```
public record ConferenceRegistration(
  string CampaignSource,
  string FirstName,
  string? MiddleName,
  string LastName,
  string Email,
  DateTimeOffset CreatedOn);
```

It will automatically scaffold properties with the same name of the arguments we specify in the parameter list and it will make the properties immutable, so the record code will behave exactly like the class in listing 2.2. You can also add methods and additional constructors in the body of a record block like a regular class instead of ending the declaration with a semicolon. It's phenomenal. Such a timesaver.

That's a tough decision because us humans are quite terrible at estimating the cost of future events and usually work with the near future. That's the reason I'm able to write this book,

obeying the shelter-in-place order in San Francisco due to COVID-19 pandemic, because humankind has failed to foresee the future costs of a small outbreak in Wuhan, China. We are terrible estimators. Let's accept this fact.

Consider this: you have the chance of eliminating a whole class of bugs caused by missing null checks and incorrect state by simply having this constructor, or you can go ahead and leave it as is and deal with the consequences for every bug filed: bug reports, issue trackers, talking it out with PM, triaging and fixing the relevant bug, only to encounter another bug of the same class until you decide to "ok that's enough, I'll make it how Sedat told me". Which path do you want to choose?

As I said before, this requires some kind of intuition about how much bugs you anticipate on that part of the code. You shouldn't blindly apply suggestions. You should have a sense of the future *churn*, that is amount of change on a piece of code. The more the code changes in the future the more prone it is to bugs.

But let's say you did all that, and decided "nah, that will work okay, not worth the trouble". Then, you can still get some level of null safety with keeping nullable checks in place but initializing your fields beforehand like this:

```
class ConferenceRegistration
{
    public string CampaignSource { get; set; } = "organic";
    public string FirstName { get; set; } = null!;    #A
    public string? MiddleName { get; set; }
    public string LastName { get; set; } = null!;    #A
    public string Email { get; set; } = null!;    #A
    public DateTimeOffset CreatedOn { get; set; }
}
```

#A Notice "null!" as a new construct

The bang operator ("`!`") tells the compiler precisely "I know what I'm doing", in this case "I will make sure that I will initialize those properties right after I create this class. If I don't, I accept that nullability checks won't work for me at all". Basically, you still retain nullability assurances if you keep your promise of initializing those properties right away.

That's a thin ice to cross as it may not be possible to bring everyone in your team to the same page about this, and they might still initialize the properties later. If you think you can manage the risks, you can stick to this. It can even be inevitable for some libraries such as Entity Framework which requires a default constructor and settable properties on objects.

Maybe<T> is dead, long live Nullable<T>!

Because nullable types in C# used to have no compiler support to enforce their correctness, and a mistake would crash the entire program, they were historically seen as an inferior way of signifying optionality. Because of that, people implemented their own optional types, called either `Maybe<T>` or `Option<T>`, without risk of causing null reference exceptions. C# 8.0 makes compiler safety checks for null values first-class, so the era of rolling your own optional type is officially over. The compiler can both check and optimize nullable types better than ad-hoc implementations. You get syntactic support from the language too, such as with operators, and pattern matching. Long live `Nullable<T>`!

Nullability checks help you to think about your intentions on the code you're writing. You will have clearer idea if that value is truly optional, or it isn't needed to be optional at all. It will reduce bugs, make you a better developer.

2.3.6 Better performance for free

Performance shouldn't be your first concern when writing a prototype but having a general understanding of performance characteristics of types, data-structures, and algorithms can correct your heading towards a faster path. You can write faster code without knowing it. Using the specific type for the job instead of more generic ones, can help you behind the scenes.

Existing types can use more efficient storage *for free*. A valid IPv6 string, for instance, can be up to 65 characters. An IPv4 address is at least seven characters long. That means a string-based storage would occupy between 14 and 130 bytes, and when included with object headers, which makes it between 30 and 160 bytes. `IPAddress` type, on the other hand, stores IP address as series of bytes and uses between 20 and 44 bytes.

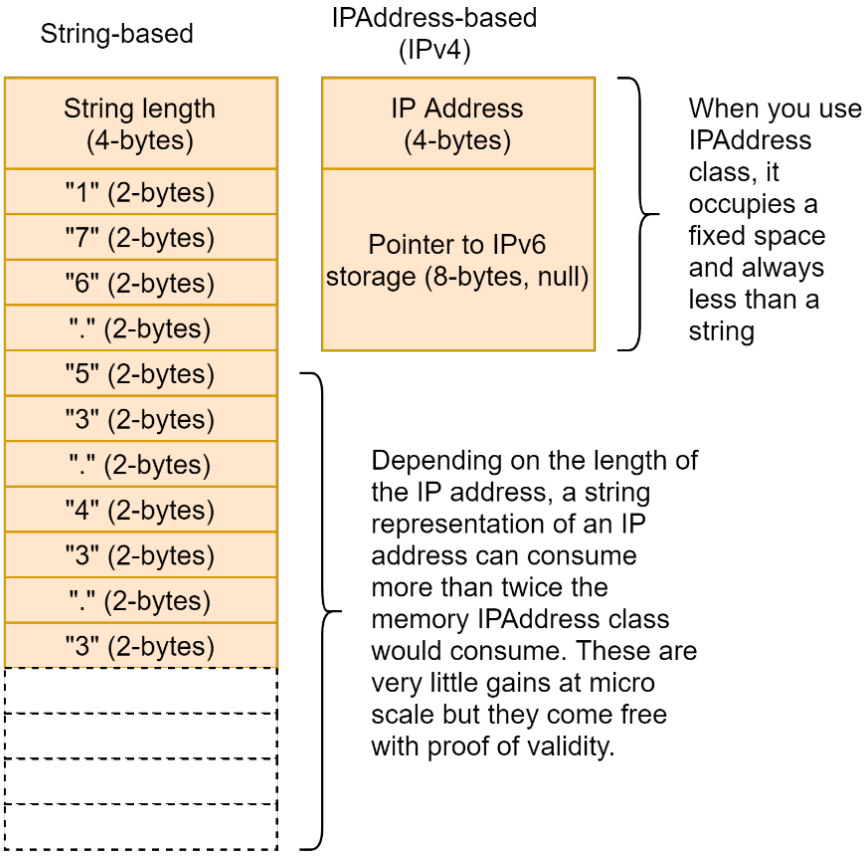


Figure 2.12 Storage differences of data types, excluding common object headers.

It may not look much but remember, this comes for free. The longer the IP address gets, the more space savings you get. It also provides you the proof of validation, you can safely trust that the passed along object holds a valid IP address throughout the code. Your code becomes easier to read because types also describe the intention behind the data.

On the other hand, we all know that there is no free lunch. What's the catch here? When should you not use it? Well, there is a small string parsing overhead for the string in order to deconstruct it into bytes. A code goes over to string to decide if it's an IPv4 or IPv6 address and parses it accordingly with an optimized code. On the other hand, because you'll have the string validated after parsing, it essentially removes the requirement of validation in the rest of your code, compensating for the small parsing overhead. Using the correct type from the get-go lets you avoid the overhead of trying to make sure the passed arguments are the correct type. Last but not least, preferring the correct type can also leverage value types in some cases where they're beneficial. We'll see about the benefits of value types in the next section.

Performance and scalability aren't single dimensional concepts. For example, optimizing data storage can actually lead to worse performance in some cases, as explained in chapter seven. But, with all the given advantages of using the specific type for the job, using a specialized type for data is a no brainer most of the time.

2.3.7 Reference types vs value types

The distinction between reference types and value types is pretty much about how types are stored in memory. In simple terms, the contents of value types are stored in the call stack and reference types are stored in *the heap* and only a reference to their content is stored in the call stack instead. This is a simple example of how they look in the code:

```
int result = 5;    #A
var builder = new StringBuilder();    #B
var date = new DateTime(1984, 10, 9);    #E
string formula = "2 + 2 = ";    #C
builder.Append(formula);
builder.Append(result);
builder.Append(date.ToString());
Console.WriteLine(builder.ToString());    #D
```

#A Primitive value type
#B Reference type
#C Primitive reference type
#D Outputs a mathematical abomination.
#E All structs are value types.

Java doesn't have value types except primitive ones like `int`. C# additionally lets you define your own value types. Knowing the difference between reference and value types can make you a more efficient programmer for free by making you use the correct type for the correct job. It's not something hard to learn either.

A *reference* is analogous to a managed *pointer*. A pointer is an address of memory. I usually imagine memory as an exceptionally long array of bytes:

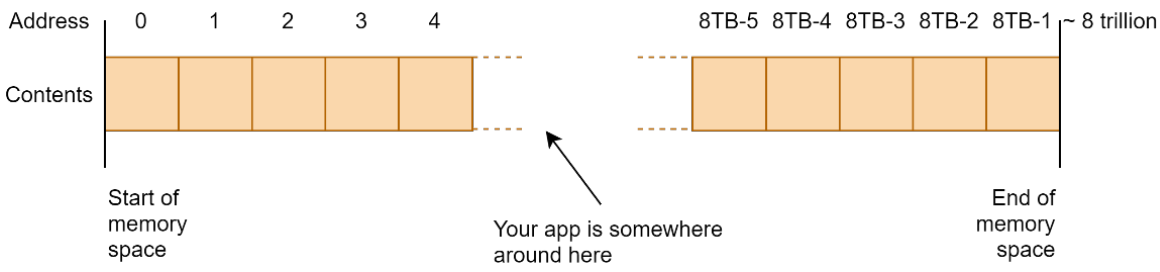


Figure 2.13 Memory layout of a 64-bit process that can address up to 8TB

This isn't all of your RAM; this is just the memory layout of a single process. Contents of your physical RAM looks much more complicated but operating systems hide the fact that RAM is a mess by showing you a tidy, clean, contiguous area of memory for each process, which

may not even exist on your RAM. That's why it's called *virtual memory*. As of 2020, nobody has close to 8TB of RAM on their computers, yet you can access 8 terabytes of memory on a 64-bit operating system. I'm sure somebody in the future looking at this and laughing like I laugh at my old PC with 1-megabyte memory in the 90's.

Why 8TB? I thought 64-bit processors could address 16 exabytes!

They can. The reasons behind limiting user space are mostly practical. Creating virtual memory mapping tables for a smaller memory range consumes less resources, and faster. For example, switching between processes requires memory to be remapped in its entirety. A larger address space would require more memory to be swapped to perform that. It's possible to increase user space range in the future when 8TB RAM becomes common commodity but until then, 8TB is our horizon.

A pointer is basically a number that points to an address in memory. The advantage of using pointers instead of the actual data is to avoid unnecessary duplication which can be quite expensive. We can just pass around gigabytes of data from function to function, by simply passing around its address aka a pointer. Otherwise we would have to copy gigabytes of memory at every function call. We just copy a number instead.

Obviously, it doesn't make sense to use pointers for anything less than the size of the pointer itself. A 32-bit integer (`int` on C#) is just the half the size of a pointer on a 64-bit system. Therefore, primitive types like int, long, bool, and byte are all considered value types. That means instead of a pointer to their address, only their value is passed to functions.

A reference is synonymous with a pointer except that your access to its contents is managed by the .NET runtime. You can't know the value of a reference either. This allows *Garbage Collector* to move the memory pointed by reference around as it needs, without you knowing. You can also use pointers with C#, but that's only possible in an unsafe context.

Garbage collection

A programmer needs to track its allocation of memory and needs to free (or, deallocate) the allocated memory when they are done with it. Otherwise, your application's memory usage constantly increases; also known as a memory leak. Manually allocating and freeing memory is prone to bugs. Programmer might forget to free a memory or even worse, try to free an already freed memory which is root of all many security bugs.

One of the first proposed solutions to the problems with manual memory management was reference counting. Instead of leaving the initiative to free the memory to the programmer, the runtime would keep a secret counter for each allocated object. Every reference to the given object would increment the counter, and every time a variable referencing the object goes out of the scope, the counter would be decremented. Whenever the counter reaches zero, that would mean that there are no variables referencing the object, so it would be freed.

Reference counting works fine for many scenarios, but it has couple of quirks: it's slow because every time a reference goes out of scope, it performs deallocation, which is usually inefficient than, say, freeing relevant blocks of memory together. It also creates a problem with cyclical references which requires extra work and diligence on the programmer's part to avoid.

Garbage collection is a tradeoff between reference counting and manual memory management. With garbage collection, no separate reference counts are kept. Instead, a separate task goes over all the object tree to find objects that are not referenced anymore and marks them as garbage. The garbage is kept for a while, and when it grows beyond a certain threshold, Garbage Collector arrives and frees the unused memory in a single pass. That reduces the overhead of memory deallocation operations and memory fragmentation due to micro deallocations. Not keeping counters makes the code faster too.

C# allows complex value types called *structs*. A struct is remarkably similar to a class in definition but unlike a class, it's passed by value everywhere. That means if you have a struct and you send it to a function, a copy of the struct gets created; and when that function passes it to another function, another copy will be created. *Structs are always copied*. Consider the example in listing 2.4.

Listing 2.4 Immutability example

```
struct Point
{
    public int X;
    public int Y;
    public override string ToString() => $"X:{X},Y:{Y}";
}

static void Main(string[] args) {
    var a = new Point() {
        X = 5,
        Y = 5,
    };
    var b = a;
    b.X = 100;
    b.Y = 200;
    Console.WriteLine(b);
    Console.WriteLine(a);
}
```

What do you think the program above would write to console? When you assign `a` to `b`, the runtime creates a new copy of `a`. That means, when you modify `b`, you're modifying a new struct with `a`'s values, not `a` itself. What if `Point` was a class? Then `b` would have the same reference as `a` and changing the contents of `a` would mean changing `b` at the same time.

Value types exist because there are cases where they can be more efficient than reference types both in terms of storage and performance. We already discussed how a type with a size of a reference or less can be more efficient to be passed by value. Reference types also incur a single level of indirection. Whenever you need to access the field of a reference type, .NET runtime has to read the value of reference first, then go the address pointed by the reference, then read the actual value from there. For a value type, the runtime reads the value directly, making the access faster.

2.4 Summary

- Computer science theory can be boring but knowing some of the theory can make you

better developers.

- Types are normally known as boilerplate in strongly typed languages, but they can be used to write less code too.
- .NET comes with better, more efficient data structures for certain data types that can easily make your code faster and more reliable.
- Using types can make your code more self-explanatory therefore makes write less comments.
- Nullable references feature introduced with C# 8.0 can make your code much more reliable and make you spend less time debugging your application.
- The difference between value types and reference types is significant and knowing about it will make you a more efficient developer.
- Strings are more useful and more efficient if you know how their internals work.
- Arrays are fast and convenient, but they may not be the most suitable candidate for a publicly exposed API.
- Lists are great for growing lists but arrays are more efficient if you don't intend to dynamically grow their contents.
- A linked list is a niche data structure but knowing its characteristics can help you make you understand the tradeoffs of dictionaries.
- Dictionaries are great for fast key lookups, but their performance relies heavily on the correct implementation of `GetHashCode()`.
- A list of unique values can be represented with a HashSet instead for awesome lookup performance.
- Stacks are great data structures for retracing your steps. The call stack is finite.
- Knowing how call stack works also complements the performance implications of value and reference types.

3

Useful anti-patterns

This chapter covers

- **Known bad practices that can be put to good use**
- **Anti-patterns that are, in fact, useful**
- **Identifying when to use a best practice versus its evil twin**

Programming literature is full of best practices and design patterns. Some of them are even seemingly indisputable and cause people to give you the side-eye if you argue about them. They eventually turn into dogmas and are rarely questioned. Once in a while, someone writes a blog post about it, and if their article gets the approval of The Hacker News[1] community, it can be accepted as valid criticism and can open a door for new ideas. Otherwise, you can't even argue about them. If I had to leave a single message to the world of programming, it would be to question everything that's taught to you—their usefulness, their reason, their gain, and their cost.

Dogmas, immutable laws create blind spots for us, and their size grows as we stick to them longer. Those blind spots can cloak some useful techniques where some can even be better for certain use cases.

Anti-patterns, or "bad practices" if you will, get a bad rap, deservedly so, but that doesn't mean we should avoid them like radioactive material. I'll be going over some of those patterns that can help you more than their best practice counterparts. This way, you'll also be using the best practices and great design patterns with better understanding of how they help, and when they aren't helpful. You'll see what you're missing in your blind spot and what kind of gems are there.

[1] Hacker News is a tech news sharing platform where everyone is an expert about everything, https://news.ycombinator.com

3.1 If it ain't broke, break it

One of the first things I learned at the former companies that I worked at was, first, where the restrooms were, and then, to avoid changing the code, aka *code churn*, at all costs. Every change you make carries the risk of creating a *regression*, which is a bug that breaks an already working scenario. Bugs are already costly and fixing them takes time when they are part of a new feature. When it's a regression, that's worse than releasing a new feature with bugs, it's a step back. Missing a shot in basketball is a bug. Scoring a goal on your own hoop, effectively scoring for your opponent, is a regression. Time is the most critical resource in software development and losing time has the utmost severity. Regressions lose you the most time. It makes sense to avoid regressions and to avoid breaking the code.

Avoiding changes can lead to a conundrum eventually though, because if a new feature requires something to be broken and made again, it might cause resistance to its development. You can get inclined to tiptoe around existing code and try to add everything in new code without touching existing code. Your effort to leave the code untouched can force you to create more code, which just increases the amount of code to maintain.

If you have to change existing code, that's a bigger problem. There is no tiptoeing around this time. It can be awfully hard to modify existing code because it is tightly coupled to a certain way of doing things and changing it will cause you to change many other places. This resistance of existing code to change is called *code rigidity*. That means the more rigid the code gets, the more of the code you have to break in order to manipulate it.

3.1.1 Facing code rigidity

Code rigidity is based on multiple factors, and one of them is too many dependencies in the code. *Dependency* can mean multiple things: it can be a reference to a framework assembly, to an external library, or to another entity in your own code. All types of dependency can create problems if your code gets tangled up in them. Dependency can be both a blessing and a curse. Figure 3.1 depicts a software with a terrible dependency graph; it that violates the concern boundaries, and any break in one of the components would require changes in almost all of the code.

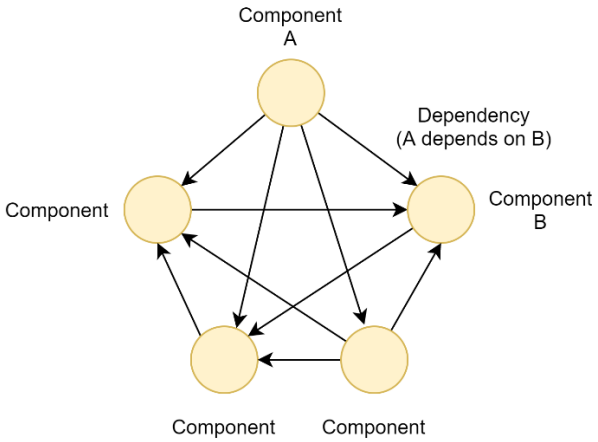


Figure 3.1 The occult symbol for dependency hell

Why do dependencies cause problems? When adding dependencies, consider every component as a different customer or every layer as a different market segment with different needs. Serving multiple segments of customers is a greater responsibility than serving only a single type of customer. Customers have different needs, which might force you to cater to different needs, unnecessarily. Think of these relationships when deciding on dependency chains. Ideally, try to serve as few types of customers as possible. This is the key to keeping your component, or your entire layer as simple as possible.

We can't avoid dependencies. They are essential for reusing code. Code reuse is a two-clause contract. If component A depends on component B, the first clause is "B will provide services to A." There is also a second clause which is often overlooked. It is, "A will go through maintenance whenever B introduces a breaking change." Dependencies caused by code reuse is okay as long as you can keep the dependency chain organized and compartmentalized.

3.1.2 Move fast, break things

Why do you need to break that code, as in making it not even compile or fail the tests? Because intertwined dependencies cause rigidity in the code, make the code resistant to change. It's is a steep hill that will get you slower over time, bringing you to a halt eventually. It's easier to handle breaks at the beginning, that's why you need to identify these issues and break your code, even when it's working. You can see how dependencies force our hand in figure 3.2:

- A component with zero dependencies is the easiest to change. It's impossible to break anything else. If your component depends on one of your other components that creates some rigidity because dependency implies a contract.
- If you change the interface on B, that means you need to change A too. If you change the implementation of B without changing the interface, you can still break A because

you break B. That becomes a bigger issue when you have multiple components depend on a single component.

- Changing A becomes harder because it needs change in the dependent component and incurs a risk of breaking any of them. Programmers tend to assume the more they reuse code the more time they save. But at what cost? You need to consider this.

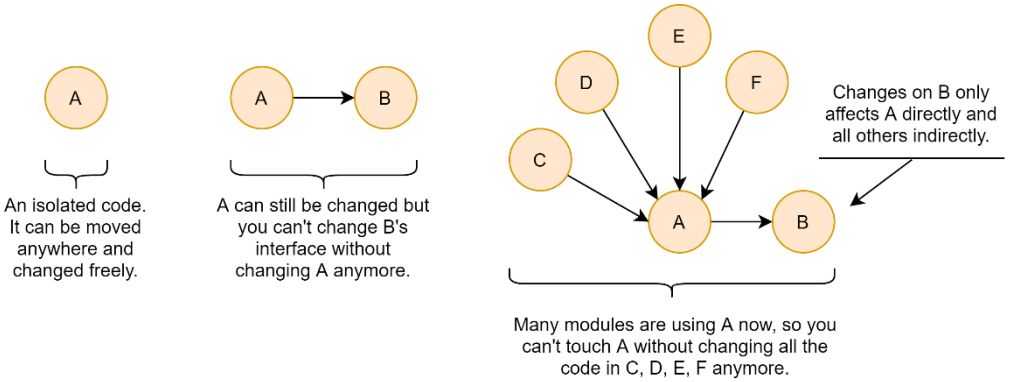


Figure 3.2 Resistance to change is proportional to dependencies.

3.1.3 Respecting boundaries

First thing you must adopt is to avoid violating *abstraction boundaries* for dependencies. An abstraction boundary is the logical borders you draw around layers of your code, a set of the concerns of a given layer. For example, you can have web, business, and database layers in your code as abstractions. When you layer code like that, the database layer shouldn't know about web layer or business layer and web layer shouldn't know about the database, as seen in figure 3.3.

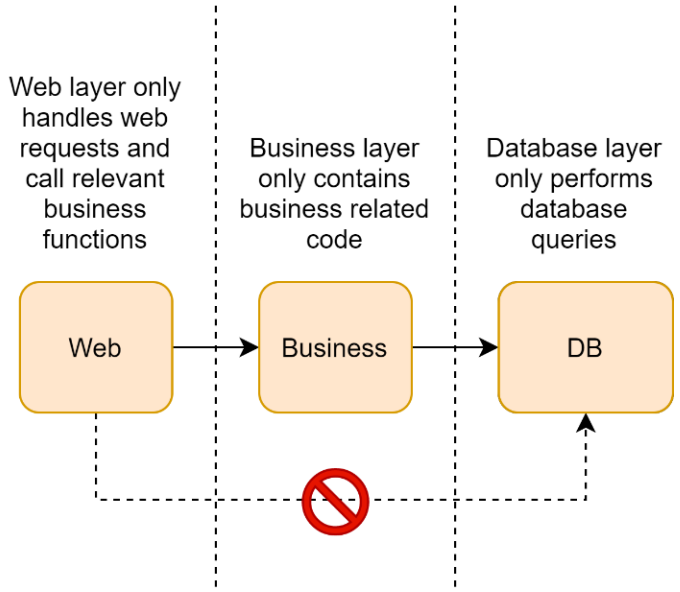


Figure 3.3 Violation of abstraction that you need to avoid

Why is stepping over boundaries a bad idea? Because it eliminates the benefits of an abstraction. When you pull complexity of lower layers into higher layers, you become responsible to maintain the impact of the changes on the lower layers to everywhere. Think about a team whose members are responsible for their own layers. Suddenly, the developer of the web layer needs to learn SQL. Not only that, but the changes in the DB layer also need to be communicated now with more people than necessary. It burdens the developer with unnecessary responsibilities. The time to reach a consensus increases exponentially to the people who need to be convinced. You lose time, and you lose the value of abstractions.

If you bump into such boundary issues, break the code, as in deconstruct it so it might stop working, remove the violation, refactor the code, and deal with the fallout. Fix other parts of the code that depend on it. You have to be vigilant to such issues and immediately cut them off, risking breaking the code. If the code makes you afraid to break it, it's badly designed code. That doesn't mean good code doesn't break, but when it does, it's much easier to glue the pieces back together.

The importance of tests

You need to have a way to see if a change in code would cause a scenario to fail or not. You can rely on your own understanding of code for that, but your effectiveness will diminish as the code gets more complex over time. Tests are simpler in that sense. Tests can be a list of instructions on a piece of paper, or they can be fully automated tests. Automated tests are usually more preferable because you write them only once and don't waste your time

executing them yourself. Thanks to testing frameworks, writing them is quite straightforward too. We'll delve more into this subject in the chapter about testing.

3.1.4 Isolating common functionality

Does this all mean the web layer in figure 3.3 can't ever have common functionality with the DB? It can, of course. But such cases are an indication of a need for a separate component. For instance, both layers can rely on the common model classes. In that case you'd have a relationship diagram like that shown in figure 3.4.

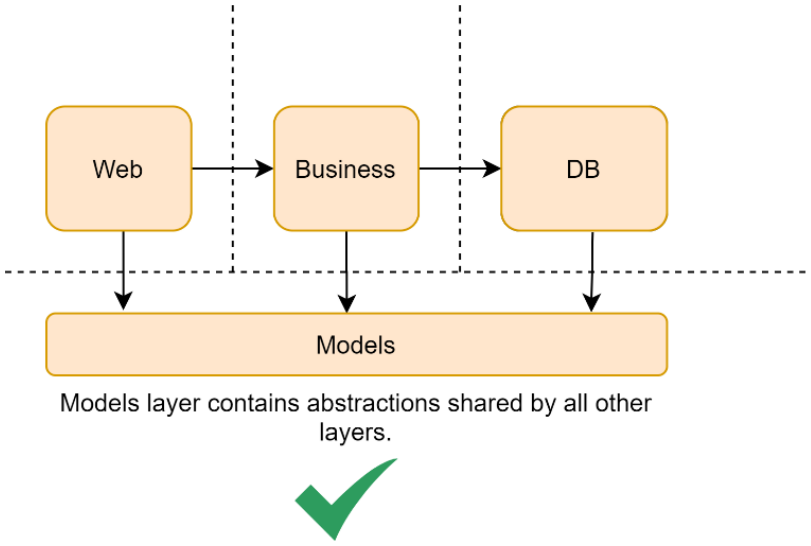


Figure 3.4 Extracting common functionality without violating abstractions

Refactoring code can break your build process or make your tests fail, and theoretically it's something you should never do. But I regard such violations as hidden breaks. They need immediate attention, and if they cause more breakage and more bugs in the process, that doesn't mean you caused the code to stop working: it means the bug that was already there now manifests itself in a way that is easier to reason about.

Let's look at an example. Consider that you're writing an API for a chat app where you can communicate only in emojis. Yes, it sounds horrible, but there was once a chat app in which you could send only "Yo" as a message.[2] Ours is an improvement, if nothing else.

We design the app with a web layer that accepts requests from mobile devices and calls the *business* layer, (aka *logic layer*), that performs the actual operations. This kind of separation allows us to test the business layer without a web layer. We can also later use the

[2] The chat app called "Yo" in which you could only send a text containing "Yo" was once valued ten million dollars. The company got shut down in 2016. https://en.wikipedia.org/wiki/Yo_(app)

same business logic in other platforms too, such as a mobile web site. So, separating business logic makes sense.

> **NOTE** *Business* in business logic or business layer doesn't necessarily mean something related to a business, but more like the core logic of the application with abstract models. Arguably, reading business-layer code should give you an idea about how the application works in higher-level terms.

A business layer doesn't know anything about databases or storage techniques. It calls the database layer for that. The database layer encapsulates the database functionality in a DB-agnostic fashion. This kind of separation of concerns can make testability of business logic easier because we can easily plug a mocked implementation of the storage layer to the business layer. More importantly, that architecture allows us to change a DB behind the scenes without changing a single line of code in the business layer or web layer, for that matter.

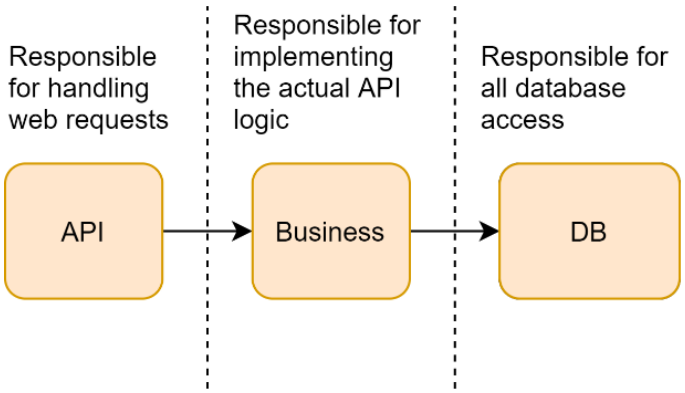


Figure 3.5 Basic architecture of our mobile app API

The downside is, every time you add a new feature to the API, you need to create a new business layer class or method, and relevant database layer class and methods. This seems like a lot of work, especially when the deadlines are tight, and the feature is somewhat simple. "A simple SQL query, why do I need to go through all this hassle?" you might think. Let's go ahead and fulfill the fantasy of many developers and violate the existing abstractions.

3.1.5 Example web page

Suppose you receive a request to implement a new feature from your manager, a new statistics tab that shows how many messages the user sent and received in total. It's just two simple SQL queries on the backend:

```
SELECT COUNT(*) as Sent FROM Messages WHERE FromId=@userId
SELECT COUNT(*) as Received FROM Messages WHERE ToId=@userId
```

You can run these queries in your API layer. Even if you're not familiar with ASP.NET Core, web development, or SQL for that matter, you should have no problems understanding the gist of the code in listing 3.1, which defines a model to return to the mobile app. The model is then automatically serialized into JSON. We retrieve a connection string to our SQL Server database. We use that string to open a connection, run our queries against the database, and return the results.

`StatsController` class in listing 3.1 is an abstraction over web handling wherein received query parameters are in function arguments and the URL is defined by the name of the controller and the result is returned as an object. So, you would reach to the code in listing 3.1 with a URL like https://yourwebdomain/Stats/Get?userId=123 and the MVC infrastructure maps the query parameters into function parameters and the returned object to a JSON result automatically. It makes writing web handling code simpler as you don't really have to deal with URLs, query strings, HTTP headers, and JSON serialization yourself.

Listing 3.1 Implementing a feature by violating abstractions

```
public class UserStats {    #A
  public int Received { get; set; }
  public int Sent { get; set; }
}

public class StatsController: ControllerBase {    #B
  public UserStats Get(int userId) {    #C
    var result = new UserStats();
    string connectionString = config.GetConnectionString("DB");
    using (var conn = new SqlConnection(connectionString)) {
      conn.Open();
      var cmd = conn.CreateCommand();
      cmd.CommandText =
        "SELECT COUNT(*) FROM Messages WHERE FromId={0}";
      cmd.Parameters.Add(userId);
      result.Sent = (int)cmd.ExecuteScalar();
      cmd.CommandText =
        "SELECT COUNT(*) FROM Messages WHERE ToId={0}";
      result.Received = (int)cmd.ExecuteScalar();
    }
    return result;
  }
}
```

#A Defines the model
#B Our controller
#C Our API endpoint

I spent probably five minutes writing this implementation. It looks straightforward. Why do we bother with abstractions? Just put everything in API layer, right?

Such solutions can be okay when working on prototypes. You shouldn't try to work out the perfect design for a prototype. But in a production system, you need to be careful making such decisions. Are you allowed to break production? Is it okay if the site goes down for a couple of minutes? If these are okay, then feel free to use this. How about your team? Is the maintainer of the API layer okay with these SQL queries all around the place? How about testing? How do you test this code and make sure that it runs correctly? How about

new fields added to this? Try to imagine the office the next day. How do you see the people treating you? Do they hug you? Cheer you? Or do you find your desk and your chair decorated with tacks?

You added a dependency to the physical DB structure. If you need to change the layout of `Messages` table, or the DB technology you used, you'll have to go around all the code and make sure that everything works with the new DB or the new table layout.

3.1.6 Leave no debt behind

We programmers are not good at predicting future events and their cost. When we make certain unfavorable decisions just for the sake of meeting a deadline, we make it even harder to meet the next one because of the mess we created. That's commonly called *technical debt* among programmers.

Technical debts are conscious decisions. The unconscious ones are called technical ineptitude. The reason they are called debts is because you either pay it back later, or the code will come looking for you in an unforeseen future and break your legs with a tire iron.

There are many ways a technical debt can accumulate. It might look easier just pass an arbitrary value instead of having the trouble to creating o constant for it. "A string seems to work fine there", "No harm will come from shortening a name", "Let me just copy everything and change some of its parts", "I know, I'll just use regular expressions". Every small bad decision will add seconds to your and your team's performance. Your throughput will degrade over time cumulatively. You will get slower and slower, getting less satisfaction from your work, getting less positive feedback from the management. By being the wrong kind of lazy, you are dooming yourself for failure. Be the right kind of lazy: serve your future lazy.

The best way to deal with technical debt is to procrastinate with it. You have a larger job ahead of you? Use this as an opportunity to get yourself warmed up. It might break the code. That's good, use it as an opportunity to identify rigid parts of the code, get them granular, flexible. Try to tackle it, change it, and then if you think it doesn't work well enough, undo all your changes.

3.2 Write it from scratch

If changing code is risky, writing it from scratch must be orders of magnitude riskier. It essentially means any untested scenario might be broken. Not only does it mean writing everything from scratch but fixing all the bugs from scratch too. It's regarded as a seriously cost-inefficient method for fixing design deficiencies.

That's only true for already working code though. For code that you already have been working on, starting anew can be a blessing. How, you might ask? It's all related to the spiral of desperation when writing new code. It goes like this:

1. You start with a simple and elegant design.
2. You start writing code.
3. Then, some edge cases that you didn't think of appear.
4. You start revising your design.
5. Then you notice that the current design doesn't work for the requirements.

6. You start tweaking the design again, but you avoid redoing it, because it would cause too many changes. Every line adds to your shame.
7. Your design is now a Frankenstein's monster of ideas and code mashed together. Elegance is lost, simplicity is lost, and all hope is lost.

At that point, you've entered a loop of sunk-cost fallacy. The time you spent already with your existing code makes you averse to redoing it. But because it can't solve the main issues, you spend days trying to convince yourself that the design might work. Maybe you fix it at some point but it might lose you weeks, just because you dug yourself into a hole.

3.2.1 Erase and rewrite

I say, *start from scratch*: rewrite it. Toss away everything you already did, write every bit from scratch. You can't believe how refreshing and quick that will be. You might think that writing it from scratch would be hugely inefficient, and you'd be spending double the time but it's not the case because you've already done it once. You already know your way around the problem.

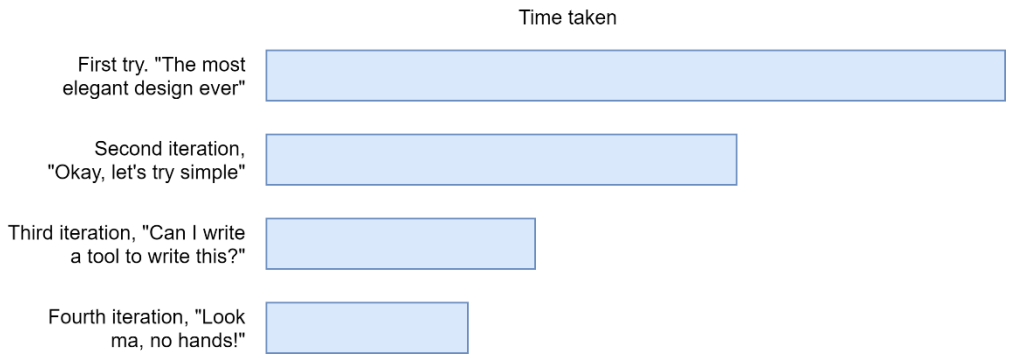


Figure 3.6 The brilliance of doing something over and over and expecting the same results

It's hard to overstate the gains in speed when doing something the second time. Unlike hackers depicted in movies, most of your time is spent looking at the screen, not writing stuff but thinking about things, considering the right way of doing things. Programming isn't about crafting things as much as navigating a maze of a complex decision tree. When you start the maze from the beginning again, you already know possible mishaps, familiar pitfalls, certain designs you've reached in your previous attempt.

If you feel stuck developing something new, write it from scratch. I'd say don't even save the previous copy of your work, but you might want to in case you're not really sure if you can do it again really fast. Okay, save a copy somewhere, but I assure you, most of the time, you won't even need to look at your previous work. It's already in your mind, guiding you much faster, and without going into the same spiral of desperation this time.

More importantly, you'll know if you're following the wrong path much earlier in your process than you previously did when you start from scratch. Your pitfall radar will come

installed this time. You'll have developed an innate sense of developing that certain feature the right way. Programming this way is a lot like playing console games like Marvel's Spider-Man or The Last Of Us. You die constantly and start that sequence again. You die, you respawn. You become better with this repetition, and the more you repeat, you become better at programming. Doing it from scratch improves how you develop that single feature, yes, but it also improves your development skills in general for all the future code you will be writing.

Don't hesitate to throw your work away and write it from scratch. Don't fall for the sunk-cost fallacy.

3.3 Fix it, even if it ain't broke

There are ways to deal with code rigidity, and one of them is to keep the code churning so it doesn't solidify as far as the analogy goes. Good code should be easy to change, it shouldn't give you a list of a thousand places that you need to change in order to make the change you need. There are certain changes that can be performed on code that aren't necessary but can help you in the long term. You can make it a regular habit to keep your dependencies up to date to keep your app fluid, and identify the most rigid parts that are hard to change. You can also improve the code as a *gardening activity*, taking care of the small issues in the code regularly.

3.3.1 Race toward the future

You'll inevitably be using one or more packages from the package ecosystem and you'll leave them as is as they keep working for you. The problem with this is that when you need to use another package and it requires a later version of your package, the upgrade process can be much more painful than gradually upgrading your packages and staying current. You can see such a conflict in figure 3.7.

Most of the time, package maintainers only think about the upgrade scenarios between two major versions, rather than multiple versions in-between. For example, the popular ElasticSearch search library requires major version upgrades to be performed one by one; it doesn't support upgrading from one version to another directly.

.NET supports *binding redirects* to avoid the problem of multiple versions of the same package to a certain degree. A binding redirect is a directive in application configuration that causes .NET to forward calls to an older version of an assembly to its newer version, or vice versa. This only works when both packages are compatible of course. You don't normally need to deal with binding redirects yourself because Visual Studio can do that for you if you have already selected "Automatically generate binding redirects" in project properties screen.

Keeping your packages up to date periodically will have two important benefits. First, you'll have spread the effort of upgrading to the current version over the maintenance period. Every step will be less painful. Second, and more importantly, every minor upgrade might break your code or your design in small or subtle ways which you need to fix to move to the future. This may sound undesirable, but it will make you improve the code and design in small steps as long as you have tests in place.

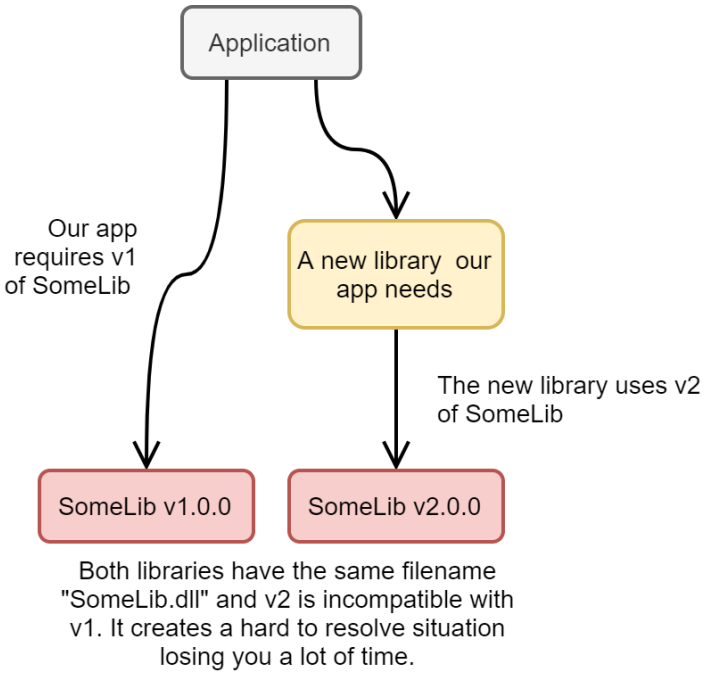


Figure 3.7 Unfixable version conflicts

You might have a web application that uses ElasticSearch for search operations and Newtonsoft.Json for parsing and producing JSON. They are among the most common libraries out there. The problem starts when you need to upgrade Newtonsoft.Json package to use a new feature, but ElasticSearch uses the old one. But, to upgrade ElasticSearch, you need to change the code that handles ElasticSearch too. What do you do?

Most packages only support single version upgrades. ElasticSearch, for example, expects you to upgrade from 5 to 6, and it has guidelines on how to do that. It doesn't have guidelines for upgrading from 5 to 7. You'll have to apply each individual upgrade step separately. Some upgrades require you to change code significantly too. ElasticSearch 7 almost makes you write the code from scratch.

You might as well stay in the older versions under the safety of unchanged code, but not only does the support for older versions end at some point, the documentation and code examples don't stay around forever either. StackOverflow gets filled with the answers about the newer versions because people use the latest version when starting a new project. Your support network for the older version fades out over time. That makes it even harder to upgrade every passing year which pushes you a downward spiral of desperation.

My solution to this problem is to join the race toward the future. Keep the libraries up to date. Make it a regular habit to upgrade libraries. This will break your code occasionally and thanks to that, you'll find out which part of your code is more fragile, and you can add more test coverage.

The key idea in this is that upgrades may cause your code to break but letting them have micro-breaks will prevent huge roadblocks that have had become really hard to tackle. You are not only investing into a fictional future gain either, you are also investing in the flexing of dependencies of your app, letting it break and mend it in a way so it doesn't break as easily in the next change, regardless of package upgrades. The less resistant your app is to change, the better it is in terms of design and ease of maintenance.

3.3.2 Cleanliness is next to codeliness

What I liked first about computers was their determinism. What you wrote would happen the same way all the time, guaranteed. Code that's working would stay working. I found comfort in that. How naïve of me. In my career, I've seen many instances of bugs that could only be observed occasionally based on speed of your CPU, or the time of the day. The first truth of the streets can be summarized as "everything changes." Your code will change. Requirements will change. Documentation will change. The environment will change. It's impossible for you to keep running code stable just by not touching the code.

Since we got this out of the way, we can relax and say that it's okay to touch code. We shouldn't be afraid of change because it will happen anyway. That means that you shouldn't hesitate to improve working code. Improvements can be small: adding some necessary comments, removing some unnecessary ones, naming things better. Keep the code alive. The more change you make on a code, it becomes less resistant to future change. That's because changes will cause breaks and breaks will let you identify weak parts and make them more manageable. You should develop an understanding of how your code breaks, where it breaks, how it breaks. Eventually, you'll have an innate sense what kind of change would be the least risky.

You can call this kind of code-improvement activity *gardening*. You are not necessarily adding features or fixing bugs, but the code should be slightly improved when you're done with it. Such a change can let the next developer who visits the code understand it better or improve the test coverage on the code, as if Santa left some gifts overnight, or the bonsai at the office was mysteriously alive.

Why should you bother doing a chore that will never be recognized by anyone in your career though? Ideally, it should be recognized and rewarded, but that may not be always the case. You can even get some backlash from your peers because they may not like the change you made. You can even break their workflow without breaking the code. You can turn it to a worse design than what the original developer intended while trying to improve it.

Yes, and that's expected. The only way to get mature about how to handle code is to change lots of it. Make sure that your changes are easily reversible so in case you upset someone you can take your changes back. You will also learn how to communicate with your peers on changes that might impact them. Good communication is the greatest skill you can improve in software development.

The greatest benefit of trivial code improvements is that it puts you into the programming state of mind very quickly. Large work items are the heaviest mental dumbbells. You usually don't know where to start, and how to handle such a large change. The pessimism in the form of "oh, that will be so hard to do, I will suffer through this" makes

you postpone to starting the project. The more you postpone it, the more you dread coding it.

Doing minor improvements on code is a trick to get your mental wheels turning so you can warm up enough to tackle a larger problem. Because you're already coding, your brain resists less to switching gears up than trying to switch to coding from browsing social media. Relevant cognitive parts will have already been fired and ready for a larger project.

If you can't find anything to improve, you can get help from code analyzers. They are great tools for finding out minor issues in the code. Make sure you customize the options of the code analyzer you use to avoid offending people as much as possible. Talk to your peers about what they think about it. If they think that they can't bother to fix the issues, promise them to fix the first batch yourself and use that as an opportunity to fix. Otherwise, you can use a command-line alternative or Visual Studio's own code analysis features to run code analysis without violating team's coding guidelines.

You don't even have to apply the changes you make as they are only for warming up to you to coding. For example, you may not be sure if you can apply a certain fix, it might look risky, but you have already done so much. But as you learned, throw it away. You can always start from scratch and do it again. Don't worry much about throwing away your work. If you are keen on it, keep a backup if you'd like but I wouldn't really worry about it.

If you know that your team will be okay with the changes you made, publish them. The satisfaction of improvement, however small, can be motivating enough for you to make larger changes.

3.4 Do repeat yourself

Repetition and *copy-paste programming* are concepts that are looked down in the circles of software development. Like every sane recommendation, it's been eventually turned into a religion, causing people to suffer.

The theory goes like this: You write a piece of code. You need the same piece of code somewhere else in the code. A beginner's inclination would be just copying and pasting the same code and use it. It's all good so far. Then you find a bug in the copy-pasted code. Now, you need to change the code in two different places. You need to keep them in-sync. That will create more work and cause you to miss deadlines.

It makes sense, right? The solution to the problem is usually to put the code in a shared class or module and use it in both parts of the code instead. So, when you change the shared code, you would be changing it magically in everywhere it's referenced, saving you great deal of time.

It's all good so far, but it doesn't last forever. The problems begin to appear when you apply this principle to everything imaginable, and blindly too. One minor detail you miss when you try to refactor code into reusable classes is that you are inherently creating new dependencies and dependencies influence your design, sometimes they can even force your hand.

The biggest problem with shared dependencies is that the parts of software that use the shared code can diverge in their requirements. When this happens, a developer's reflex is to cater to different needs in the same code. That means adding optional parameters, conditional logic to make sure that the shared code can serve two different requirements.

This makes the actual code more complicated than it is, eventually causing more problems than it solves. At some point, you start thinking about a more complicated design than copy-pasted code.

Consider an example where you are tasked to write an API for an online shopping web site. The client needs to change the shipping address for the customer, which is represented by a class called `PostalAddress` like this:

```
public class PostalAddress {
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public string Address1 { get; set; }
  public string Address2 { get; set; }
  public string City { get; set; }
  public string ZipCode { get; set; }
  public string Notes { get; set; }
}
```

and you need to apply some *normalization* to the fields, such as capitalization, so they look decent even when the user doesn't provide the correct input. An update function might look like a sequence of normalization operation and the update on the database:

```
    public void SetShippingAddress(Guid customerId,
      PostalAddress newAddress) {
      normalizeFields(newAddress);
      db.UpdateShippingAddress(customerId, newAddress);
    }

    private void normalizeFields(PostalAddress address) {
      address.FirstName = TextHelper.Capitalize(address.FirstName);
      address.LastName = TextHelper.Capitalize(address.LastName);
      address.Notes = TextHelper.Capitalize(address.Notes);
    }
```

Our capitalize method would work by making the first character upper case and the rest of the string lower case:

```
public static string Capitalize(string text) {
  if (text.Length < 2) {
    return text.ToUpper();
  }
  return Char.ToUpper(text[0]) + text.Substring(1).ToLower();
}
```

Now, this seems to work for shipping notes and names: "`gunyuz kapanoglu`" becomes "`Gunyuz Kapanoglu`" and "`PLEASE LEAVE IT AT THE DOOR`" becomes "`Please leave it at the door`," saving the delivery person unwanted anxiety. After you run your application for a while, you want to normalize city names too. You add it to the `normalizeFields` function:

```
address.City = TextHelper.Capitalize(address.City);
```

It's all good so far but when you start to receive orders from San Francisco, you notice that they are normalized to "`San francisco`" instead. Now you have to change the logic of your capitalization function so it capitalizes every word so the city name becomes "`San`

`Francisco`". It will also help with the names of Elon Musk's kids. But then you notice the delivery note becomes, "`Please Leave It At The Door`" instead. It's better than all uppercase but the boss wants it perfect. What do you do?

The easiest change that touches the least code might seem like to change `Capitalize` function, so it receives an additional parameter about the behavior. The code below receives an additional parameter called `everyWord` that specifies if it's supposed to capitalize every word or only the first word. Please note that we didn't name the parameter `isCity` or something like that because what we're using it for isn't the problem of the `Capitalize` function. Names should explain things in the terms of the context they are in, not the caller's. Anyway, we split the text into words if `everyWord` is true and capitalize each word individually by calling ourselves for each word and then join the words back into a new string in listing 3.2.

Listing 3.2 Initial implementation of `Capitalize` function

```
public static string Capitalize(string text,
  bool everyWord = false) {    #A
  if (text.Length < 2) {
    return text;
  }
  if (!everyWord) {    #B
    return Char.ToUpper(text[0]) + text.Substring(1).ToLower();
  }    #B
  string[] words = text.Split(' ');    #C
  for (int i = 0; i < words.Length; i++) {    #C
    words[i] = Capitalize(words[i]);    #C
  }    #C
  return String.Join(" ", words);    #C
}
```

#A Newly introduced parameter.
#B The case that handles only the first letter.
#C Capitalizes every word by calling the same function.

It already started to look complicated, but bear with me, I really want you to get convinced on this. Changing the behavior of the function seems like the simplest solution. You just add a parameter and `if` statements here and there and there you go. This creates a bad habit, almost a reflex, to handle every small change this way and can create enormous amount of complexity.

Let's say you also need capitalization for file names to download in your app, and you already have a function that corrects letter cases, you just need the file names turned into capitalized and separated with an underscore. For example, if the API received "invoice report," it should turn into `Invoice_Report`. Because you already have a capitalize function, your first instinct will be to modify its behavior slightly again. We add a new parameter called `filename` because the behavior we are adding doesn't have a more generic name to it and check the parameter at the places where it matters. When converting to upper and lower case, we must use culture invariant versions of `ToUpper` and `ToLower` functions so the filenames on Turkish computers don't suddenly become `İnvoice_Report` instead. Notice the dotted I in "İnvoice_Report"? Our implementation would look like this now:

Listing 3.3 A Swiss-army knife function that can do anything

```
public static string Capitalize(string text,
    bool everyWord = false, bool filename = false) {    #A
    if (text.Length < 2) {
      return text;
    }
    if (!everyWord) {
      if (filename) {    #B
        return Char.ToUpperInvariant(text[0])
          + text.Substring(1).ToLowerInvariant();
      }
      return Char.ToUpper(text[0]) + text.Substring(1).ToLower();
    }
    string[] words = text.Split(' ');
    for (int i = 0; i < words.Length; i++) {
      words[i] = Capitalize(words[i]);
    }
    string separator = " ";
    if (filename) {
      separator = "_";    #B
    }
    return String.Join(separator, words);
  }
```

#A Our new parameter.
#B Filename-specific code.

Look at what kind of monster we've created. We violated our principle of cross-cutting concerns and made our Capitalize function aware of our file naming conventions. It suddenly became part of a specific business logic, rather than staying generic. Yes, we are reusing code as much as possible, but we are making our job in the future really hard.

Notice that you also created a new case that isn't even in your design: a new filename format where not all words are capitalized; it's exposed through the condition where `everyWord` is `false` and `filename` is `true`. You didn't intend it but now you have it. Another developer might rely on the behavior and that's how your code becomes a spaghetti over time.

I propose a cleaner approach: *repeat yourself*. Instead of trying to merge every single bit of logic into the same code, try to have separate ones with, maybe slightly repetitive code. You can have separate functions for each use case. You can have one that capitalizes only the first letter, you can have another one that capitalizes every word, and you can have another one that actually formats a file name. They don't even have to reside next to each other, the code about file name can stay closer to the business logic it's required for. You instead have these three functions that convey their intent much better. The first one is named `CapitalizeFirstLetter` so its function is clearer. The second one is `CapitalizeEveryWord`, and also explains what it does better. It calls `CapitalizeFirstLetter` for every word, which is much easier to understand than trying to reason about recursion. Finally, we have `FormatFilename`, which has an entirely different name because capitalization isn't the only thing it does. It has the all capitalization logic implemented from scratch. This lets us freely modify the function when our file name

formatting conventions change without needing to think that it would impact our capitalization work:

Listing 3.4 Repeated work with much better readability and flexibility

```
public static string CapitalizeFirstLetter(string text) {
  if (text.Length < 2) {
    return text.ToUpper();
  }
  return Char.ToUpper(text[0]) + text.Substring(1).ToLower();
}

public static string CapitalizeEveryWord(string text) {
  var words = text.Split(' ');
  for (int n = 0; n < words.Length; n++) {
    words[n] = CapitalizeFirstLetter(words[n]);
  }
  return String.Join(" ", words);
}

public static string FormatFilename(string filename) {
  var words = filename.Split(' ');
  for (int n = 0; n < words.Length; n++) {
    string word = words[n];
    if (word.Length < 2) {
      words[n] = word.ToUpperInvariant();
    } else {
      words[n] = Char.ToUpperInvariant(word[0]) +
        word.Substring(1).ToLowerInvariant();
    }
  }
  return String.Join("_", words);
}
```

This way, you won't have to cram every possible logic into a single function. This gets especially important when requirements diverge between callers.

3.4.1 Reuse or copy?

How do you decide between reusing the code and replicating it somewhere else? The greatest factor would be how you frame the caller's concerns, that is, describing the caller's requirements for what they actually are. When you describe the requirements of the function where a filename needs to be formatted, you get biased by the existence of a function that is quite close to what you want to do (capitalization), and that immediately signals to your brain to use that existing function. If the filename would be capitalized exactly the same way, it might still make sense, but the difference in requirements should signal a red flag to you.

There are two hard things in computer science: cache invalidation, naming things and off-by-one errors[3]. Naming things correctly is one of the most important factors when understanding conflicting concerns in code reuse. The name `Capitalize` frames the function

[3] This is an excellent variation by Leon Bambrick (https://twitter.com/secretGeek/status/7269997868) on the famous quote by Phil Karlton who said it without the "off-by-one errors" part.

in a correct way. We could have called it NormalizeName when we first created it but it would have prevented us from reusing it in other fields. What we did was to name things as close as possible to its actual functionality. This way, our function can serve all the different purposes without creating confusion and more importantly, it explains its job better wherever its used.

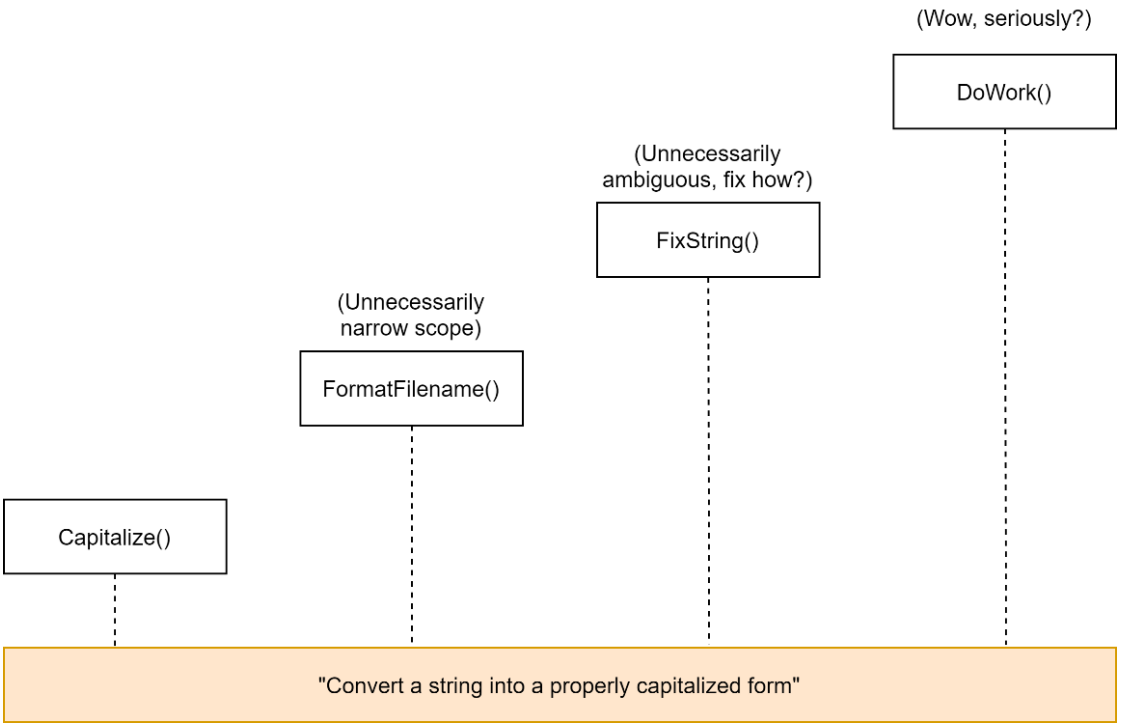


Figure 3.8 Pick a name as close as possible to the actual functionality.

We could go deeper with the actual functionality like "this function converts first letters of each word in a string to upper-case and converts all the remaining letters to lower-case" but that's hard to fit in a name. Names should be the short and as unambiguous as possible. "Capitalize" works in that sense.

Awareness of concerns for a piece of code is an important skill to have. I usually assign personalities to functions and classes to categorize their concerns. I'd say "this function doesn't care about this" as if they were a person. You can similarly get an understanding of the concerns of a piece of code. That's why we named the parameter to capitalize every word `everyWord` instead of `isCity`. Because the function just doesn't care if it's a city or not. It isn't function's concern.

When you name things closer to their circle of concern, their usage patterns become more apparent. Then why did we end up naming filename formatting function

`FormatFilename`? Shouldn't we have called it `CapitalizeInvariantAndSeparateWithUnderscores`? No. Functions can do multiple things but only perform a single task and they should be named after that task. If you feel the need to use the conjunctions "and" or "or" in your function's name, you're either naming it wrong, or you are putting too much responsibility on your function.

Name is the one aspect of the concerns of a code. Where the code resides, its module, its class can also be an indication on how to decide in reusing it.

3.5 Invent it here

There is a common Turkish expression which literally translates to "don't come up with an invention now". It means "don't cause us trouble by trying a novel thing now, we don't have time for that". Reinventing the wheel is known to be problematic. That pathology even has its own name in the computer science circles: *Not Invented Here Syndrome*. It specifically addresses a type of people who cannot sleep at night if they don't invent an already invented product themselves.

It's certainly a lot of work to go to great lengths to create something from scratch where there is a known and working alternative. It's prone to errors too. The problem is when reusing existing stuff becomes the norm and creating something becomes something unreachable. The calcification of this perspective eventually turns into the motto of "never invent anything". You shouldn't let yourself be scared of inventing things.

First, an inventor has a questioning mindset. If you keep on questioning things, you will inevitably become an inventor. When you explicitly prevent yourself from asking questions, you start to become dull, you turn yourself into a menial worker. You should avoid that mindset because it's impossible for someone without a questioning mindset to optimize their work.

Secondly, not all inventions have alternatives. Your own abstractions are also inventions. Your classes, your design, the helper functions you come up with. They are all productivity enhancements, yet they require invention.

I always wanted to write a web site that provides Twitter statistics reports about my followers and people I follow. The thing is I don't want to learn how Twitter API works. I know there are libraries out there that handle this, but I also don't want to learn how they work, or more importantly, I don't want their implementation to influence my design. Because if I use a certain library, it will bind me to the API of that library and if I want to change the library, I will need to rewrite code everywhere.

The way to deal with these involves invention. We come up with our dream interface and put it as an abstraction in front of the library we use. This way, we avoid binding ourselves to a certain API design. If we want to change the library we use, we just change our abstraction, not everything in our code. I currently have no idea how Twitter web API works, but I imagine that it is a regular web request with something to identify the authorization to access the Twitter API. That means, getting an item from Twitter.

A programmer's first reflex is to find a package and check out its documentation on how it works to integrate it into their code. Instead of doing that, invent a new API yourself and use it, which eventually calls the library that you're using behind the scenes. Your API should be the simplest possible for your requirements. Become your own customer.

First, go over the requirements of an API. Web-based APIs provide a user interface on the web to give permissions to an application. It opens up a page on Twitter that asks for permissions and redirects back to the app if the user confirms it. That means we need to know which URL to open for authorization, and which URL to redirect back and use the data in the redirected page to make additional API calls later.

We shouldn't need anything else after we authorize. So, I imagine an API like this for this purpose:

Listing 3.5 Our imaginary Twitter API

```
public class Twitter {
  public static Uri GetAuthorizationUrl(Uri callbackUrl) {    #A
    string redirectUrl = "";
    // … do something here to build the redirect url
    return new Uri(redirectUrl);
  }

  public static TwitterAccessToken GetAccessToken(    #A
    TwitterCallbackInfo callbackData) {
    // we should be getting something like this
    return new TwitterAccessToken();
  }

  public Twitter(TwitterAccessToken accessToken) {
    // we should store this somewhere
  }

  public IEnumerable<TwitterUserId> GetListOfFollowers(    #C
    TwitterUserId userId) {
    // no idea how this will work
  }
}

public class TwitterUserId {    #B
  // who knows how twitter defines user ids
}

public class TwitterAccessToken {    #B
  // no idea what this will be
}

public class TwitterCallbackInfo {    #B
  // this neither
}
```

#A Static functions that handle the authorization flow
#B Classes to define Twitter's concepts
#C The actual functionality we want

We invented something from scratch, a new Twitter API, even though we know little about how Twitter API works in reality. It might not be the best API for general use, but our customer is ourselves, so we have the luxury to design it to fit our needs. For instance, I don't think I'll need to handle how the data is transferred in chunks from the original API and I don't care if it makes me wait and blocks the running code which may not be desirable in a more generic API.

MINI-HINT This approach to having your own convenient interfaces that act as an adapter is unsurprisingly called *adapter pattern* in the streets. I avoid emphasizing names over actual utility but in case somebody asks you, now you know it.

We can later extract an interface from the classes we defined, so we don't have to depend on concrete implementations. It makes testing easier that way. We don't even know if the Twitter library we're going to use support replacing their implementation easily. You may occasionally encounter cases where your dream design doesn't really fit with the design of the actual product. In that case you need to tweak your design too but that's a good sign. That means your design also represents your understanding of the underlying technology.

So, I might have lied a little. Don't write a Twitter library from scratch. But don't stray from the inventor's mindset either. Those go hand in hand, and you should stick to both.

3.6 Don't use inheritance

Object-Oriented Programming (OOP) fell into the programming world like an anvil in the 90's and it caused a paradigm shift from structured programming. It was considered revolutionary. The decades old problem of how to reuse code had finally been resolved.

The most emphasized feature of OOP was inheritance. You could define code reuse as a set of inherited dependencies. Not only did this allow simpler code reuse, it also allowed simpler code modification too. In order to have a new code that has a slightly different behavior, you didn't need to think about changing the original code. You just derived from it and overrode the relevant member in order to have modified behavior.

Inheritance caused more problems than it solved in the long run. *Multiple inheritance* was one of the first issues. What if you had to reuse the code from multiple classes and both had the method with the same name, and with the same signature perhaps. How would it work? What about the diamond dependency problem as shown in Figure 3.2? It would be really complicated, therefore very few programming languages went ahead and implemented it.

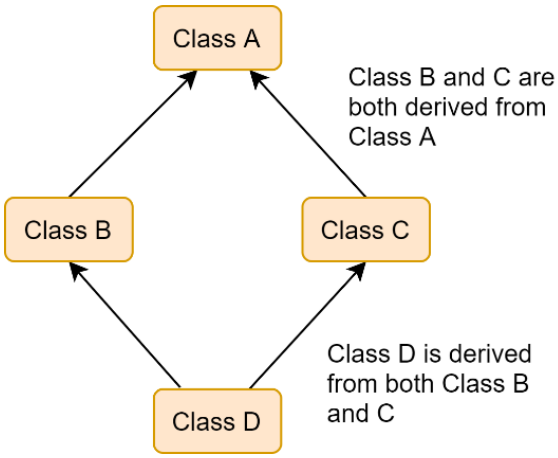


Figure 3.9 Diamond dependency problem, how should class D behave?

A greater problem with inheritance other than multiple inheritance is the problem of strong dependency, also known as tight coupling. As we already discussed, dependencies are the root of all evil. Because of its nature, inheritance binds you to a concrete implementation, which is considered as violating one of the well-regarded principles of object-oriented programming called dependency inversion principle which states that code should never depend on the concrete implementation, but an abstraction instead.

On SOLID principles

There is a famous acronym SOLID that stands for five principles of Object Oriented Programming. The problem is that SOLID is an acronym that feels like it was invented to make a meaningful word rather than making us better programmers. I don't think all of the principles carry the same importance, and some may not matter at all. I strongly oppose embracing a set of principles without getting convinced of their value.

Single-responsibility principle, the S of SOLID, says a class should be responsible for one thing only as opposed to having one class do multiple things, aka God classes. That's a bit vague as it's us who defines what "one thing" entails. Can we say a class with two methods still responsible for one thing anymore? Even a God class is responsible for one thing at a certain level: being the God class. I'd replace this with "clear name principle": the name of a class should explain its function with as little vagueness as possible. If the name is too long or too vague, the class needs to be split into multiple classes.

Open-closed principle states that a class should be open for extension but closed for modification. It means that we should design our classes in a way that its behavior can be modified externally. This is, again, very vague, and can even be unnecessarily time consuming. Extensibility is a design decision and may not be desirable, practical or even safe at times. It feels like the advice "use racing tires" of programming. I would instead say, "treat extensibility as a feature."

Liskov substition principle, coined by Barbara Liskov, states that a program's behavior shouldn't change if one of the classes used is replaced with a derived class. Although the advice is sound, I don't think it matters in daily programming work. It feels like an advice like "don't have bugs" to me. If you break an interface's contract, the

program will have bugs. If you design a bad interface, the program will also have bugs. That's the natural order of things. Perhaps, this can be turned into a simpler and more actionable advice like "stick to the contract."

Interface segregation principle favors smaller and goal-specific interfaces over generalized, broadly scoped interfaces. This is an unnecessarily complicated and vague advice if not wrong. There could be cases where broadly scoped interfaces are more suitable for the job, and overly granular interfaces can create too much overhead. Splitting interfaces shouldn't be based on scope, but actual requirements of the design. If a single interface isn't suitable for the job, feel free to split it, not to satisfy some granularity criteria.

Dependency inversion principle is the final one, again, not a very good name. Just call it "depend on abstractions." Yes, depending on concrete implementations create tight coupling and we've already seen its undesirable effects. But, that doesn't mean you should start creating interfaces for every dependency you have. I say the opposite, prefer depending abstractions when you prefer flexibility, you see value in it, and depend on the concrete implementation in cases where it just doesn't matter. Your code should adapt to your design, not the other way around. Feel free to experiment with different models.

Why is there such a principle? Because when you are bound to a concrete implementation your code becomes rigid and immovable. Rigid code is very hard to test or modify as we discussed previously.

How do you reuse code then? How do you inherit your class from an abstraction? It's simple, and it's called *composition*. Instead of inheriting from a class, you receive its abstraction as a parameter in your constructor. Think of your components as lego pieces that support each other, rather than a hierarchy of objects.

With regular inheritance, the relationship between common code and its variations is expressed with an ancestor/descendant model. Composition thinks of the common code as a separate component instead.

Composition is more like a client-server relationship than a parent-child one. You call reused code by its reference instead of having their methods inherited in your scope, and occasionally differentiated with the "`super.`" prefix. You can construct the class you're depending on in your constructor, or even better, you can receive it as a parameter, which would let you use it as an external dependency. That allows you to make that relationship more configurable, and flexible.

Receiving it as a parameter has the extra advantage of making it easier to unit test the object by injecting mock versions of the concrete implementations. We'll discuss dependency injection more in chapter five.

Using composition over inheritance can require writing substantially more code, as you might need to define dependencies with interfaces instead of concrete references, but it would free the code from dependencies. You still need to weigh pros and cons of composition before you use it.

3.7 Don't use classes

Make no mistake, classes are great. They do their job, and get out of the way. But as we discussed in chapter two, they incur a small reference indirection overhead and occupy slightly more indirection compared to value types. These won't be an issue most of the time,

but it's important for you to know pros and cons of them for understanding the code and how you can impact it by making the wrong decisions.

Value types can be, well, valuable. The primitive types that comes with C# such as int, long, and double are value types already. You can also compose your own value types with constructs like enum and struct.

3.7.1 Enum is yum!

Enums are great for holding discrete ordinal values. Classes can also be used to define discrete values, but they lack certain affordances that enums have. A class is still, of course, better than hardcoding values.

If you're writing code that handles the response of a web request that you make in your app, you may need to deal with different numerical response codes. Say that you're querying weather information from National Weather Service for user's given location, and you write a function to retrieve the required information. In Listing 3.6, we're using RestSharp for API requests and Newtonsoft.JSON to parse the response if the request is successful by checking the HTTP status code is successful or not. Notice that we're using a hard coded value (200) on the "`if`" line to check for the status code. We then use Json.NET library to parse the response into a dynamic object to extract the information we need.

Listing 3.6 Function that returns NWS temperature forecast for a given coordinate

```
static double? getTemperature(double latitude,
  double longitude) {
  const string apiUrl = "https://api.weather.gov";
  string coordinates = $"{latitude},{longitude}";
  string requestPath = $"/points/{coordinates}/forecast/hourly";
  var client = new RestClient(apiUrl);
  var request = new RestRequest(requestPath);
  var response = client.Get(request);    #A
  if (response.StatusCode == 200) {    #B
    dynamic obj = JObject.Parse(response.Content);    #C
    var period = obj.properties.periods[0];
    return (double)period.temperature;    #D
  }
  return null;
}
```

#A Send the request to NWS.
#B Check for successful HTTP status code.
#C We parse JSON here.
#D Yay, result!

The greatest problem with hard coded values is the inability of humans to memorize numbers. We're not good at it. We don't understand them at first sight with the exception of number of zeros on our paychecks. They are harder to type than simple names as it's hard to associate numbers with mnemonics yet they are easier to make a typo with. The second problem with hard coded values is that values can change. If you use the same value everywhere else, that means changing everything else too, just to change a value.

The second problem with numbers is that they lack intent. A numeric value like 200 can be anything. We don't know what it is. So, don't hard code values.

Classes are one way to encapsulate values. You can encapsulate HTTP status codes in a class like this:

```
class HttpStatusCode {
  public const int OK = 200;
  public const int NotFound = 404;
  public const int ServerError = 500;
  // ... and so on
}
```

This way you can change the line that checks for a successful HTTP request with a code like this:

```
if (response.StatusCode == HttpStatusCode.OK) {
…
}
```

That version looks way more descriptive. We immediately understand the context, what the value means and what it means in which context. It's perfectly descriptive.

Then, what are enums for? Can't we use classes for this? Consider that we have another class for holding values:

```
class ImageWidths {
  public const int Small = 50;
  public const int Medium = 100;
  public const int Large = 200;
}
```

Now this code would compile, and more importantly it would return true:

```
return HttpStatusCode.OK == ImageWidths.Large;
```

That's something you probably don't want. If we had written it with an enum instead:

```
enum HttpStatusCode {
  OK = 200,
  NotFound = 404,
  ServerError = 500,
}
```

Way easier to write, right? Its usage would be the same in our example. More importantly, every enum type you define is distinct, which makes the values type-safe unlike our example with classes with consts. Enum is a blessing in our case. If we tried the same comparison with two different enum types, compiler would throw an error:

```
error CS0019: Operator '==' cannot be applied to operands of type 'HttpStatusCode' and
       'ImageWidths'
```

Awesome! Enums save us time by not allowing us to compare apples to oranges during compilation. They convey intent as good as classes that contain values. Enums are also value types, which means they are as fast as passing around an integer value.

3.7.2 Struct rocks!

As you saw in chapter two, classes have a little storage overhead. Every class needs to keep some object header and virtual method table when instantiated. Additionally, classes are allocated on the heap and they are garbage collected.

That means .NET needs to keep track of every class instantiated and get them out of memory when not needed. That's a very efficient process, most of the time you don't even notice it's there. It's magical. No manual memory management. So, no, you don't have to be scared of using classes.

But, as we've seen before, it's good to know when you can take advantage of a free benefit when it's available. Structs are like classes. You can define properties, fields, and methods in them. Structs can also implement interfaces. However, a struct cannot be inherited, or cannot inherit from another struct or class. That's because structs don't have a virtual method table, or the object header. They are not garbage collected because they are allocated on the stack, the call stack.

As we discussed in chapter two, call stack is just a contiguous block of memory with only its head pointer moving around. That makes stack a very efficient storage mechanism because cleanup is fast and automatic. There is no possibility of fragmentation because it's always LIFO (Last In First Out).

If stack is that fast, why don't we use it for everything? Why is there heap, or garbage collection? That's because stack can only live for the lifetime of the function. When your function returns, anything on the function's stack frame is gone, so other functions can use the same stack space. We need heap for the objects that outlive functions.

Also, stack is limited in size. That's why there is a whole web site named Stack Overflow because your application will crash if you overflow the stack. Respect the stack, know its limits.

Structs are lightweight classes. They are allocated on stack because they are value types. That means assigning a struct value to a variable means copying its contents since there is no single reference that represents it. You need to keep this in mind because copying is slower than passing around references for any data larger than a size of a pointer.

Although structs are value types themselves, they can still contain reference types. Say, if a struct contains a `string`, it's still a reference type inside a value type, similar to how you can have value types inside a reference type. You will see this depicted in the figures throughout this section.

If you have a struct that contains only an integer value, it occupies less space in general than a reference to a class that contains an integer value as you can see in Figure 3.3. Consider that our struct and class variants are about holding identifiers as we discussed in chapter two. Two flavors of the same structure would look like as in Listing 3.7:

Listing 3.7 Similarity of class and struct declarations

```
public class Id {
  public int Value { get; private set; }

  public Id (int value) {
    this.Value = value;
  }
```

```
}

public struct Id {
  public int Value { get; private set; }

  public Id (int value) {
    this.Value = value;
  }
}
```

The only difference in code is struct vs class keywords but see how they differ in how they are stored when you create them in a function like this:

```
var a = new Id(123);
```

Figure 3.9 shows you how they are laid out.

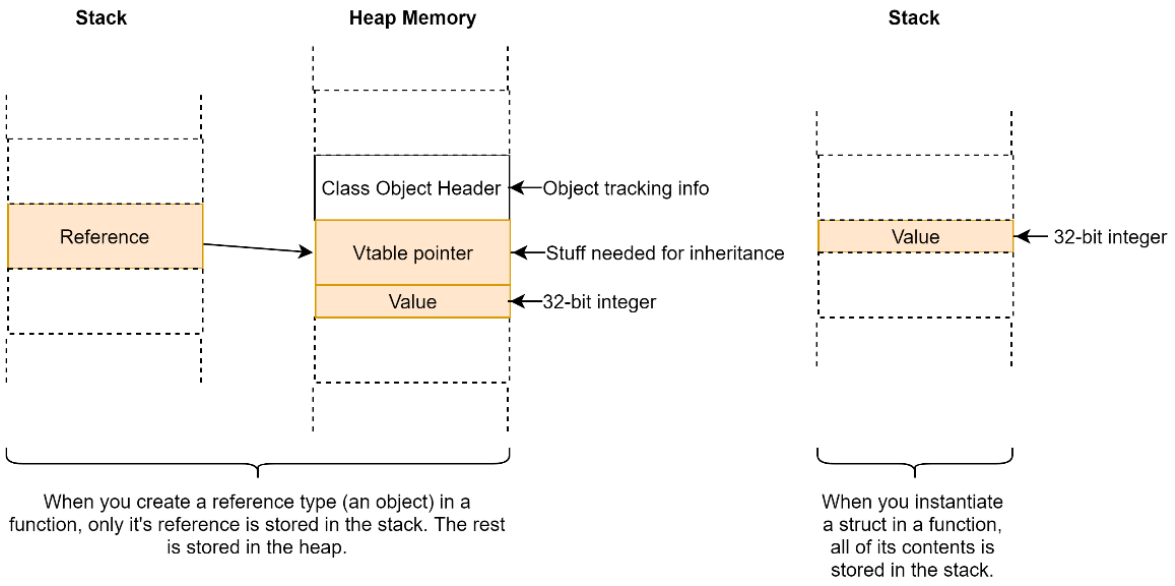


Figure 3.10 The difference between how classes and structs are laid out in memory

Because structs are value types, assigning one to another also creates another copy of the whole content of the struct instead of just creating another copy of the reference.

```
var a = new Id(123);
var b = a;
```

In this case, you can see how structs can be efficient for storage of small types in Figure 3.10.

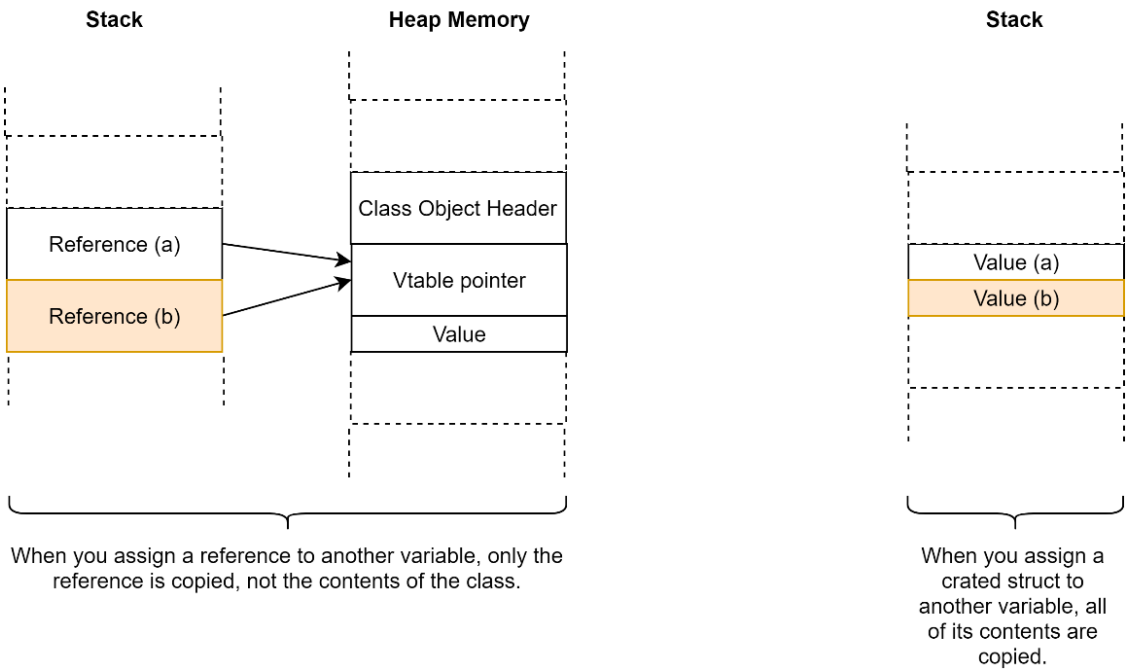


Figure 3.11 Efficiency of small structs in memory storage

Although stack storage is temporary during the execution of the function, it's miniscule compared to the heap. Stack is one megabyte in size in .NET, while heap can contain terabytes of data. Stack is fast, but if you fill it with large structs, it can get filled easily. Besides, copying large structs is also slower than only copying a reference. Consider that we'd like to keep some user information along with our identifiers. Our implementation would look like Listing 3.8.

Listing 3.8 Defining a larger class or a struct

```
public class Person {    #A
  public int Id { get; private set; }
  public string FirstName { get; private set; }
  public string LastName { get; private set; }
  public string City { get; private set; }

  public Person(int id, string firstName, string lastName,
    string city) {
    Id = id;
    FirstName = firstName;
    LastName = lastName;
    City = city;
  }
}
```

#A We can make a class a struct by changing the "class" word here to "struct".

The only difference between two definitions are struct and class keywords. Yet creating and assigning one from another has profound impact on how things work behind the scenes. Consider this simple code where it can be either a struct or a class:

```
var a = new Person(42, "Sedat", "Kapanoglu", "San Francisco");
var b = a;
```

After you assign a to b, the difference in resulting memory layouts is shown in figure 3.11.

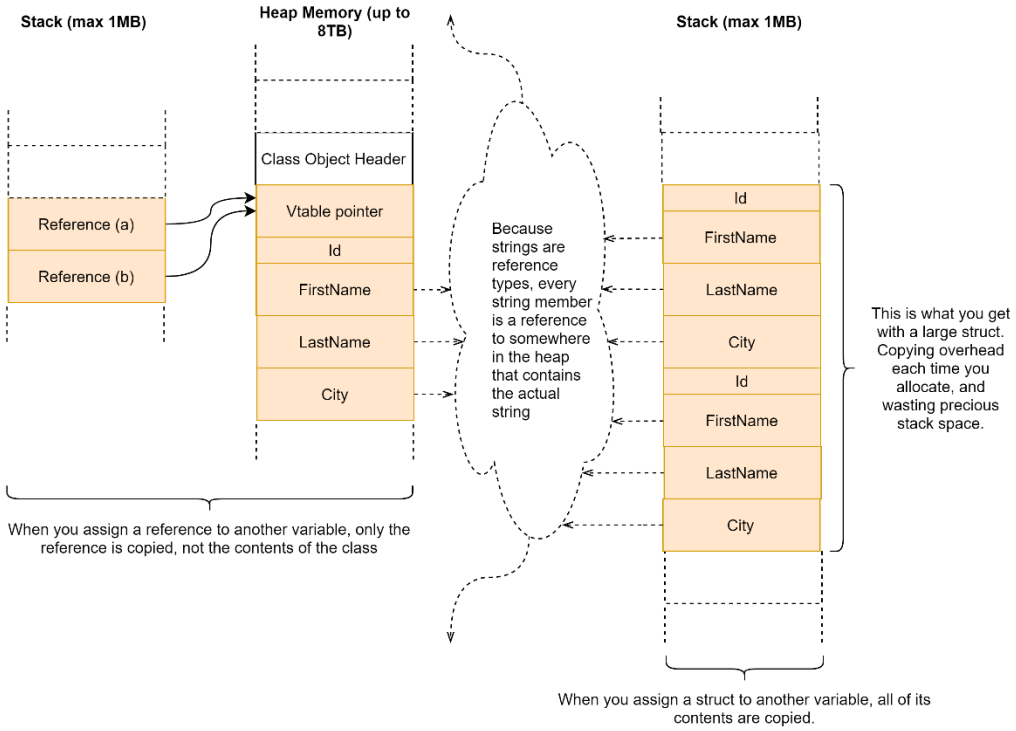


Figure 3.12 Impact difference between value types and reference types in larger types

Call stack can be extremely fast and efficient to store things. They are great for working with small values with less overhead because they are not subject to Garbage Collection. Because they are not reference types, they cannot be null either, which makes null reference exceptions impossible to happen with structs.

You can't use structs for everything as is apparent from how they are stored: you can't share a common reference to them, which means you can't change a common instance from different references. That's something we do a lot unconsciously and never think of about. Consider if we wanted the struct to be mutable, and used "get; set;" modifiers instead of "get; private set;". That meant we could modify the struct on the fly.

Listing 3.9 A mutable struct

```
public struct Person {
  public int Id { get; set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public string City { get; set; }

  public Person(int id, string firstName, string lastName,
    string city) {
    Id = id;
    FirstName = firstName;
    LastName = lastName;
    City = city;
  }
}
```

Consider this piece of code with a mutable struct:

```
var a = new Person(42, "Sedat", "Kapanoglu", "San Francisco");
var b = a;
b.City = "Eskisehir";
Console.WriteLine(a.City);
Console.WriteLine(b.City);
```

What do you think the output would be? If it were a class, both lines would show "Eskisehir" as the new city. But since we have two separate copies, it would print "San Francisco" and "Eskisehir". Because of this, it's always a great idea to make structs almost always immutable so they can't be accidentally changed later and cause bugs.

Although you should prefer composition over inheritance for code reuse, inheritance can also be useful at times when the given dependency is contained. Classes can provide you better flexibility than structs in those cases.

Classes can provide more efficient storage when they are larger in size because only their references would be copied in an assignment. In light of what we have discussed, feel free to use structs for small, immutable value types which have no need for inheritance.

3.8 Write bad code

Best practices come from bad code, yet bad code can also emerge from the blind application of best practices. Structured, object-oriented, and even functional programming are all developed to make developers write better code. When best practices are taught, some bad practices are also singled out as "evil" and they were completely banished. Let's visit some of them.

3.8.1 Don't use if/else

If/Else is one of the first constructs you learn about programming. They are the expression of one of the fundamental parts of computers: logic. We love If/Else. It lets us to express the logic of our program in a flowchart-like way. But that kind of expression can make code less readable too.

Like many programming constructs, If/Else blocks make the code in the conditionals to be indented. Suppose that we want to add some functionality to our Person class from the last section to process a record in the DB. We want to see if the City property of the Person class was changed or not and change it in the DB too if the Person class points to a valid record. This is quite a stretched implementation, there are better ways to do these but I want to show you how the code can turn out, rather than its actual functionality. I'd like to draw a shape to you in Listing 3.10.

Listing 3.10 An example of If/Else festival in the code

```
public UpdateResult UpdateCityIfChanged() {
  if (Id > 0) {
    bool isActive = db.IsPersonActive(Id);
    if (isActive) {
      if (FirstName != null && LastName != null) {
        string normalizedFirstName = FirstName.ToUpper();
        string normalizedLastName = LastName.ToUpper();
        string currentCity = db.GetCurrentCityByName(
          normalizedFirstName, normalizedLastName);
        if (currentCity != City) {
          bool success = db.UpdateCurrentCity(Id, City);
          if (success) {
            return UpdateResult.Success;
          } else {
            return UpdateResult.UpdateFailed;
          }
        } else {
          return UpdateResult.CityDidNotChange;
        }
      } else {
        return UpdateResult.InvalidName;
      }
    } else {
      return UpdateResult.PersonInactive;
    }
  } else {
    return UpdateResult.InvalidId;
  }
}
```

Even if I explained what the function did step by step, it's impossible to come back to this function five minutes later and not to be confused again. One of the reasons for the confusion is too much indentation. People are not accustomed to read things in indented format, with the small exception of Reddit users. It's hard to find which block a line belongs to, what's the context. It's hard to follow the logic.

The general principle to avoid unnecessary indentation is exiting the function as early as possible and avoiding using "else" when the flow already implies an else. In Listing 3.11, you can see how return statements already imply end of the code flow, eliminating the need for else.

Listing 3.11 Look ma, no elses!

```
public UpdateResult UpdateCityIfChanged() {
  if (Id <= 0) {
```

```
    return UpdateResult.InvalidId;    #A
  }
  bool isActive = db.IsPersonActive(Id);
  if (!isActive) {
    return UpdateResult.PersonInactive;    #A
  }
  if (FirstName is null || LastName is null) {
    return UpdateResult.InvalidName;    #A
  }
  string normalizedFirstName = FirstName.ToUpper();
  string normalizedLastName = LastName.ToUpper();
  string currentCity = db.GetCurrentCityByName(
    normalizedFirstName, normalizedLastName);
  if (currentCity == City) {
    return UpdateResult.CityDidNotChange;    #A
  }
  bool success = db.UpdateCurrentCity(Id, City);
  if (!success) {
    return UpdateResult.UpdateFailed;    #A
  }
  return UpdateResult.Success;    #A
}
```

#A No code runs after a return.

The technique used there is called following on *the happy path*. The happy path in code is the part of the code which runs if nothing else goes wrong. It's what ideally happens during execution. Since happy path summarizes function's main work, it must be the easiest part to read. By converting the code in `else` statements into early return statements, we allow the reader to identify happy path much easier than having matryoshka dolls of `if` statements.

Validate early, and return as early as possible. Put the exceptional cases inside `if` statements and try to put your happy path outside of the blocks. Try to familiarize yourself with these two shapes to make your code more readable and maintainable.

3.8.2 Use "go to"

Entire theory of programming can be summarized with memory, basic arithmetic, `if` and `goto` statements. A "go to" statement transfers the execution of the program to an arbitrary destination point directly. They are hard to follow and their use are discouraged since Edsger Dijkstra wrote the paper titled "Go to statement is considered harmful". There are many misconceptions about both Dijkstra's paper, first and foremost its title. Dijkstra gave his paper the title "A case against the GO TO statement", but his editor, also the inventor of the Pascal language, Niklaus Wirth changed the title of the paper which made Dijkstra's stance more aggressive and turned the war against `goto` into a witch hunt.

This all happened before 1980's though. Programming languages had ample time to create new constructs to address the functions of the `goto` statement. The `for`/`while` loops, `return`/`break`/`continue` statements, even exceptions were created to address specific scenarios that was previously only possible with `goto`. Former BASIC programmers would remember the famous error handling statement `ON ERROR GOTO` which was a primitive exception handling mechanism.

Although many modern languages don't have a goto equivalent anymore, C# does, and it works great for one single scenario: eliminating redundant exit points in a function. It's possible to use goto statement in an easy to understand fashion and make your code less prone to bugs while saving you time. It's like a three combo hit on Mortal Kombat.

An exit point is each statement in a function which causes it to return to its caller. Every `return` statement is an exit point in C#. Eliminating exit points in the olden era of programming languages was more important than it is today because manual cleanup was a more prominent part of a programmer's daily life. You had to remember that what you allocated and what you needed to clean up before you return.

C# provides great tools for structured cleanup such as `try`/`finally` blocks and `using` statements. There may be cases where neither works for your scenario and you can use `goto` for cleanup too, but it actually shines more in eliminating redundancy. Let's say we're developing the shipment address entry form for an online shopping web page. Web forms are great for demonstrating the multi-level validation happening with them. Assume that we'd like to use ASP.NET Core for that. That means we need to have a "submit" action for our form. Its code might look like Listing 3.12. We have model validation that happens in the client, but at the same time we need some server validation with our form too, so we can check if the address is really correct using USPS API. After the check, we can later try to save the information to the database and if it succeeds we redirect the user to the billing information page. Otherwise we need to display the shipping address form again.

Listing 3.12 A shipping address form handling code with ASP.NET Core

```
[HttpPost]
public IActionResult Submit(ShipmentAddress form) {
  if (!ModelState.IsValid) {
    return RedirectToAction("Index", "ShippingForm", form);    #A
  }
  var validationResult = service.ValidateShippingForm(form);
  if (validationResult != ShippingFormValidationResult.Valid) {
    return RedirectToAction("Index", "ShippingForm", form);    #A
  }
  bool success = service.SaveShippingInfo(form);
  if (!success) {
    ModelState.AddModelError("", "Problem occurred while " +
      "saving your information, please try again");
    return RedirectToAction("Index", "ShipingForm", form);    #A
  }
  return RedirectToAction("Index", "BillingForm");    #B
}
```

#A Redundant exit points
#B The happy path

We already discussed some of the issues with copy/paste, but multiple exit points in Listing 3.12 pose another problem. Did you notice the typo in the third return statement? We accidentally deleted a character without noticing, and since it's in a string, that bug is impossible to detect unless we encounter a problem when saving the form in the production or we build elaborate tests for our controllers. Duplication can cause problems in these cases.

The `goto` statement can help you to merge the return statements under a single `goto` label like in Listing 3.13. We create a new label for our error case under our happy path and

Listing 3.13 Merging common exit points into a single return statement

```
[HttpPost]
public IActionResult Submit2(ShipmentAddress form) {
  if (!ModelState.IsValid) {
    goto Error;    #A1
  }
  var validationResult = service.ValidateShippingForm(form);
  if (validationResult != ShippingFormValidationResult.Valid) {
    goto Error;    #A2
  }
  bool success = service.SaveShippingInfo(form);
  if (!success) {
    ModelState.AddModelError("", "Problem occurred while " +
      "saving your shipment information, please try again");
    goto Error;    #A3
  }
  return RedirectToAction("Index", "BillingForm");
Error:    #B
  return RedirectToAction("Index", "ShippingForm", form);    #C
}
```

#A The infamous goto!
#B Destination label
#C Common exit code

The great thing about this kind of consolidation is if you ever want to add more in your common exit code, you only need to add it to a single place. Let's say you want to save a cookie to the client when there was an error. All you need to do is to add it after the "Error" label as in Listing 3.14.

Listing 3.14 Ease of adding extra code to common exit code

```
[HttpPost]
public IActionResult Submit3(ShipmentAddress form) {
  if (!ModelState.IsValid) {
    goto Error;
  }
  var validationResult = service.ValidateShippingForm(form);
  if (validationResult != ShippingFormValidationResult.Valid) {
    goto Error;
  }
  bool success = service.SaveShippingInfo(form);
  if (!success) {
    ModelState.AddModelError("", "Problem occurred while " +
      "saving your information, please try again");
    goto Error;
  }
  return RedirectToAction("Index", "BillingForm");
Error:
  Response.Cookies.Append("shipping_error", "1");    #A
  return RedirectToAction("Index", "ShippingForm", form);
}
```

#A The code that saves the cookie

By using `goto`, we actually kept our code style more readable with less indents, saved us time, and made the code easier to maintain for the future changes.

A statement like goto can still perplex a colleague who is not used to the syntax. Luckily, C# 7.0 introduced local functions which can be used to perform the same work, perhaps in an easier to understand way. We declare a local function called `error` that performs the common error return operation and return its result instead of using `goto`.

Listing 3.15 Using local functions instead of goto

```
[HttpPost]
public IActionResult Submit4(ShipmentAddress form) {
  IActionResult error() {    #A
    Response.Cookies.Append("shipping_error", "1");
    return RedirectToAction("Index", "ShippingForm", form);
  }
  if (!ModelState.IsValid) {
    return error();    #B
  }
  var validationResult = service.ValidateShippingForm(form);
  if (validationResult != ShippingFormValidationResult.Valid) {
    return error();    #B
  }
  bool success = service.SaveShippingInfo(form);
  if (!success) {
    ModelState.AddModelError("", "Problem occurred while " +
      "saving your information, please try again");
    return error();    #B
  }
  return RedirectToAction("Index", "BillingForm");
}
```

#A Our local function
#B Common error return cases

Using local functions also allow us to declare error handling at the top of the function, which is the norm with modern programming languages like Go with statements like `defer`, although in our case, we have to explicitly call the `error()` function to execute it.

3.9 Don't write code comments

There is a famous Turkish architect called Sinan who lived in 16th century. He built the famous Suleymaniye Mosque in Istanbul and countless other buildings during his time. There is a story about his prowess of architecture. As the story goes, hundreds of years after Sinan passed, a group of architects started restoration work on one of his buildings. There was a keystone in one of the archways, which they needed to replace. They carefully removed the stone block and found a small glass vial wedged between blocks that contained a note. The note said: "This keystone would last only three hundred years. If you're reading this note, it must have broken down or you are trying to repair it. There is only one right way to put a new key stone back in correctly", and the note continued with the technical details on how to replace the keystone the proper way.

Sinan the Architect could be the first person in the history who used code comments correctly. Consider the opposite case where the building had writings everywhere on it. Doors would have the text "this is a door". Windows would have "window" written over them. Between every brick there would be a glass vial with a note in them saying "these are bricks".

You don't need to write code comments if your code is sufficiently self-explanatory. Conversely, you can hurt the readability of code with extraneous comments. Don't write code comments just for the sake of writing comments. Use them wisely and only when necessary.

Consider the example in the previous section, Listing 3.15. If we had gone overboard with code comments it could have looked like this:

Listing 3.16 Code comments everywhere!

```
/// <summary>
/// Receive a shipment address model and update it in the
/// database and then redirect the user to billing page if
/// it's successful.    #A
/// </summary>
/// <param name="form">The model to receive.</param>    #A
/// <returns>Redirect result to the entry form if
/// there is an error, or redirect result to the
/// billing form page if successful.</returns>
[HttpPost]    #A
public IActionResult Submit(ShipmentAddress form) {    #A
  // Our common error handling code that saves the cookie
  // and redirects back to the entry form for
  // shipping information.    #B
  IActionResult error() {    #B
    Response.Cookies.Append("shipping_error", "1");    #B
    return RedirectToAction("Index", "ShippingForm", form);    #B
  }
  // check if the model state is valid    #C
  if (!ModelState.IsValid) {    #C
    return error();
  }
  // validate the form with server side validation logic.    #D
  var validationResult = service.ValidateShippingForm(form);    #D
  // is the validation successful?    #E
  if (validationResult != ShippingFormValidationResult.Valid) {    #E
    return error();
  }
  // save shipping information    #F
  bool success = service.SaveShippingInfo(form);    #F
  if (!success) {
    // failed to save. report the error to the user.     #G
    ModelState.AddModelError("", "Problem occurred while " +    #G
      "saving your information, please try again");
    return error();
  }
  // go to the billing form    #H
  return RedirectToAction("Index", "BillingForm");    #H
}
```

#A These are already explained by the function's context and declaration.
#B Literally a repetition of the following code.

#C Again, completely unnecessary.
#D Another repetition.
#E Come on!
#F Really? We've come to this now?
#G No kidding, Sherlock.
#H I would've never guessed.

The code we're reading tells us a story even without the comments. Let's go over the same code without comments and find the hidden hints in it.

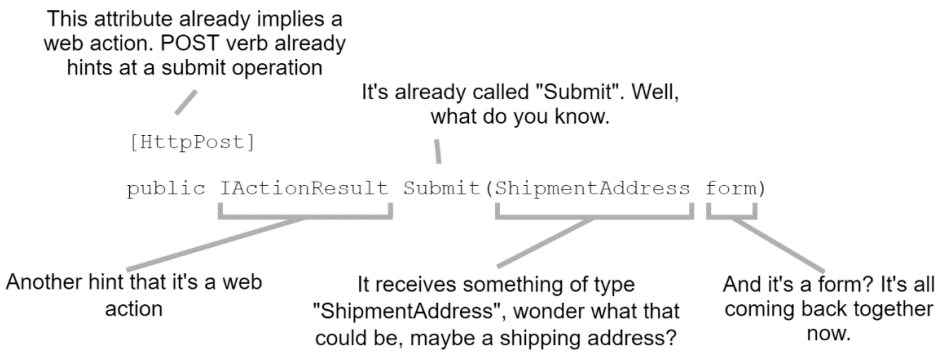


Figure 3.13 Reading hints in a code

This might look like a lot of work. Trying to bring the pieces of together just to understand what it does. It gets better over time though. You spend less effort the more you get better at it. There are things that you can do to improve the lives of the poor soul who reads your code, and even yourself six months later, because after six months, it might as well be somebody else's code.

3.9.1 Choose great names

We already touched to the importance of good names at the beginning of this chapter, how our names should represent or summarize the functionality as close as possible. Functions shouldn't have ambiguous names like "Process", "DoWork", "Make" and so forth unless the context is absolutely clear. That might sometimes need you to type longer names than usual but it's usually possible to create good names and still keep it concise.

The same applies for variable names. Reserve single letter variable names only for loop variables (`i`, `j`, `n`) and coordinates like `x`, `y`, `z` where they are obvious. Otherwise, always pick a descriptive name, and avoid abbreviations. It's still okay to use well known initialisms like HTTP, JSON or well-known abbreviations like "ID" and "DB", but don't shorten words. You only type the variable name once anyway. Code completion can take care of the rest later. The benefits of descriptive names are tremendous. Most importantly, they save you time. When you pick a descriptive name, you don't have to write a full sentence comment to explain it wherever it's used. Try to consult to the convention documentation of the programming language you're using. Microsoft's guideline for .NET naming conventions, for

example, is a great starting point for C#: https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/naming-guidelines

3.9.2 Leverage functions

Small functions are easier to understand. Try to keep a function small enough to fit in a developer's screen. Scrolling back and forth is a terrible way to understand what a code does. You should be able to see everything the function does in front of you.

How do you shorten a function? Beginners might be inclined to put as much as on a single line to make function more compressed. NO! Never put multiple statements on a single line. Always have at least one line per statement. You can even have blank lines in a function to group relevant statements together. Let's look at our function in light of this in listing 3.17.

Listing 3.17 Using blank lines to separate logical parts of a function

```
[HttpPost]
public IActionResult Submit(ShipmentAddress form) {
  IActionResult error() {
    Response.Cookies.Append("shipping_error", "1");
    return RedirectToAction("Index", "ShippingForm", form);
  }
    #A
  if (!ModelState.IsValid) {
    return error();
  }
    #B
  var validationResult = service.ValidateShippingForm(form);
  if (validationResult != ShippingFormValidationResult.Valid) {
    return error();
  }
    #C
  bool success = service.SaveShippingInfo(form);
  if (!success) {
    ModelState.AddModelError("", "Problem occurred while " +
      "saving your information, please try again");
    return error();
  }
    #D
  return RedirectToAction("Index", "BillingForm");
}
```

#A Error handling code part
#B MVC Model validation part
#C Server-side model validation part
#D Saving part and the successful case

You may ask that how this helps in making the function smaller. Yes, in fact, it makes the function bigger. But identifying logical parts of a function lets you to refactor those parts into meaningful functions too, which is the key to having small functions and descriptive code at the same time. You can refactor the same code into an even more digestible chunks if the logic isn't straightforward to understand. In listing 3.18, we extract parts of the logic in our Submit function by using what we identified as logical parts. We basically have a validation

part, an actual saving part, a save error handling part and a successful response. We only leave those four parts in the body of the function.

Listing 3.18 Keeping only the descriptive functionality in the function

```
[HttpPost]
public IActionResult Submit(ShipmentAddress form) {
  if (!validate(form)) {    #A
    return shippingFormError();
  }
  bool success = service.SaveShippingInfo(form);    #B
  if (!success) {    #C
    reportSaveError();
    return shippingFormError();
  }
  return RedirectToAction("Index", "BillingForm");    #D
}

private bool validate(ShipmentAddress form) {
  if (!ModelState.IsValid) {
    return false;
  }
  var validationResult = service.ValidateShippingForm(form);
  return validationResult == ShippingFormValidationResult.Valid;
}

private IActionResult shippingFormError() {
  Response.Cookies.Append("shipping_error", "1");
  return RedirectToAction("Index", "ShippingForm", form);
}

private void reportSaveError() {
  ModelState.AddModelError("", "Problem occurred while " +
    "saving your information, please try again");
}
```

#A Validation
#B Saving
#C Error handling
#D Successful response

The actual function is so simple that it almost reads like an English sentence; well, maybe a hybrid of English and Turkish, but very readable. We achieved a greatly descriptive code without writing a single line of comment, and that's the key you need keep on your mind if you ever ask if it's too much work. It's less work than writing paragraphs of comments. You'll also thank yourself by shaking your left hand with your right hand when you find out that you also don't need to keep comments and the actual code in sync, for the comments to stay useful over the lifetime of the project. This is way better.

Extracting functions may look like a chore but it's in fact a breeze with development environments like Visual Studio. You just select the part of the code that you want to extract and press `Ctrl+.` (period key) or choose the light bulb icon appearing next to the code and select "Extract method". All you need to do is to give it a name.

When you extract those pieces, you also open a door to reusing those pieces of code in the same file which can save you time when you're writing billing form, if the error handling semantics aren't any different.

This all may sound like I'm against code comments. It's the exactly opposite. Avoiding unnecessary comments makes the useful comments shine like jewels. It's the only way to make comments useful. Think like Sinan when writing comments: "will someone need an explanation for this?". If it needs an explanation, be as clear as possible, be elaborate, even draw ASCII diagrams if needed. Write as many paragraphs as needed, just so the developers working on the same code don't have to come to your desk and ask you what that piece of code does, or fix it incorrectly because you forgot to explain yourself so it comes down to you to fix the code correctly when the production goes down. You owe this to yourself as much as you owe this to everybody else.

There are cases where you must write comments whether they are useful or not, such as public APIs because users may not have access to the code. But that also doesn't mean having written comments makes your code easy to understand. You still need to write clean code with small, easy to digest pieces.

3.10 Summary

- Avoid creating rigid code by avoiding violating logical dependency boundaries.
- Don't be afraid of doing a work from scratch as the next time you do it, it'll be much faster.
- Break the code when there are dependencies that might tie your shoelaces together in the future, and fix it.
- Avoid digging yourself a legacy hole by keeping the code up to date and fixing the problems it causes regularly.
- Repeat the code instead of reusing it to avoid violating logical responsibilities.
- Invent smart abstractions so the future code you write takes less time. Use abstractions as investments.
- Don't let the external libraries you use dictate your design.
- Prefer composition over inheritance to avoid binding your code to a specific hierarchy.
- Try to keep a code style that is easy to read from top down.
- Exit early from functions and avoid using else.
- Use "go to" or, even better, a local function to keep common code in one place.
- Avoid frivolous, redundant code comments that makes it impossible to see the tree from the forest.
- Write self-descriptive code by leveraging good naming for variables and functions.
- Divide functions into easy to digest sub-functions to keep the code as descriptive as possible.
- Write code comments when they are useful.

4

Tasty testing

This chapter covers:

- **Why we hate testing and how we can love it**
- **How to make testing more enjoyable**
- **Avoiding TDD, BDD and other three-letter acronyms**
- **Deciding on what to test**
- **Doing less work using tests**
- **Making tests spark joy**

Many software developers would liken testing to writing a book: it's tedious, nobody likes doing it, and it rarely pays off. Testing is considered like a second-class activity compared to coding, not doing *the real work*. Testers are subjected to preconceptions like they are having it too easy.

The reason behind the dislike for testing is that we developers see it as a process disconnected from building software. Building software is all about writing code from a programmer's perspective, whereas it's all about setting the right course for the team from a manager's vantage point. Similarly, for a tester, it's all about the quality of the product. We consider testing an external activity because of our perception that it's not part of the software development and we want to get involved as little as possible.

Testing can be an integral part of a developer's work and help them along the way. It can give you assurances that no other understanding of your code can give you. It can save you time, and you don't even need to hate yourself for it. Let's see how.

4.1 Types of tests

Software testing is about increasing the confidence in behavior of software. This is important: tests never guarantee a behavior, but they increase its likelihood quite a lot, as in

orders of magnitude. There are many ways to categorize different types of testing, but the most important distinction is how we run or implement it as it affects our time economy the most.

4.1.1 Manual testing

Testing can be a manual activity and it usually is for a developer. Developers test their code by running it and inspecting its behavior. Manual tests have their own types too, like end-to-end testing, which means testing every supported scenario on a software from beginning to end. The value that end-to-end testing provides is enormous but it's time-consuming.

Code reviews can be considered a way of testing, albeit weak. You can understand what the code does, and what it will do when run, to a certain extent. You can vaguely see how it fulfills the requirements, but you can't tell for sure. Tests, based on their types, can provide different levels of assurances about how the code will work. In that sense, code review can be considered a type of test.

What's a code review?

The main purpose of a code review was to examine a code before it gets pushed to the repository and find potential bugs in it. You could do it in a physical meeting together or use a web site like GitHub. Unfortunately, over the course of years, it has turned into many different things ranging from a rite of passage, which completely destroys the developer's self-esteem, to a pile of a software architect's unwarranted quotes from the articles they read.

The most important part of a code review is that it's the last moment that you can criticize the code without having to fix it yourself. After a piece of code passes the review, it becomes everyone's code. Because, you all have approved it. You can always say, "I wish you'd said that in the code review, Mark," whenever someone brings up your terrible $O(N^2)$ sort code and put your headphones back on. Just kidding. You should feel ashamed for writing an $O(N^2)$ sort code, especially after reading this book; and you still blame Mark? You should know better. Get along with your colleagues. You'll need them.

Ideally, code reviews are not about code style or formatting, because there are automated tools called either *linters* or *code analysis tools* that can check for those issues. It should be mainly about bugs and the technical debt that the code might introduce to other developers. Code review is async pair programming; it's a cost-efficient way to keep everyone on the same page and put their collective mind into identifying potential problems.

4.1.2 Automated tests

You are a programmer; you have the gift of writing code. That means you can make the computer do things for you, and that includes testing. You can write code that tests your code, so you don't have to. Programmers usually focus on creating tooling for only the software they're developing, not the development process itself, but that's equally important.

Automated tests can differ vastly in terms of their scope and, more importantly, how much they increase your confidence in the behavior of the software. The smallest variant of automated tests are *unit tests*. They are also the easiest to write because they test only a single unit of code: a public function. It needs to be public because testing is supposed to test externally visible interfaces rather than internal details of a class. The definition of unit

can sometimes change in the literature, be it a class or a module or some other logical arrangement of those, but I find functions as the target units convenient.

The problem with unit tests is that even though they let you to see if units work okay, they can't guarantee if they work okay *together*. So, you have to test if they get along together too. Those tests are called *integration tests*. Automated UI tests are usually integration tests too if they run the production code to build the correct user interface.

4.1.3 Living dangerously: testing in production

I had once bought a poster of a famous meme for one of our developers. It said, "I don't always test code, but when I do, I do it in *production*." I hung it on the wall right behind his monitor so he would always remember not to do it.

> **DEFINITION** In software lingo, the term production means a live environment accessed by actual users where any change affects the actual data. Many developers confuse it with their computer. There is "development" for that. "Development" as a name for runtime environment means code running locally on your machine and not affecting any data that harms production. As a precaution to harm production there is sometimes a production-like remote environment that is similar to production. It's sometimes called "staging," which doesn't affect actual data visible to your site's users.

Testing in production, aka live code, is considered a bad practice; no wonder such a poster exists. The reason is that when you find a failure, you might have already lost users or customers by then. More importantly, when you break production, there is a chance that you might break the workflow of the whole development team. You can easily understand that it happened by the disappointed looks and raised eyebrows you get if you're in an open office setting, text messages saying "WTF!!!!???"; Slack notification numbers increasing like KITT[1]'s speedometer; or the steam coming out of your boss' ears.

Like any bad practice, testing in production isn't always bad either. If the scenario you introduce isn't part of a frequently used, critical code path, you might get away with testing in production. That's why Facebook had this mantra, "move fast and break things" because they let the developers assess the impact of the change to the business. They later dropped the slogan after 2016 US elections, but it still has some substance. If it's a small break in an infrequently used feature, it might be okay to live through the fallout and fix it as soon as possible.

Even not testing your code can be okay if you think breaking a scenario isn't something your users would abandon the app for. I managed to run one of the most popular web sites in Turkey myself with zero automated tests in its first years, with a lot of errors and a lot of downtime of course, because hello: no automated tests!

[1] KITT, standing for Knight Industries Two Thousand, is a self-driving car equipped with voice recognition, depicted in the Sci-Fi TV series Knight Rider in the 80's. It's normal that you don't understand this reference as anybody who did is probably dead with the possible exception of David Hasselhoff. That guy is immortal.

4.1.4 Choosing the right testing methodology

You need to be aware of certain factors about a given scenario that you are trying to implement or change to decide how you want to test it. Those are mainly risk and cost. It's similar to what we used to calculate in our minds when our parents put us up to a chore.

- Cost
    - How much time do you need to spend to implement/run a certain test?
    - How many times will you need to repeat it?
    - If the code that is tested changes, who will know to test it?
    - How hard is it to keep the test reliable?
- Risk
    - How likely is this scenario to break?
    - If it breaks, how bad it will impact the business? How much money would you lose, aka "would this get me fired if it breaks"?
    - If it breaks, how many other scenarios will break along with this? For example, if your mailing feature stops working, many features depending on it will be broken too.
    - How frequent does the code change or how much do you anticipate that it will change in the future? Every change introduces a new risk.

You need to find a sweet spot that costs you the least and poses the least risk. Every risk is an implication of more cost to you. In time, you will have a mental tradeoff map for how much cost a test introduces and how much risk it poses, as in figure 4.1.

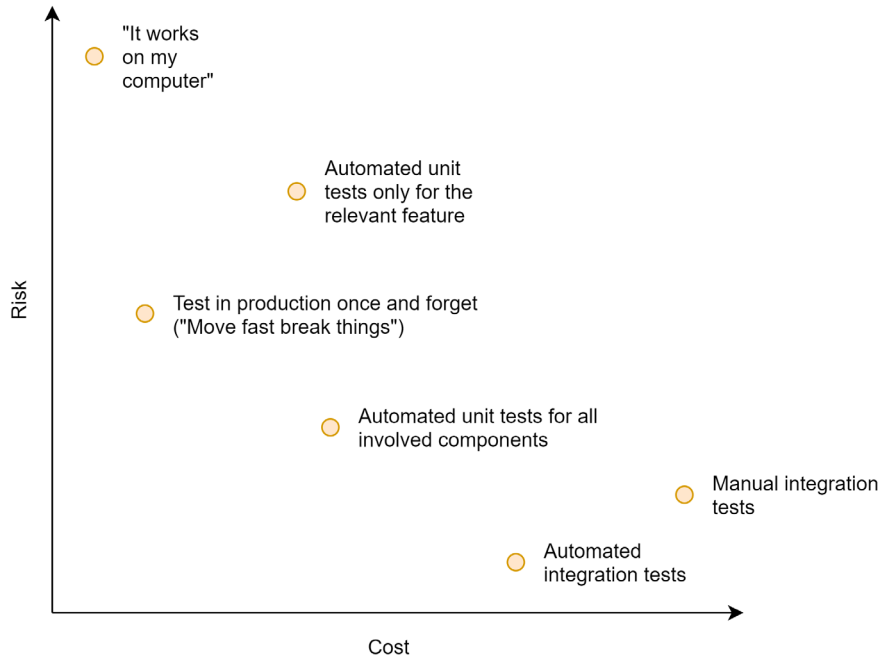


Figure 4.1 An example mental model to place different options of testing in your mind

Never say "it works on my computer" loudly to someone. It's for your internal thinking only. There will never be a code that you can describe as, "Well, it didn't work on my computer, but I was weirdly optimistic!" Of course, it works on your computer! Can you imagine deploying something that you cannot even run yourself? You can use it as a mantra while thinking about whether a feature should be tested or not as long as there is no chain of accountability present. If nobody makes you answer for your mistakes, then go for it. That means the company you're working for has a lot of excess budget to tolerate those mistakes.

If you need to fix your own bugs though, "It works on my computer" mentality puts you in a very slow and time-wasting cycle due to the delay between the deployment and the feedback loops. One of the basic problems with developer productivity is that interruptions cause significant delays. The reason is *the zone*. We already discussed how warming up to the code can get your productivity wheels turning. That mental state is sometimes called *the zone*. You're in the zone if you're in that productive state of mind. Similarly, getting interrupted can cause those wheels to stop and get you out of the zone, so it forces you to warm up again. As shown in figure 4.2, automated tests alleviate this problem by keeping you in the zone until you reach to a certain degree of confidence about a feature's completion. It shows you two different cycles where how expensive "it works on my computer" can be for both the business and the developer. Every time you get out of the zone you need to spend extra time entering it, which sometimes can even be more than the time you need to test your feature manually.

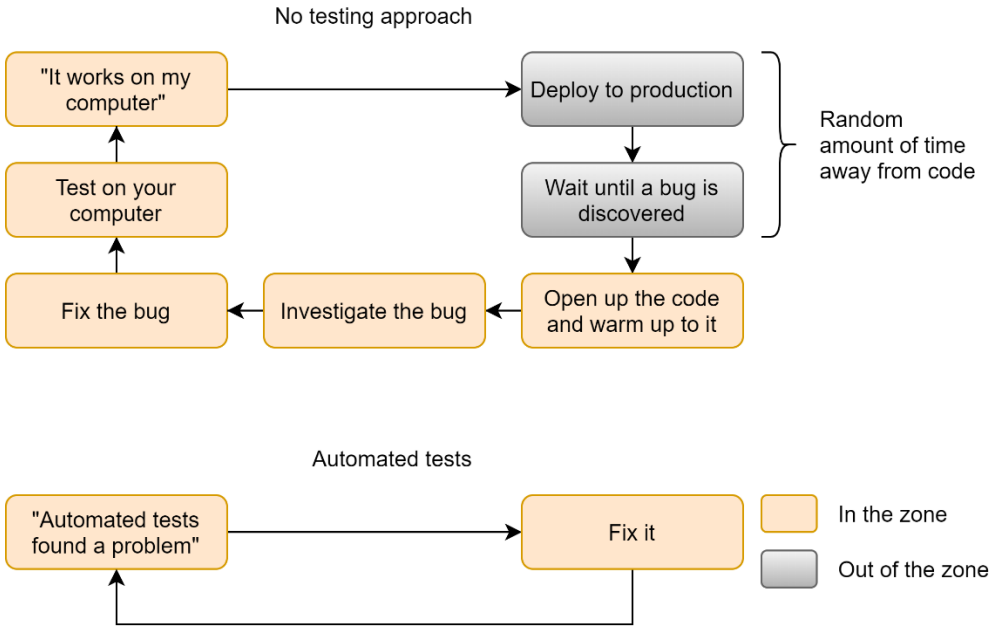


Figure 4.2 The expensive development cycle of "it works on my computer" versus "automated tests"

You can reach a similar quick cycle with manual integration tests too, but they just take more time. That's why automated tests are great: they keep you in the zone and cost you the least time. Arguably, writing and running tests can be considered as disconnected activities that might push you out of the zone. Yet, running unit tests is extremely fast and supposed to end in seconds. Writing tests is a slightly disconnected activity but it still makes you think about the code you've written. You might even consider it as a recap exercise.

This chapter is mostly about unit testing in general because it is in the sweet spot of cost versus risk in figure 4.1.

4.2 How to stop worrying and learn to love the tests

Unit testing is about writing test code that tests a single unit of your code, usually a function. You will encounter people arguing this, as what constitutes a unit. Basically, it doesn't matter much, as long as you can test a given unit in isolation. You can't test a whole class in a single test anyway. Every test actually tests only a single scenario for a function. So, it's usual to have multiple tests even for a single function.

Test frameworks make writing tests as easy as possible, but they are not necessary. A test suite can be simply a separate program that runs the tests and shows the results. As a matter of fact, that was the only way to test your program before test frameworks were a thing. I'd like to show you simple code and how unit testing is evolved over time to write tests for a given function as easy as possible.

Let's consider that you are tasked with changing how the post dates are displayed on a microblogging web site called Blabber. The post dates were displayed as a full date, and according to the new social media fashion, it's more favorable to use acronyms that shows a duration since the post was created in seconds, minutes, hours, and so forth. You need to develop a function that gets a `DateTimeOffset` and converts it into a string that shows a duration since that span of time in text like "3h" for three hours, "2m" for two minutes, or "1s" for one second. It should show only the most significant unit. If the post is three hours two minutes and one second old, it should only show "3h". Listing 4.1 shows such a function.

In listing 4.1, we define an *extension method* to `DateTimeOffset` class in .NET, so we can call it wherever we want like a native method of `DateTimeOffset`.

Avoid polluting code completion with extension methods

C# provides a nice syntax to define additional methods for a type even if you don't have access to its source. If you prefix the first parameter of a function with `this` keyword, it starts to appear in that type's method list in code completion. It's so convenient that developers like extension methods a lot and tend to make everything an extension method instead of a static method. Say, you have a simple method like this:

```
static class SumHelper {
  static int Sum(int a, int b) => a + b;
}
```

In order to call this method, you have to write `SumHelper.Sum(amount, rate);` and more importantly, you must know that there is a class called `SumHelper`. You can write it as an extension method instead like this:

```
static class SumHelper {
  static decimal Sum(this int a, int b) => a + b;
}
```

Now, you can call the method like this:

```
int result = 5.Sum(10);
```

Looks good but there is a problem. Whenever you write an extension method for a well-known class like string or int, you introduce it to code completion, which is the dropdown you see on Visual Studio when you type a dot after an identifier. It can be extremely annoying to struggle to find the method you're looking for among the list of completely irrelevant methods.

Do not introduce a purpose-specific method into a commonly used .NET class. Do that only for generic methods that will be used commonly. For example, a "Reverse" method in String class can be okay, but "MakeCdnFilename" wouldn't. Reverse can be applicable in any context, but MakeCdnFilename would only be needed when you must, well, make a filename suitable for the content delivery network you're using. Other than that, it's a nuisance for you and every developer in your team. Don't make people hate you. More importantly, don't make yourself hate you. In those cases, you can perfectly use a static class and a syntax like: Cdn.MakeFilename().

Don't create an extension method when you can make the method the part of the class. It only makes sense to do that when you want to introduce a new functionality beyond a dependency boundary. For example, you might have a web project that uses a class defined in a library that doesn't depend on web components. Later, you might want to add a specific functionality to that class related to web functionality in the web project. It's more preferable to

introduce new dependency only to the extension method in the web project, rather than making the library depend on your web components. Unnecessary dependencies can tie your shoelaces together.

We calculate the interval between current time and the post time and check its fields to find out the most significant unit of the interval and return the result based on it.

Listing 4.1 A function that converts a date to a string representation of the interval

```
public static class DateTimeExtensions {
  public static string ToIntervalString(
    this DateTimeOffset postTime) {    #A
    TimeSpan interval = DateTimeOffset.Now - postTime;    #B
    if (interval.TotalHours >= 1.0) {     #C
      return $"{(int)interval.TotalHours}h";    #C
    }    #C
    if (interval.TotalMinutes >= 1.0) {    #C
      return $"{(int)interval.TotalMinutes}m";    #C
    }    #C
    if (interval.TotalSeconds >= 1.0) {    #C
      return $"{(int)interval.TotalSeconds}s";    #C
    }    #C
    return "now";
  }
}
```

#A `this` **defines an extension method to DateTimeOffset class.**
#B Calculate the interval.
#C It's possible to write this code shorter or more performant, but not when it sacrifices readability.

We have a vague spec about the function, and we can start writing some tests for it. It'd be good idea to write possible inputs and expected outputs in a table to ensure the function works correctly, as in table 4.1.

Table 4.1 A sample test specification for our conversion function

Input	Output
< 1 second	"now"
< 1 minute	"<seconds>s"
< 1 hour	"<minutes>m"
>= 1 hour	"<hours>h"

If DateTimeOffset is a class, we should also be testing for the case when we pass null, but because it's a struct, it cannot be null. That saved us one test. Normally, you don't really need to create a table like that, and you can usually manage a mental model of it, but whenever you're in doubt, feel free to write it down.

Our tests should consist of calls with different `DateTimeOffset`'s and comparisons with different strings. At this point, test reliability becomes a concern because `DateTime.Now` always changes, and our tests are not guaranteed to run in specific time. If there was

another test running, or something slows down the computer, you can easily fail the test for the output "now". That means our tests will be flaky and can fail occasionally.

That indicates a problem with our design. A simple solution would be that we could make our function deterministic by passing a `TimeSpan` instead of a `DateTimeOffset` and calculating the difference in the caller instead. As you can see, writing tests around your code helps you identify design problems too, and that's one of the selling points of TDD (Test-Driven Development) approach, which we didn't use here, because we know that we can just go ahead and change the function easily, as in listing 4.2, to receive a `TimeSpan` directly.

Listing 4.2 Our refined design

```
public static string ToIntervalString(
  this TimeSpan interval) {    #A
  if (interval.TotalHours >= 1.0) {
    return $"{(int)interval.TotalHours}h";
  }
  if (interval.TotalMinutes >= 1.0) {
    return $"{(int)interval.TotalMinutes}m";
  }
  if (interval.TotalSeconds >= 1.0) {
    return $"{(int)interval.TotalSeconds}s";
  }
  return "now";
}
```

#A We receive a TimeSpan instead.

Our test cases didn't change, but our tests will be much more reliable. More importantly, we decoupled two different tasks, calculating the difference between two dates and converting an interval to a string representation. Deconstructing concerns in code can help you achieve better designs. It can also be a chore to calculate differences and you can have a separate wrapper function for that.

Now, how do we make sure our function works? We can simply push it to production and wait a couple minutes to hear any screams. If not, we're good to go. By the way, is your resumé up to date? No reason, just asking.

We can write a program that tests the function and see the results. An example program would be like in listing 4.3. It's a plain console application that references our project and uses `Debug.Assert` method in `System.Diagnostics` namespace to make sure it passes. It ensures that the function returns expected values. Because asserts run only in Debug configuration, we also ensure that the code isn't run in any other configuration at the beginning with a compiler directive.

Listing 4.3 A primitive unit testing code

```
#if !DEBUG    #A
#error Debug.Asserts will only run in Debug configuration
#endif
using System;
using System.Diagnostics;
namespace DateUtilsTests {
```

```
  public class Program {
    public static void Main(string[] args) {
      var span = TimeSpan.FromSeconds(3);    #B
      Debug.Assert(span.ToIntervalString() == "3s", "3s case failed");    #B
      span = TimeSpan.FromMinutes(5);    #C
      Debug.Assert(span.ToIntervalString() == "5m", "5m case failed");    #C
      span = TimeSpan.FromHours(7);    #D
      Debug.Assert(span.ToIntervalString() == "7h", "7h case failed");    #D
      span = TimeSpan.FromMilliseconds(1);    #E
      Debug.Assert(span.ToIntervalString() == "now", "now case failed");    #E
    }
  }
}
```

#A We need the preprocessor statement to make asserts work.
#B Test case for seconds
#C Test case for minutes
#D Test case for hours
#E Test case for less than a second

So, why do we need unit test frameworks? Can't we write all tests like this? We could, but it would need more work. In our example, you'll note the following:

- There is no way to detect if any of the tests failed from an external program, such as a build tool. We need special handling around that. Test frameworks and test runners coming with them handle that easily.
- The first failing test would cause the program to terminate. That will lose us time if we have many more failures. We have to run tests again and again, wasting time. Test frameworks can run all tests and report the failures altogether, like compiler errors.
- It's impossible to run certain tests selectively. You might be working on a specific feature and want to debug the function you wrote by debugging the test code. Test frameworks allow you to debug specific tests without having to run the rest.
- Test frameworks can produce code coverage report which helps you identify missing test coverage on your code. It's not possible by writing ad-hoc test code. If you happen to write a coverage analysis tool, you might as well work on creating a test framework.
- Although those tests don't depend on each other, they run sequentially, taking the longest time. Normally, that's not a problem with small number of test cases, but in a medium-scale project, you can have thousands of tests taking different amount of times. You can create threads and run the tests in parallel but that's too much work. Test frameworks can do all of that with a simple switch.
- When an error happens, you only know that there is a problem, but you have no idea about its nature. Strings mismatched, so, what kind of mismatch it is? Did the function return null? Was there an extra character? Test frameworks can report these details too.
- Anything other than using .NET provided Debug.Assert will require us writing extra code, a scaffolding if you will. I mean if you start going that path, using an existing framework is much better.
- You'll have the opportunity to join never-ending debates about which test framework is better and feel superior for completely wrong reasons.

Now, let's try writing the same tests with a test framework in listing 4.4. Many test frameworks look alike with the exception of xUnit which is supposedly developed by extraterrestrial life-forms visiting Earth, but in principle it shouldn't matter which framework you're using with the exception of slight changes in the terminology. We're using NUnit here, but you can use any framework you want. You'll see how much clearer the code is with a framework. Most of our test code is actually pretty much a text version of our input/output table in table 4.1. It's apparent what we're testing, and more importantly, although we only have a single test method, we have the capability to run or debug each test individually in the test runner. The technique we used in listing 4.4 with `TestCase` attributes is called *parameterized test*. If you have a specific set of inputs and outputs, you can simply declare them as data and use it in the same function over and over, avoiding repetition over writing a separate test for each test. Similarly, by combining `ExpectedResult` values and declaring the function with a return value, you don't even need to write `Assert`'s explicitly. The framework does it automatically. Less work!

You can run these tests in Test Explorer window of Visual Studio (View → Test Explorer), or you can run dotnet test from the command prompt, or you can even use a third party test runner like NCrunch. The test results in Visual Studio's Test Explorer will look like as in figure 4.3.

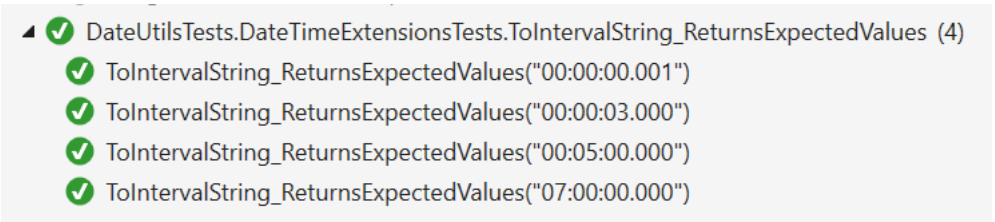


Figure 4.3 Test results that you can't take your eyes off of

Listing 4.4 Test framework magic

```
using System;
using NUnit.Framework;
namespace DateUtilsTests {
  class DateUtilsTest {
    [TestCase("00:00:03.000", ExpectedResult = "3s")]
    [TestCase("00:05:00.000", ExpectedResult = "5m")]
    [TestCase("07:00:00.000", ExpectedResult = "7h")]
    [TestCase("00:00:00.001", ExpectedResult = "now")]
    public string ToIntervalString_ReturnsExpectedValues(
      string timeSpanText) {
      var input = TimeSpan.Parse(timeSpanText);    #A
      return input.ToIntervalString();    #B
    }
  }
}
```

#A Converting string to our input type
#B No assertions!

You can see how a single function is actually broken into four different functions during test running phase, and how its arguments are displayed along with the test name in figure 4.3. More importantly, you can select a single test, run it, or debug only that test case. And if a test fails you see a brilliant report that exactly tells what's wrong with your code. Say, you accidentally wrote "nov" instead of "now". The test error would show up like this:

```
Message:
      String lengths are both 3. Strings differ at index 2.
      Expected: "now"
      But was:  "nov"
      -------------^
```

Not only do you see that there is an error but you also see a clear explanation about where it happened.

It's a no-brainer to use test frameworks, and you get to love writing tests more when you're aware of how they save you from extra work. They are NASA pre-flight check lights, "system status nominal" announcements, they are your little nanobots doing their work for you. Love tests, love test frameworks.

4.3 Don't use TDD or other acronyms

Unit testing, like every successful religion, has split into factions. Test Driven Development (TDD), and Behavior Driven Development (BDD) are some examples. I've come to believe that there are people in software industry who really love to create new paradigms and standards to be followed without question, and there are people who just love to follow without questioning. We love prescriptions and rituals because all you need to do is to follow them, without much thinking. That can cost you a lot of time and make you hate testing too.

The idea behind TDD is that writing tests before actual code can guide you to write better code. TDD prescribes that you should write tests for a class first before writing a single line of code of that class, so the code you write constitutes a guideline on how to implement the actual code. You write your tests. It fails to compile. You start writing actual code, it compiles. Then you run tests and they fail. Then you fix the bugs in your code to make the tests pass. BDD is also a test-first approach with differences in naming and layout of tests.

The philosophy behind TDD/BDD isn't completely rubbish. When you think about how a code should be tested first, it can influence how you think about its design. The problem with TDD isn't the mentality but the practice, the ritualistic approach: write tests and, because the actual code is still missing, get a compiler error (wow, really, Sherlock?); after writing the code, fix the test failures. I hate errors. They make me feel unsuccessful. Every red squiggly line in the editor, every STOP sign in the Errors list window, every warning icon is a cognitive load, making me confused and distracted.

When you focus on the test before writing a single line of code, you start thinking more about tests than your own problem domain. You start thinking about better ways to write tests. Your mental space gets allocated to the task of writing tests, the test framework's syntactic elements, organization of tests, rather than the production code itself. That's not the goal of testing. Tests shouldn't make you think. Tests should be the easiest piece of code you can write. If that's not the case, you're doing it wrong.

Having tests before writing code triggers the sunk cost fallacy. Remember how dependencies made your code more rigid in chapter 3? Surprise, tests depend on your code too. When you have a full-blown test suite at hand, you become disinclined to change the design of the code because that would mean changing the tests too. It reduces your flexibility when prototyping code. Arguably, tests can give some ideas about if the design really works or not but only in isolated scenarios. You might later discover that a prototype doesn't work well with other components and change your design, before writing any tests. That could be okay if you spend a lot of time on drawing board when designing but that's not usually the case in the streets. You need the ability to quickly change your design.

You can consider writing tests when you believe you're mostly done with your prototype and it seems to be working out okay. Yes, tests will make your code harder to change then, but at the same time it will compensate that by making you confident in the behavior of your code, letting you make changes more easily. You'll effectively get faster.

4.4 Write tests for your own good

- Yes, writing tests improves the software, but it also improves your living standards too. We already discussed how writing tests first can constrain you from changing your code's design. Yet, writing tests last can make your code more flexible because you can easily make significant changes later, without worrying about breaking the behavior after you forget about the code completely. It frees you. It works as an insurance, almost the inverse of sunk cost fallacy. The difference of writing tests after is that you are not discouraged in a rapid iteration phase like prototyping. You need to overhaul a code? The first step you need to take is to write tests for it.
- Writing tests after you have a good prototype works as a recap exercise for your design. You go over the whole code once again with tests in mind. You can identify certain problems that you didn't find when prototyping your code.
- Remember how we discussed doing small, trivial fixes in the code can get you warmed up to large coding tasks? Well, writing tests is one of the great ways to do that. Find missing tests and add them. It never hurts to have more tests unless they're redundant. They don't have to be related to your upcoming work. You can simply blindly add test coverage, and who knows, you might find bugs while doing it.
- Tests can act as a specification, or documentation if they’re written in a clear, easy to understand way. Code for each test should describe the input and the expected output of a function by how it’s written and how it’s named. Code may not be the best way to describe something, but it’s a thousand times better than having nothing at all.
- Do you hate your colleagues breaking your code? Tests are there to help. Tests enforce the contract between the code and the specification that developers can't break. You won't have to have comments like this:

```
// When this code was written,
// only God and I knew what it did.
// Now only God knows.
```

(That infamous comment block is a derivative joke originally attributed to the author John Paul Friedrich Richter who lived in the 19th century. He didn't write a single line of code, only comments. https://quoteinvestigator.com/2013/09/24/god-knows/)

Tests assure you that a fixed bug will remain fixed and it won't appear again. Every time you fix a bug, adding a test for it, will make sure you won't have to deal with that bug again, ever. Otherwise, who knows when another change won't trigger it again? Tests are critical timesavers when used like that.

Tests improve both the software and the developer. Write tests to be a more efficient developer.

4.5 Deciding on what to test

> *"That is not halted which can eternal run,*
> *And with strange eons, even tests may be down" H.P. Codecraft*

Writing one test, and seeing it pass is only the half of the story. It doesn't mean your function works. Will it fail when the code breaks? Do you cover all the possible scenarios? What should you be testing for? If your tests don't help you to find bugs, they are failures already.

One of my managers had this manual technique to ensure that his team wrote reliable tests: He removed random lines of code from the production code and ran tests again. If your tests passed, that meant you failed.

There are better approaches to find out what cases to test. A specification is a great starting point, but you rarely have those in the streets. It might make sense to create a specification yourself, but even if the only thing you have is code, there are some ways to identify what to test.

4.5.1 Respect boundaries

You can call a function that receives a simple integer with four billion different values. Does that mean that you have to test for if your function works for each one of those? No. Instead, you should try to identify which input values cause the code to diverge into a branch, or cause values to overflow and test values around those.

Consider a function that checks if a birthdate is of legal age for the registration page of your online game. It's trivial for anyone who was born 18 years before (assuming 18 is the legal age for your game): you just subtract the years and check if it's at least 18. But what if that person has become 18 last week? Are you going to deprive that person of enjoyment of your pay-to-win game with mediocre graphics? Of course not.

Let's define a function `IsLegalBirthdate`. We use a `DateTime` class instead of `DateTimeOffset` to represent a birthdate because birthdates don't have time zones. If you were born on December 21st in Samoa, your birthday is December 21st everywhere in the world, even in American Samoa, which is 24 hours ahead of Samoa despite being only a hundred miles away. I'm sure there is intense discussion every year about when to have relatives over for Christmas dinner. Time zones are weird.

Anyway, we first calculate the year difference. The only time we need to look at exact dates is for the year of that person's 18th birthday. If it's that year, we check the month and the day. Otherwise, we only check whether the person is older than eighteen. We use a constant to signify legal age instead of writing the number everywhere. Because writing the number is susceptible to typos, and when your boss comes asking you, "Hey can you raise the legal age to 21?" you only have one place to edit it out in this function. You also avoid having to write "// legal age" next to every 18 in the code to explain it. It suddenly becomes self-explanatory. Every conditional in the function—which encompasses if statements, while loops, switch cases, and so forth—causes only certain input values to exercise the code path inside. That means we can split the range of input values based on the conditionals, depending on the input parameters. In our example in listing 4.5, we don't need to test for all possible `DateTime` values between January 1st, 1 and December 31st, 9999, which is about 3.6 million. We only need to test for seven different inputs.

Listing 4.5 The bouncer's algorithm

```
public static bool IsLegalBirthdate(DateTime birthdate) {
  const int legalAge = 18;
  var now = DateTime.Now;
  int age = now.Year - birthdate.Year;
  if (age == legalAge) {    #A
    return now.Month > birthdate.Month    #A
      || (now.Month == birthdate.Month    #A
          && now.Day > birthdate.Day);    #A
  }
  return age > legalAge;    #A
}
```

#A Conditionals in the code

The seven input values are listed in Table 4.2:

Table 4.2 Partitioning input values based on conditionals

#	Year difference	Month of birthdate	Day of birthdate	Expected result
1	= 18	= Current month	< Current day	true
2	= 18	= Current month	= Current day	false
3	= 18	= Current month	> Current day	false
4	= 18	< Current month	Any	true
5	= 18	> Current month	Any	false
6	> 18	Any	Any	true
7	< 18	Any	Any	false

We suddenly brought down our number of cases from 3.6 million to 7, simply by identifying conditionals. Those conditionals that split the input range are called *boundary conditionals* because they define the boundaries for input values for possible code paths in the function.

So, we can go ahead and write tests for those input values as shown in listing 4.6. We basically create a clone of our test table in our inputs and convert it to a `DateTime` and run through our function. We can't hardcode `DateTime` values directly into our input/output table because a birthdate's legality changes based on the current time.

We could convert this to a `TimeSpan`-based function as we did before, but legal age isn't based on exact number of days, but an absolute date time instead. This table is also better because it reflects your mental model more accurately. We use `-1` for less than, `1` for greater than, `0` for equality, and prepare our actual input values using those values as references.

Listing 4.6 Creating our test function from our input/output table

```
[TestCase(18,  0, -1, ExpectedResult = true)]
[TestCase(18,  0,  0, ExpectedResult = false)]
[TestCase(18,  0,  1, ExpectedResult = false)]
[TestCase(18, -1,  0, ExpectedResult = true)]
[TestCase(18,  1,  0, ExpectedResult = false)]
[TestCase(19,  0,  0, ExpectedResult = true)]
[TestCase(17,  0,  0, ExpectedResult = false)]
public bool IsLegalBirthdate_ReturnsExpectedValues(
  int yearDifference, int monthDifference, int dayDifference) {
  var now = DateTime.Now;
  var input = now.AddYears(-yearDifference)    #A
    .AddMonths(monthDifference)    #A
    .AddDays(dayDifference);    #A
  return DateTimeExtensions.IsLegalBirthdate(input);
}
```

#A Preparing our actual input here.

We did it! We narrowed down the number of possible inputs and identified exactly what to test in our function to create a concrete test plan.

Whenever you need to find out what to test in a function, you're supposed to start with a specification. In the streets though, you'll likely figure out that a specification has never existed or was obsoleted a long time ago so, the second-best way would be to start with boundary conditionals. Using parameterized tests also helps us to focus on what to test rather than writing repetitive test code. It's occasionally inevitable to create a new function for each test, but specifically with data-bound tests like this, parameterized tests save you considerable time.

4.5.2 Code coverage

Code coverage is magic, and like magic, it's mostly stories. Code coverage is measured by injecting every line of your code with callbacks to trace how far the code called by a test executes and which parts it misses. That way, you can find out which part of the code isn't exercised and therefore missing tests.

Development environments rarely come with code-coverage measurement tools out of the box. They are either in astronomically priced versions of Visual Studio, or other paid third-party tools like NCrunch, dotCover, and NCover. Codecov (https://codecov.io) is a service that can work with your online repository, and they have a free plan. Free code-

coverage measurement locally in .NET was possible only with Coverlet library and Code Coverage reporting extensions in Visual Studio Code at the time of drafting this book.

Code-coverage tools tell you which parts of your code ran when you run your tests. That's quite handy to see what kind of test coverage you're missing to exercise all code paths. It's not the only part of the story, and it's certainly not the most effective. You can have 100% code coverage and still have missing test cases. We'll discuss them later in the chapter.

Assume that we comment out the tests where calls to our `IsLegalBirthdate` function with a birthdate that is exactly 18 years old, as in listing 4.7.

Listing 4.7 Missing tests

```
//[TestCase(18,  0, -1, ExpectedResult = true)]    #A
//[TestCase(18,  0,  0, ExpectedResult = false)]    #A
//[TestCase(18,  0,  1, ExpectedResult = false)]    #A
//[TestCase(18, -1,  0, ExpectedResult = true)]    #A
//[TestCase(18,  1,  0, ExpectedResult = false)]    #A
[TestCase(19,  0,  0, ExpectedResult = true)]
[TestCase(17,  0,  0, ExpectedResult = false)]
public bool IsLegalBirthdate_ReturnsExpectedValues(
  int yearDifference, int monthDifference, int dayDifference) {
  var now = DateTime.Now;
  var input = now.AddYears(-yearDifference)
    .AddMonths(monthDifference)
    .AddDays(dayDifference);
  return DateTimeExtensions.IsLegalBirthdate(input);
}
```

#A Commented-out test cases

In that case, a tool like NCrunch, for example, would show the missing coverage as in figure 4.4. The coverage circle next to the return statement inside the if statement is grayed out because we never call the function with a parameter that matches the condition `age == legalAge`. That means we're missing some input values.

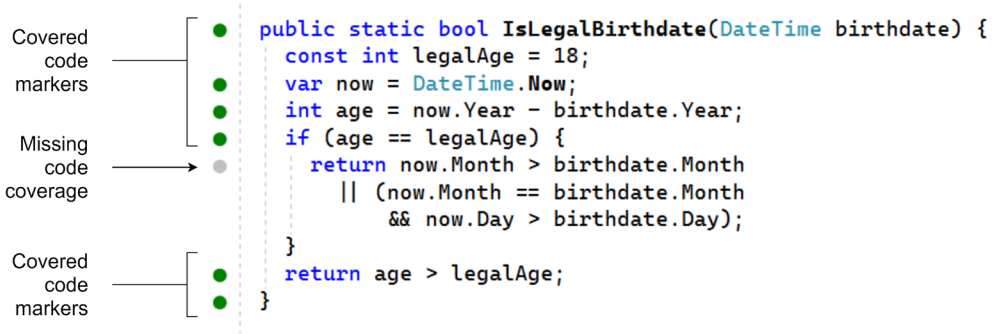


Figure 4.4 Missing code coverage

When you uncomment those commented out test cases and run tests again, code coverage shows that you have 100% code coverage.

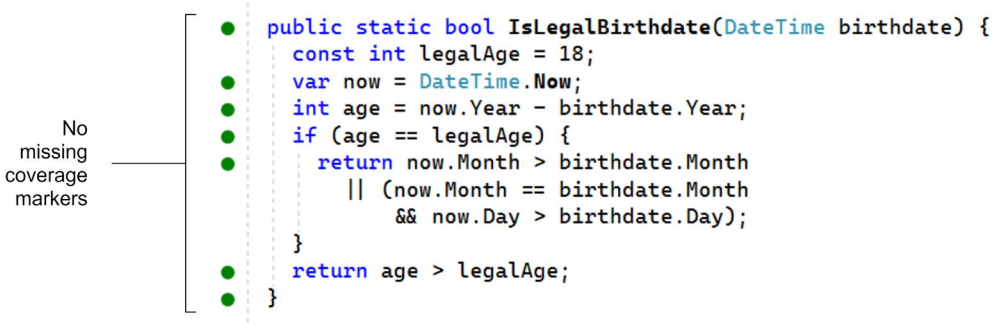


Figure 4.5 Full code coverage

Code coverage tools are a good starting point but they are not fully effective to show you actual test coverage. You should still have a good understanding of the range of input values and boundary conditionals. 100% code coverage doesn't mean 100% test coverage. Consider the following function where you need to return an item from list by index:

```
public Tag GetTagDetails(byte numberOfItems, int index) {
  return GetTrendingTags(numberOfItems)[index];
}
```

Calling that function with `GetTagDetails(1, 0);` would succeed and we would immediately achieve 100% code coverage. Would we have tested all the possible cases? No. Our input coverage would nowhere be close. What if `numberOfItems` is zero and `index` is non-zero? What happens if index is negative?

They all mean that we shouldn't be focusing solely on code coverage and trying to fill all the gaps. Instead, we should be conscious about our test coverage by taking all possible inputs into account, being smart about the boundary values. They are not mutually exclusive though; you can use both approaches at the same time.

4.6 Don't write tests

Yes, testing is helpful, but nothing's better than to avoid writing tests completely. How do you get away without writing tests and still keep your code reliable?

4.6.1 Don't write code

If a piece of code doesn't exist, it doesn't need to be tested either. There are no bugs in deleted code. Think about this when writing code. Is that something worth writing tests for? Maybe, you don't need to write that code at all. For example, can you opt for using an existing package over implementing it from scratch? Can you leverage an existing class that does the exact same thing you are trying to implement? For example, you might be tempted

to write custom regular expressions for validating URL's while all you need to do is to leverage `System.Uri` class.

Third-party code isn't guaranteed to be perfect or always suitable for your purposes of course. You might later discover that the code doesn't work for you. It's usually worth taking that risk before trying to write something from scratch. Similarly, the same codebase you're working on might have the code doing the same job implemented by a colleague. Search your code base to see if something's there.

If nothing works, be ready to implement your own. Don't be scared of reinventing the wheel. It can be very educational, as we discussed in chapter 3.

4.6.2 Don't write all the tests

The famous *Pareto principle* states that 80% of consequences are the results of 20% of the causes. At least, that's what 80% of the definitions say. It's more commonly called the *80/20 principle*. It's applicable in testing too. You can get 80% reliability from 20% test coverage if you choose your tests wisely.

Bugs don't appear homogenously. Not every code line has the same probability of producing a bug. It's more likely to find bugs in more commonly used code, and code with high churn. You can call those areas of the code *hot paths* where a problem is more likely to happen.

That's exactly what I did with my web site. It had no tests whatsoever even after it became one of the most popular Turkish web sites in the world. Then, I had to add tests because too many bugs started to appear with the text markup parser. The markup was custom, it barely resembled Markdown, but I developed it before Markdown was even a vitamin in the oranges Dave Gruber ate. Because parsing logic was complicated and prone to bugs, it became economically infeasible to fix every issue after deploying to production. I developed a test suite for it. That was before the advent of test frameworks, so I had to develop my own. I incrementally added more tests as more bugs appeared, because I hated creating the same bugs, and we developed a quite extensive test suite later, which saved us thousands of failing production deployments. Tests just work.

Even just viewing your web site's home page provides a good amount of code coverage, because it exercises many shared code paths with other pages. That's called *smoke testing* around the block. It comes from the times when they developed the first prototype of the computer and just tried to turn it on to see if smoke came out of it. If there was no smoke, that was pretty much a good sign. Similarly, having good test coverage for critical, shared components is more important than having 100% code coverage. Don't spend hours just to add test coverage to that missing line in the constructor of a class if it won't make much difference. You already know that code coverage isn't the whole story.

4.7 Let the compiler test your code

With a strongly-typed language, you can leverage the type system to reduce the number of test cases needed. We already discussed how nullable references can help you to avoid null checks in the code, which also reduces the need to write tests for null cases. Let's go over a simple example. We already validated that if the person who wants to register is at least 18

years old in the previous section. We now need to validate if the chosen username is valid, so we need a function that validates usernames.

4.7.1 Eliminate null checks

Let our rule for a username be lowercase alphanumeric characters up to eight characters long. A regular expression pattern for such a username would be `"^[a-z0-9]{1,8}$"`. We can write a username class as in listing 4.8. We define a Username class to represent all usernames in the code. We avoid the need to think about where we should validate our input by passing this to any code that requires a username.

In order to make sure that a username is never invalid, we validate the parameter in the constructor and throw an exception if it's not in the correct format. Apart from the constructor, the rest of the code is boilerplate to make it work in comparison scenarios. Remember, you can always derive such a class by creating a base `StringValue` class and write minimal code for each string-based value class. I wanted these to remain here to be explicit about what the code entails. Notice the use of `nameof` operator instead of hard-coded strings for references to parameters. It lets you keep names in-sync after renaming. It can be used for fields and properties too, especially useful for test cases where data is stored in a separate field, and you have to refer to it by its name.

Listing 4.8 A username value type implementation

```
public class Username {
  public string Value { get; private set; }
  private const string validUsernamePattern = @"^[a-z0-9]{1,8}$";

  public Username(string username) {
    if (username is null) {    #A
      throw new ArgumentNullException(nameof(username));
    }
    if (!Regex.IsMatch(username, validUsernamePattern)) {    #A
      throw new ArgumentException(nameof(username),
        "Invalid username");
    }
    this.Value = username;
  }

  public override string ToString() => base.ToString();    #B
  public override int GetHashCode() => Value.GetHashCode();    #B
  public override bool Equals(object obj) {    #B
    return obj is Username other && other.Value == Value;    #B
  }    #B
  public static implicit operator string(Username username) {    #B
    return username.Value;    #B
  }    #B
  public static bool operator==(Username a, Username b) {    #B
    return a.Value == b.Value;    #B
  }    #B
  public static bool operator !=(Username a, Username b) {    #B
    return !(a == b);    #B
  }    #B
}
```

#A We validate the username here, once and for all.
#B Our usual boilerplate to make a class comparable.

Myths around regular expressions

Regular expressions are one of the most brilliant inventions in the history of computer science. We owe them to the venerable Stephen Cole Kleene. They let you create a text parser out of a couple of characters. The pattern `"light"` matches only the string "light" while `"[ln]ight"` matches both "light" and "night". Similarly, `"li(gh){1,2}t"` matches only the words "light" and "lighght" which is not a typo but a single-word Aram Saroyan poem.

Jamie Zawinski famously said, "Some people, when confronted with a problem, think 'I know, I'll use regular expressions.' Now they have two problems." The phrase "regular expression" implies certain parsing characteristics. Regular expressions are not context aware therefore you can't use a single regular expression to find innermost tag in an HTML document or detect unmatched closing tags. That means they are not suitable for complicated parsing tasks. Yet, you can use them to parse text with a non-nested structure.

Regular expressions are surprisingly performant for the cases they are suitable for. If you need extra performance, you can pre-compile them in C# by creating a `Regex` object with the option `RegexOptions.Compiled`. That means a custom code that parses a string based on your pattern will be created on demand. Your pattern turns into C# and eventually machine code. Consecutive calls to the same Regex object will reuse the compiled code, gaining you performance for multiple iterations.

Despite how performant they are, you shouldn't use regular expressions when a simpler alternative exists. If you need to check if a string is a certain length, a simple `"str.Length == 5"` would be way faster and more readable than `"Regex.IsMatch(@"^.{5}$", str)"`. Similarly, the string class contains many performant methods for common string check operations like `StartsWith`, `EndsWith`, `IndexOf`, `LastIndexOf`, `IsNullOrEmpty`, and `IsNullOrWhiteSpace`. Always prefer them over regular expressions for their specific use cases.

That said, it's also important for you to know at least basic syntax of regular expressions because they can be powerful in a development environment too. You can manipulate code in quite complicated ways that can save you from hours of work and all popular text editors support regular expressions for find and replace operations. I'm talking about operations like "I want to move hundreds of bracket characters in the code to the next line only when they appear next to a code line". You can think about correct regular expression pattern for a couple of minutes as opposed to doing it manually for an hour.

Testing the constructor of Username would require us to create three different test methods as in listing 4.9. One for nullability because a different exception type is raised, the other one is for non-null but invalid inputs, and finally for the valid inputs because we need to make sure that it recognizes valid inputs as valid too.

Listing 4.9 Tests for `Username` class

```
class UsernameTest {
  [Test]
  public void ctor_nullUsername_ThrowsArgumentNullException() {
    Assert.Throws<ArgumentNullException>(
      () => new Username(null));
  }

  [TestCase("")]
  [TestCase("Upper")]
  [TestCase("toolongusername")]
```

```
  [TestCase("root!!")]
  [TestCase("a b")]
  public void ctor_invalidUsername_ThrowsArgumentException(string username) {
    Assert.Throws<ArgumentException>(
      () => new Username(username));
  }

  [TestCase("a")]
  [TestCase("1")]
  [TestCase("hunter2")]
  [TestCase("12345678")]
  [TestCase("abcdefgh")]
  public void ctor_validUsername_DoesNotThrow(string username) {
    Assert.DoesNotThrow(() => new Username(username));
  }
}
```

Had we enabled nullable references for the project Username class was in, we wouldn't need to write tests for the null case at all. The only exception to that would be when writing a public API, which may not run against a nullable-references-aware code. In that case, you'd still need to check against nulls.

Similarly, declaring Username as a `struct` when suitable would make it a value type, which would also remove the requirement for a null check. Using correct types and correct structures for types would help you reducing number of tests. The compiler would ensure the correctness of our code instead.

Using specific types for our purposes reduces the need for tests. When your registration function receives a Username instead of a string, you don't need to check if registration function validates its arguments. Similarly, when your function receives a URL argument as a Uri class, you don't need to check if your function processes the URL correctly anymore.

4.7.2 Eliminate range checks

You can use unsigned integer types to reduce the surface area of invalid input space. You can see unsigned versions of primitive integer types in table 4.3. There you can see the varieties of data types with their possible ranges which might be more suitable for your code. It's also important that you keep in mind whether the type is directly compatible with int or not as it's the go-to type of .NET for integers. You probably have already seen these types, but you might not have considered that they can save you from writing extra test cases. For example, if your function needs only positive values, then why bother with int and checking for negative values and throwing exceptions? Just receive `uint` instead.

Table 4.3 Alternative integer types with different value ranges

Name	Integer type	Value range	Assignable to int without loss?
`int`	32-bit signed	-2147483648..2147483647	Duh
`uint`	32-bit unsigned	0..4294967295	No
`long`	64-bit signed	-9223372036854775808..9223372036854775807	No

ulong	64-bit unsigned	0..18446744073709551615	No
short	16-bit signed	-32768..32767	Yes
ushort	16-bit unsigned	0..65535	Yes
sbyte	8-bit signed	-128..127	Yes
byte	8-bit unsigned	0..255	Yes

When you use an unsigned type, trying to pass a negative constant value to your function will cause a compiler error. Passing a variable with a negative value is possible only with explicit type casting which makes you think about if the value you have really suitable for that function at the call site. It's not the function's responsibility to validate for negative arguments anymore. Assume that a function needs to return trending tags in your microblogging web site up to only specified number of tags. It receives a number of items to retrieve rows of posts as in listing 4.10.

In listing 4.10, we have a `GetTrendingTags` function which returns items by taking the number of items into account. Notice that the input value is a byte instead of int because we don't have any use case more than 255 items in trending tag list. That actually immediately eliminates the cases where an input value can be negative or too large. We don't even need to validate the input anymore. One fewer test case and a much better range of input values, which reduces the area for bugs immediately.

Listing 4.10 Receive posts only belonging a certain page

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace Posts {
  public class Tag {
    public Guid Id { get; set; }
    public string Title { get; set; }
  }

  public class PostService {
    public const int MaxPageSize = 100;
    private readonly IPostRepository db;

    public PostService(IPostRepository db) {
      this.db = db;
    }

    public IList<Tag> GetTrendingTags(byte numberOfItems) {    #A
      return db.GetTrendingTagTable()
        .Take(numberOfItems)    #B
        .ToList();
    }
  }
}
```

#A We chose byte instead of int.
#B A byte or a ushort can be passed as safely as int too.

There are two things happening here. First, we chose a smaller data type for our use case. We don't intend to support billions of rows in a trending tag box. We don't even know what that would look like. We narrowed down our input space. Second, we chose byte, an unsigned type, that's impossible to become negative. That way, we avoided a possible test case and a potential problem that might cause an exception. LINQ's Take function doesn't throw an exception with a List, but it can when it gets translated to a query for a database like Microsoft SQL Server. By changing the type, we avoided those cases, and we don't need to write tests for them.

Note that .NET uses `int` as the de facto standard type for many operations like indexing and counting. Opting for a different type might need you to cast and convert values into `int` if you happen to interact with standard .NET components. You need to make sure that you're not digging yourself into a hole by being pedantic. Your quality of life and the enjoyment you get from writing code is more important than a certain one-off case you're trying to avoid. For example, if you need more than 255 items in the future, you'll have to replace all references to bytes with `short`s or `int`s which can be a time-consuming task. You need to make sure that you are saving yourself from writing tests for a worthy cause. You might even find writing additional tests more favorable in many cases rather than dealing with different types. In the end, it's only your comfort and your time that matters despite how powerful it is to use types for hinting at valid value ranges.

4.7.3 Eliminate valid value checks

There are times we use values to signify an operation in a function. A common example is `fopen` function in C programming language. It takes a second string parameter that symbolizes the open mode, which can mean "open for reading", "open for appending", "open for writing", and so forth.

.NET team, decades after C of course, has made a better decision and created separate functions for them. You have `File.Create`, `File.OpenRead`, `File.OpenWrite` methods separately, avoiding the need for an extra parameter and the need for parsing that parameter. It's impossible to pass along the wrong parameter. It's impossible for functions to have bugs in parameter parsing because there is no parameter.

It's common to use such values to signify a type of operation. You should consider separating them into distinct functions instead, which can both convey the intent better, and reduce your test surface.

One of the common ways in C# is to use Boolean parameters to change the logic of the running function. An example would be to have a sorting option in our trending tags retrieval function as in listing 4.11. Assume that we need trending tags in our tag management page too, and it's better to show them sorted by title there. In contradiction with laws of thermodynamics, developers tend to constantly lose entropy. They always try to make the change with the least entropy, without thinking that how much burden it will be in the future. The first instinct of a developer can be to add a Boolean parameter and be done with it.

Listing 4.11 Boolean parameters

```
public IList<Tag> GetTrendingTags(byte numberOfItems,
  bool sortByTitle) {    #A
```

```
  var query = db.GetTrendingTagTable();
  if (sortByTitle) {    #B
    query = query.OrderBy(p => p.Title);
  }
  return query.Take(numberOfItems).ToList();
}
```

#A Newly added parameter
#B Newly introduced conditional

The problem is, if we keep adding Booleans like this, it can get really complicated because of the combinations of those variables. Let's say another feature required trending tags from yesterday. We add that in too with other parameters in listing 4.12. Now, our function needs to support combinations of `sortByTitle` and `yesterdaysTags` too.

Listing 4.12 More Boolean parameters

```
public IList<Tag> GetTrendingTags(byte numberOfItems,
  bool sortByTitle, bool yesterdaysTags) {    #A
  var query = yesterdaysTags    #B
    ? db.GetTrendingTagTable()    #B
    : db.GetYesterdaysTrendingTagTable();    #B
  if (sortByTitle) {    #B
    query = query.OrderBy(p => p.Title);
  }
  return query.Take(numberOfItems).ToList();
}
```

#A More parameters!
#B More conditionals!

There is an ongoing trend here. Our function's complexity increases with every Boolean parameter. Although we have three different use cases, we have four flavors of the function. With every added Boolean parameter, we are creating fictional versions of the function that no one will use, yet someone might someday and get into a bind. A better approach to have a separate function for each client, as in listing 4.13.

Listing 4.13 Separate functions

```
public IList<Tag> GetTrendingTags(byte numberOfItems) {    #A
  return db.GetTrendingTagTable()
    .Take(numberOfItems)
    .ToList();
}

public IList<Tag> GetTrendingTagsByTitle(    #A
  byte numberOfItems) {
  return db.GetTrendingTagTable()
    .OrderBy(p => p.Title)
    .Take(numberOfItems)
    .ToList();
}

public IList<Tag> GetYesterdaysTrendingTags(byte numberOfItems) {    #A
  return db.GetYesterdaysTrendingTagTable()
    .Take(numberOfItems)
```

```
    .ToList();
}
```

#A We separate functionality by function names instead of parameters.

We now have one less test case. You get much better readability and slightly increased performance for free as a bonus. The gains are miniscule of course, and unnoticeable for a single function, but at points where the code needs to scale, they can make a difference without you even knowing. The savings will increase exponentially when you avoid trying to pass state in parameters and leverage functions as much as possible. You might still be irked by repetitive code, which can easily be refactored into common functions as in listing 4.14.

Listing 4.14 Separate functions with common logic refactored out

```
private IList<Tag> toListTrimmed(byte numberOfItems,    #A
  IQueryable<Tag> query) {    #A
  return query.Take(numberOfItems).ToList();    #A
}    #A

public IList<Tag> GetTrendingTags(byte numberOfItems) {
  return toListTrimmed(numberOfItems, db.GetTrendingTagTable());
}

public IList<Tag> GetTrendingTagsByTitle(byte numberOfItems) {
  return toListTrimmed(numberOfItems, db.GetTrendingTagTable()
    .OrderBy(p => p.Title));
}

public IList<Tag> GetYesterdaysTrendingTags(byte numberOfItems) {
  return toListTrimmed(numberOfItems,
    db.GetYesterdaysTrendingTagTable());
}
```

#A Common functionality

Our savings are not impressive here, but such refactors can make greater differences in other cases. The important takeaway is to use refactoring to avoid code repetition and combinatorial hell.

The same technique can be used with enum parameters that are used to dictate a certain operation to a function. Use separate functions, and you can even use function composition, instead of passing along shopping list of parameters.

4.8 Naming tests

There is a lot in a name. That's why it's important to have good coding conventions in both production and test code, although they shouldn't necessarily overlap. Tests with good coverage can serve as specifications if they're named correctly. From the name of a test, you should be able to tell:

- Name of the function being tested
- Input and initial state
- Expected behavior
- Whom to blame

Kidding about the last one of course. Remember? You already greenlit that code in the code review. You have no right to blame someone else anymore. If anything, you both should be blamed. I commonly use an "`A_B_C`" format to name tests which is quite different than what you're used to name your regular functions. We used a simpler naming scheme in previous examples because we were able to use the TestCase attribute to describe the initial state of the test. I use an additional "`_ReturnsExpectedValues`" but you can simply suffix the function name with `Test`. It's better if you don't use the function name alone because that might confuse you when it appears in code completion lists. Similarly, if the function doesn't take any input or doesn't depend on any initial state, you can skip the part describing that. The purpose here is to make you spend less time dealing with tests, not to put you through a military drill about naming rules.

Say, your boss came and asked you to write a new validation rules for registration form, where you need to make sure registration code returns failure if user hasn't accepted policy terms. A name for such a test would be `Register_LicenseNotAccepted_ShouldReturnFailure` as in figure 4.6.

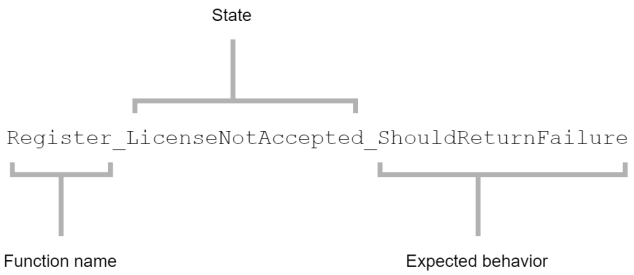


Figure 4.6 Components of a test name

That's not the only naming convention possible. There are people who prefer creating inner classes for each function to be tested and name tests with only state and expected behavior, but I find that unnecessarily cumbersome. It's important that you pick the convention that works for you the best.

4.9 Summary

- It's possible to overcome the disdain for writing tests by not writing many of them in the first place.
- Test-Driven Development and similar paradigms can make you hate writing tests even more. Seek to write tests that spark joy.
- The effort to write tests can be significantly shortened by test frameworks, especially with parameterized, data-driven tests.
- The number of tests cases can be reduced significantly by properly analyzing boundary values of a function input.
- Proper use of types can let you get away from writing many unnecessary tests.

- Tests don't just ensure the quality of the code. They can help you improve your own development skills and throughput too.
- Testing in production can be acceptable as long as your resumé is up to date.

5

Rewarding refactoring

This chapter covers

- Getting comfortable with refactoring
- Incremental refactoring on large changes
- Using tests to make code changes faster
- Dependency injection

We discussed in chapter 3 how resistance to change caused the downfall of the French royal family and software developers. Refactoring is the art of changing the structure of the code. According to Martin Fowler,[1] Leo Brodie coined the term in his book called *Thinking Forth* back in 1984. That makes the term as old as *Back to the Future* and *Karate Kid*, my favorite movies when I was a kid.

Writing great code is usually only the half of being an efficient developer. The other half is to be agile in transforming code. In an ideal world, we should be writing and changing code at the speed of thought. Hitting keys, nailing the syntax, memorizing keywords, changing the coffee filter, they are all obstacles between your ideas and the product. Since it'll probably take a while until we get AI to do programming work for us, it's a good idea to polish our refactoring skills.

IDEs are instrumental in refactoring. You can rename a class with a single keystroke (F2 on Visual Studio for Windows) and rename all the references to it instantly. You can even access most of the refactoring options with a single keypress. I strongly recommend familiarizing yourself with keyboard shortcuts for the features that you frequently use on your favorite editor. The time savings will accumulate and you'll look cool in front of your colleagues.

[1] Etymology of Refactoring, Martin Fowler, https://martinfowler.com/bliki/EtymologyOfRefactoring.html

5.1 Why do we refactor?

Change is inevitable, code change is doubly so. Refactoring serves other purposes than simply changing the code though. It lets you:

- Reduce repetition and increase code reuse. You can move a class that can be reused by other components to a common location so other components can start using it too. Similarly, you can extract methods from the code and make them available for reuse.
- Bring your mental model and the code closer. Names are important. Some names may not be as easily understandable as others. Renaming things are part of refactoring process and can help you achieve a better design that matches your mental model better.
- Make the code easier to understand and maintain. You can reduce code complexity by splitting long functions into smaller, more maintainable ones. Similarly, a model can be easier to understand if complex data types are grouped in smaller, atomic parts.
- Prevent certain class of bugs from appearing. Certain refactoring operations, like changing a class to a struct can prevent bugs related to nullability as we've seen in chapter two. Similarly, enabling nullable references on a project and changing data types to non-nullable ones can prevent bugs that are basically refactoring operations.
- Prepare for a significant architectural change. Big changes can be performed faster if you prepare the code for the change beforehand. We will see how that can happen in the next section.
- Get rid of the rigid parts of the code. Through dependency injection, you can remove dependencies and have a loosely coupled design.

Most of the time we developers see refactoring as a mundane task, part of our programming work. Refactoring is also a separate external work that you do even if you're not writing a single line of code. You can even do it for the purpose of reading the code because it's hard to grasp. Richard Feynman once said, "If you want to truly learn a subject, write a book about it." In a similar vein, you can truly learn about a piece of code when refactoring it.

Simple refactoring operations need no guidance at all. You want to rename a class? Go ahead. Extract methods or interfaces? They are no brainers. They are even on the right-click menu for Visual Studio, which can also be brought up with `Ctrl` + `.` on Windows. Most of the time, they don't affect code reliability at all. However, when it comes to a significant architectural change in the code base, that's when you might need some advice.

5.2 Architectural changes

It's almost never a good idea to perform a large architectural change in one shot. That's not because it's technically hard, but mostly because large changes bring large number of bugs and *integration problems* due to the long and broad nature of the work. By integration problems, I mean that if you're working on a large change, you need to work on it for a long time without being able to integrate changes from other developers. That puts you in a bind. Do you wait until you're done with your work and manually apply every change that's been

made on the code in that timeframe and fix all the conflicts yourself, or do you tell your team members to stop working until you finish your changes? Note that this is mostly a problem when refactoring. You don't have the same problem when developing a new feature because the possibility of conflicting with other developers is way less due to the feature itself not existing in the first place. That's why incremental approach is better.

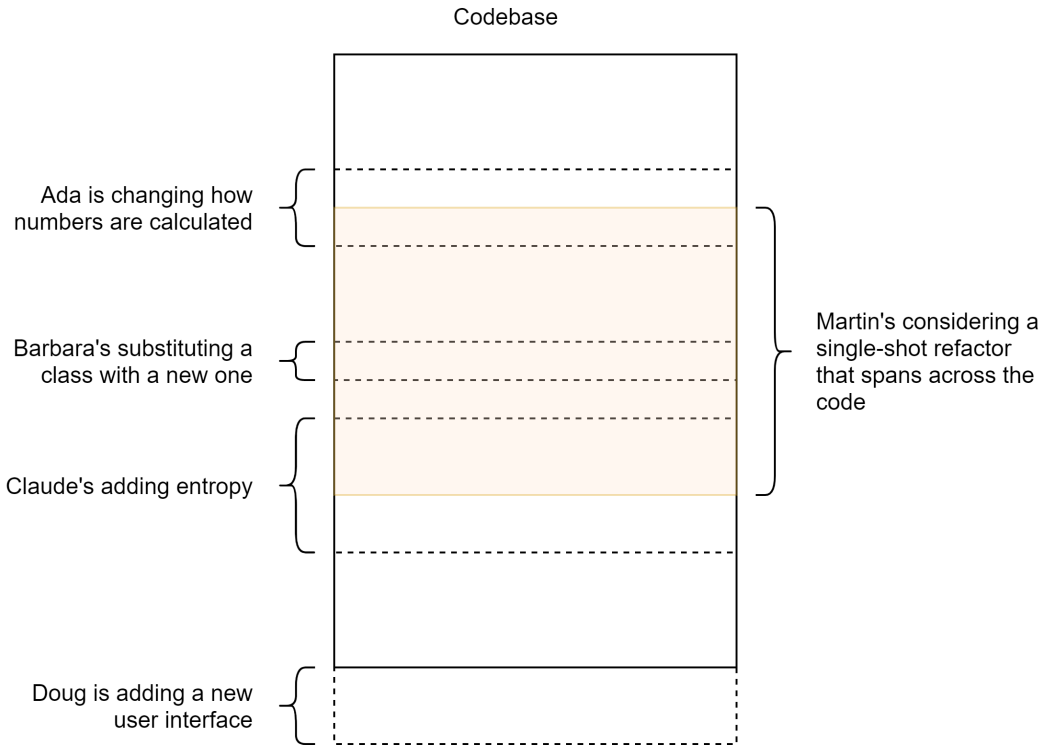


Figure 5.1 Why one-shot large refactors are a bad idea

In order to create a roadmap you need to have a destination and know where you are first. How do you want the end result to look like? It may not be possible to imagine everything altogether as large software is really hard to wrap your head around. You can have certain list of requirements instead.

Let's work on a migration example. Microsoft has two flavors of .NET in the wild, the first one is .NET Framework, which is decades old, and the second one is just called .NET (previously known as .NET Core), which was recently released. Both are still supported by Microsoft at the of writing this book, but it's obvious that Microsoft wants to move forward with .NET and drop .NET Framework at some point. It's very likely that you'll encounter work that needs migration from .NET Framework to .NET.

.NET Framework is dead, long live .NET!

The name .NET meant many things back in the 90s, when internet was getting big. There was even a magazine called *.net,* which was about Internet and pretty much worked as a slower version of Google. Browsing the web was commonly called "surfing the net", "traveling the information superhighway", "connecting to the cyberspace" or any other combination of a misleading metaphoric verb with a made-up noun.

.NET Framework was the original software ecosystem created to make developer's lives easier back in the late 90's. It came with the runtime, standard libraries, compilers for C#, Visual Basic, and later F# languages. The Java equivalent of .NET Framework would be JDK (Java Development Kit) having the Java runtime, a Java language compiler, Java Virtual Machine and probably some other things starting with Java.

There came other .NET flavors over time that were not directly compatible with .NET Framework such as **.NET Compact Framework** and **Mono**. In order to allow code sharing between different frameworks, Microsoft created a common API specification that defined a common subset of the .NET functionality that was called **.NET Standard**. Java doesn't suffer from a similar problem because Oracle successfully killed all the incompatible alternatives with an army of lawyers.

Microsoft later created a new generation of .NET Framework that was cross-platform. It was initially called .NET Core, and recently renamed solely to **.NET** itself starting with .NET 5. It's not directly compatible with .NET Framework but it can interoperate using a common .NET Standard subset specification.

.NET Framework is still plugged to the life support but we probably won't be seeing it around in five years. I strongly recommend anyone using .NET to start out with .NET, rather than .NET Framework, and that's why I picked up an example based on this migration scenario.

In addition to your destination, you need to know where you are too. That reminds me of the story of a CEO in a helicopter. A CEO was getting a ride in a helicopter and they got lost in the fog. They noticed the silhouette of a building and saw someone at the balcony. The CEO said, "I've got an idea, get us closer to that person." They got closer carefully next to the person and the CEO shouted, "Hey! Do you know where we are?" The person replied, "Yes, you're in a helicopter!" The CEO said, "Okay, then we must be at the college campus and that must be the engineering building!" The person at the balcony was surprised, "How did you figure it out?" The CEO replied, "The answer you gave us was technically correct, but completely useless!" The person shouted, "Then you must be a CEO!" The CEO was surprised now and asked, "How did you know?" The person answered, "You got lost, have no idea where you are, or where you're going, and it's still my fault!"

I can't help myself from imagining the CEO jumping to the balcony from the helicopter and a Matrix-like fight sequence breaking out between the runaway engineer and the CEO wielding double katanas simply because the pilot didn't know how to read a GPS instead of practicing precision approach maneuver to balconies.

Consider that we have our anonymous microblogging web site called Blabber written in .NET Framework and ASP.NET and we'd like to move it to the new .NET platform and ASP.NET Core. Unfortunately, ASP.NET Core and ASP.NET are not binary compatible, and only a little source compatible. The code for the platform is included in the source code of the book. I won't be listing the full code here as ASP.NET template comes with quite boilerplate, but I'll sketch out the architectural details that will guide us in creating a refactoring roadmap. You don't need to know about the architecture of ASP.NET or how web apps work

in general to understand our refactoring process, as it's not directly relevant to the refactoring work.

5.2.1 Identify the components

The best way to work with a large refactor is to split your code into semantically distinct components. Let's split our code into several parts for the sole purpose of a refactor. Our project is an ASP.NET *MVC* application with some model classes and controllers we added. We can have an approximate list of components as in figure 5.1. It doesn't need to be accurate; it can be what you come up with initially as it will change.

What's MVC?

The entire history of computer science can be summarized as fighting with entropy, also known as spaghetti by the believers of Flying Spaghetti Monster, the creator of all entropy. MVC is an idea that splitting a code into three parts to avoid too much interdependency, aka spaghetti code: the part that decides how the user interface will look, the part that models your business logic, and the part that coordinates the two. They are respectively called View, Model, and Controller. There are many other similar attempts on splitting application code into logically separate parts like MVVM (Model, View, ViewModel), or MVP (Model, View, Presentation), but the idea behind them are pretty much the same: decoupling distinct concerns from each other.

Such compartmentalization can help you in writing code, creating tests, and refactoring, as the dependencies between those layers become more manageable. But as stated eloquently in No Free Lunch Theorem by scientists David Wolpert and William Macready, there is no free lunch. You usually have to write slightly more code, work with a greater number of files, have more subdirectories, and more moments that you curse at the screen to get the benefits of MVC, but in the big picture, you become faster and more efficient.

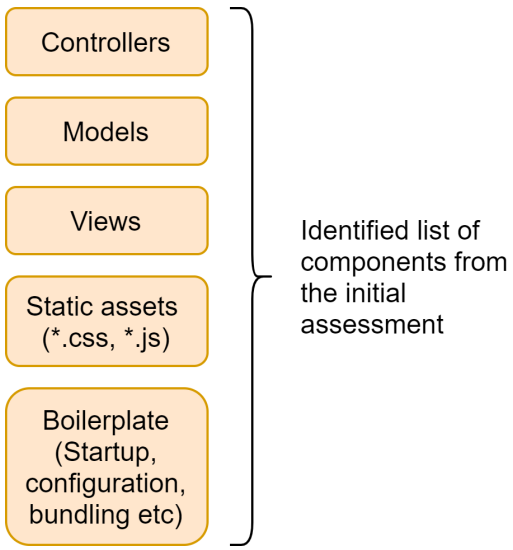


Figure 5.2 Our initial assessment of components

After you have the list of components down, start assessing how many of the components you can transfer directly to your destination, in our example .NET 5. Note that "destination" means *the destination state* that symbolizes the end result. Can the components be manipulated into the destination state without breaking anything? Do you think they will need some work? Assess this per component, and we will use this guesswork to prioritize. You don't really need to accurately know this, but a guesswork is enough at this moment. You can have a work estimation table like in table 5.1.

5.2.2 Estimate the work and the risk

How will you know how much work will be needed? You must have a vague idea about how both frameworks work to do that. It's important that you know your destination before you start walking towards it. You can be wrong about some of these guesses, and that's okay, but the primary reason to have this practice is to prioritize work towards reducing your workload without breaking anything as long as possible.

For example, I know controllers and views require minimal effort because I know that their syntax hasn't changed much between frameworks. Yet, I anticipate a little work with the syntax of some Html helpers or controller constructs, but there is a great chance that I should be moving them without any issues. Similarly, I know static assets are moved under `wwwroot/` folder in ASP.NET Core, which requires only a little work, but definitely not directly transferable. I also know that startup and configuration code has completely been overhauled in ASP.NET Core, which means I'll have to write them from scratch.

I assume all other developers will be working on features, so I expect their work would involve work under controllers, views, and models. I don't expect existing models to change as frequent as the business logic or how the features look, so I assign Models a medium risk while Controllers and Views a higher risk probability. Remember, other developers are working on the code while you're working on your refactoring. So, you must find a way to integrate your work to their workflow as early as possible without breaking their flow. The most feasible component for that looks like Models in table 5.1 despite that the possibility of high conflict, it requires minimal change, so resolving any conflicts should be straightforward.

Table 5.1 Assessing relative cost and risks of manipulating components

Component	Changes needed	Risk of conflicting with another developer
Controllers	Minimal	High
Models	None	Medium
Views	Minimal	High
Static assets	Some	Low
Boilerplate	Rewrite	Low

It needs no change to be refactored. How do you make existing code and the new code with the same component at the same time? You move it into a separate project. We discussed this in chapter three when talking about breaking dependencies to make a project structure more open to change.

5.2.3 The prestige

Refactoring without disrupting your colleagues is pretty much like changing the tire of a car while driving on the highway. It resembles an illusion act that makes the old architecture disappear and gets it replaced with the new one without anyone noticing. Your greatest tool when doing that would be extracting code into shareable parts as shown in figure 5.3.

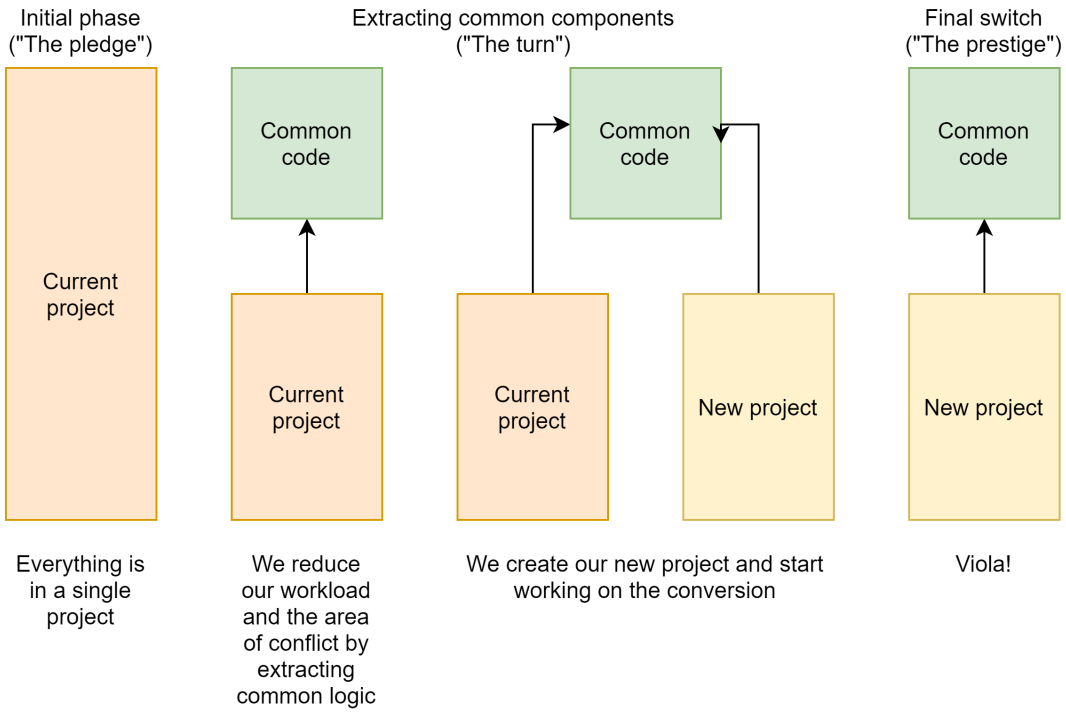


Figure 5.3 The illusion of refactoring without any developer noticing

Of course, it's impossible for developers not to notice the new project in the repository, but as long as you communicate with them the changes you're trying to implement beforehand, and it's straightforward for them to adapt, you should have no problems implementing your changes as the project goes forward.

You create a separate project, as in our example, Blabber.Models, move your Models classes to that project, and then add a reference to that project from the web project. Your code will keep running as it was before, but the new code will need to be added in the

Blabber.Models project rather than Blabber, and your colleagues need to be aware of this change. You can then create your new project, and reference Blabber.Models from that too. Our roadmap resembles figure 5.4.

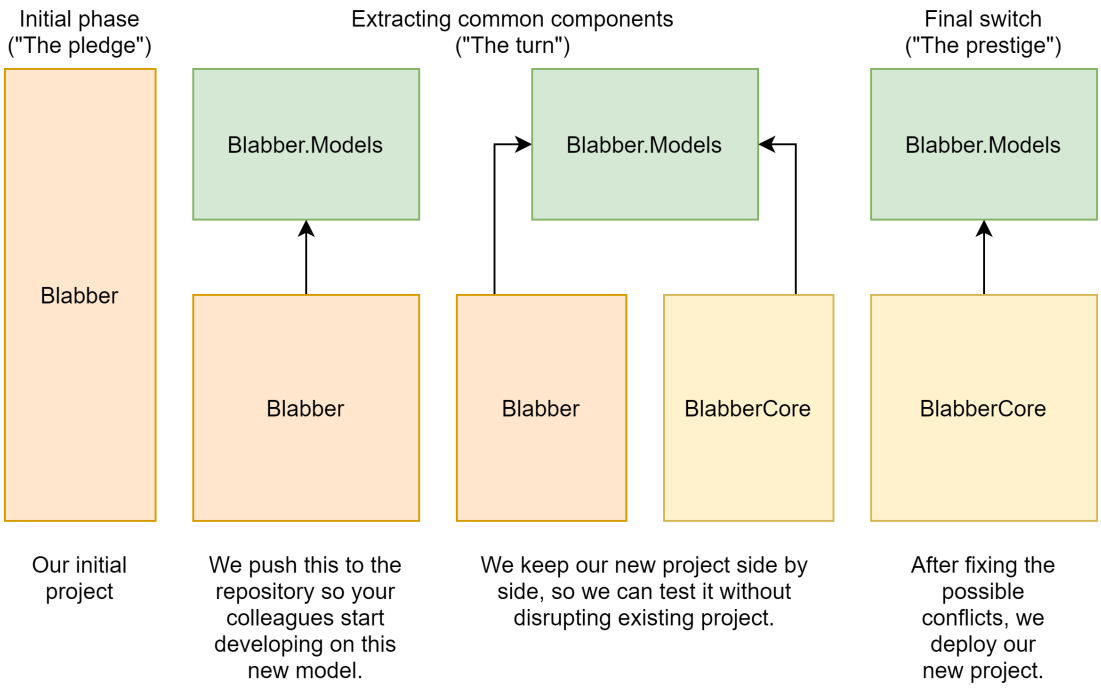


Figure 5.4 Our project's refactoring roadmap

The reason we are going through this is to reduce our work while staying as current as possible with the main branch. This method also lets you perform your refactoring work in a timeframe distributed over a longer timeline while squeezing other, more urgent work in your schedule. It pretty much resembles checkpoint systems in video games where you can start at the same Valkyrie fight the hundredth time in God of War instead of the beginning of the entire game. Whatever you can integrate into main branch without breaking the build becomes a last known good spot where you don't have to repeat. Planning your work with multiple integration steps is the most feasible way to perform a large refactor.

5.2.4 Refactor to make refactoring easier

When moving code across projects, you'll encounter strong dependencies that cannot be easily moved out. In our example, some of the code might depend on web components and moving them to our shared project would be meaningless as our new project, BlabberCore, wouldn't work with the old web components.

In such cases, composition comes to our rescue. We can extract an interface that our main project can provide and pass it to our implementation instead of the actual dependency.

Our current implementation of Blabber uses an in-memory storage for the content posted on the web site. That means, whenever you restart the web site, all the platform content is lost. That makes sense for a post-modern art project, but users expect at least a level of persistence. Let's assume we'd like to use either Entity Framework or Entity Framework Core, based on the framework we're using, but we still would like to share the common DB access code among two projects while our migration is keep going, so the actual work needed for the final stretch for migration will be far less.

DEPENDENCY INJECTION

You can abstract away dependencies that you don't want to deal by creating an interface for it and receiving in its implementation in a constructor. That technique is called *dependency injection*. Do not confuse dependency injection with *dependency inversion* which is an overhyped principle basically states "depend on abstractions" but sounds less profound when put like that.

"Dependency injection" (DI) is also a slightly misleading term. It implies interference or disturbance. There is nothing going on like that. Perhaps, it should have been called dependency reception because that's what it's about: receiving your dependencies during initialization such as in your constructor. DI is also called IoC (Inversion of Control), which sometimes is even more confusing. A typical dependency injection is a design change as shown in figure 5.5. Without dependency injection, you instantiate your dependent classes in your code. With dependency injection, you receive the classes you depend on in a constructor.

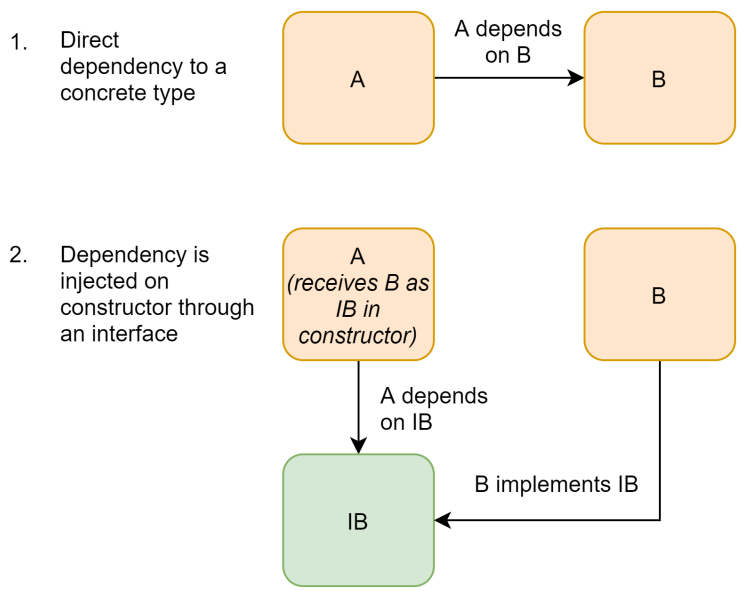


Figure 5.5 How dependency injection changes the design of a class

Let's go over how it's performed in a simple and abstract code so you can focus on the actual changes happening. In this example, you can also see how C# 9.0 top-level program code looks, without a `Main` method or a `Program` class per se. You can actually type the code in listing 5.1 in a ".cs" file under a project folder and run it right away, without any extra code. Note that how class A initializes an instance of a class B every time the method X is called.

Listing 5.1 Code that uses direct dependency

```
using System;

var a = new A();    #A
a.X();

public class A {
  public void X() {
    Console.WriteLine("X got called");
    var b = new B();    #B
    b.Y();
  }
}

public class B {
  public void Y() {
    Console.WriteLine("Y got called");
  }
}
```

#A Main code creates an instance of A here.
#B Class A creates the instance of class B.

When you apply dependency injection, your code gets its instance of class B in its constructor and through an interface so, you have zero coupling between classes A and B. You can see how it shapes up in listing 5.2. There is a difference in conventions though. Because we moved initialization code of class B to constructor, it always uses the same instance of B, instead of creating new one, which how it used to work in listing 5.1. That's actually good thing as it reduces the load on GC, but it can create unexpected behavior if the state of class changes over time. You might be breaking behavior. That's why having test coverage is a good idea in the first place.

What we've accomplished with the code in listing 5.2 is, we now can completely remove the code for B, and move it to an entirely different project without breaking the code in A as long as we keep the interface we've created (`IB`). More importantly, we can move everything B needs to have along with it too. It gets us quite a freedom to move the code around.

Listing 5.2 Code with dependency injection

```
using System;

var b = new B();    #A
var a = new A(b);    #B
a.X();
```

```
public interface IB {
  void Y();
}

public class A {
  private readonly IB b;    #C
  public A(IB b) {
    this.b = b;
  }
  public void X() {
    Console.WriteLine("X got called");
    b.Y();    #D
  }
}

public class B : IB {
  public void Y() {
    Console.WriteLine("Y got called");
  }
}
```

#A Caller initializes class B
#B Passes it to class A as a parameter
#C Instance of B is kept here.
#D Common instance of B is called.

Now, let's apply this technique to our example in Blabber and change our code to use database storage instead of memory, so our content survives restarts. In our example, instead of depending on a specific implementation of a DB engine, in this case Entity Framework and EF Core, we can instead receive an interface we devise that provides required functionality to our component. This lets two projects with different technologies to use the same code base, despite that the common code depends on the specific DB functionality. In order to achieve that, we create a common interface, `IBlabDb` that points to the database functionality and use it in our common code. Our two different implementations share the same code yet, it lets the common code to use different DB access technologies. Our implementation will look like in figure 5.6.

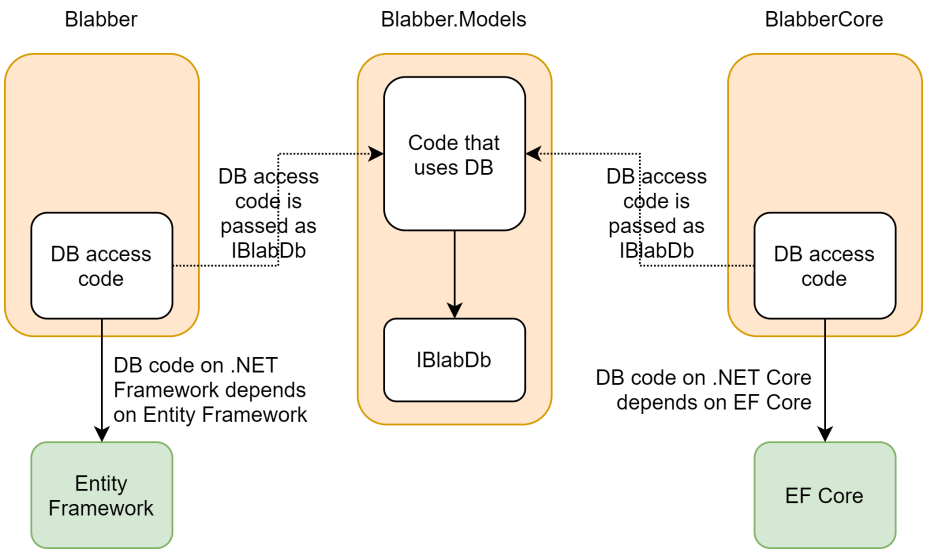


Figure 5.6 Using different technologies in common code with dependency injection

In order to implement that we first change our implementation of `BlabStorage` in `Blabber.Models` that we refactored, so it defers work to an interface instead. The in-memory implementation of `BlabStorage` class looked like as in listing 5.3. It keeps a static instance of a list that is shared between all requests, so it uses locking to ensure that things don't get inconsistent. We don't care about the consistency of our Items property as we only add items to this list and items are never removed. Otherwise, it would have been a problem. Note that we use `Insert` instead of `Add` in `Add()` method because it lets us to keep posts in descending order by their creation date without resorting to any sorting.

Listing 5.3 Initial in-memory version of BlabStorage

```
using System.Collections.Generic;

namespace Blabber.Models {
    public class BlabStorage {
        public IList<Blab> items = new List<Blab>();    #A
        public IEnumerable<Blab> Items => items;
        public object lockObject = new object();    #B
        public static readonly BlabStorage Default = new BlabStorage();    #C

        public BlabStorage() {
        }

        public void Add(Blab blab) {
            lock (lockObject) {
                items.Insert(0, blab);    #D
            }
        }
```

```
    }
}
```

#A Creating an empty list by default.
#B We're using lock object to allow concurrency.
#C Default singleton instance that's used everywhere
#D Most recent item goes to top.

When we implement dependency injection, we remove everything related to in-memory lists and we use an abstract interface for anything related to database instead. The new version looks like as in listing 5.4. You can see how we remove anything related to the logic of data storage and our `BlabStorage` class actually became an abstraction by itself. It looks like `BlabStorage` itself doesn't do anything extra but as we add more complicated tasks, we're able to share some logic between our two projects. So, for the sake of the example, this is okay.

We keep the dependency in a private and read-only field called `db`. It's good habit to mark fields with the `readonly` keyword if they won't change after the object is created, so the compiler can catch if you or one of your colleagues accidentally try to modify it outside the constructor.

Listing 5.4 BlabStorage with dependency injection

```
using System.Collections.Generic;

namespace Blabber.Models {
  public interface IBlabDb {    #A
    IEnumerable<Blab> GetAllBlabs();
    void AddBlab(Blab blab);
  }

  public class BlabStorage {
    private readonly IBlabDb db;

    public BlabStorage(IBlabDb db) {    #B
      this.db = db;
    }

    public IEnumerable<Blab> GetAllBlabs() {
      return db.GetAllBlabs();    #C
    }

    public void Add(Blab blab) {
      db.AddBlab(blab);    #C
    }
  }
}
```

#A The interface that abstracts away the dependency
#B Receiving the dependency in the constructor
#C Deferring work to the component that does the actual work

Our actual implementation is called `BlabDb`, which implements the interface `IBlabDb` and resides in the project BlabberCore, rather than Blabber.Models. It uses a SQLite (pronounced "sequel-light") database for practical purposes as it requires no setup of third-party software

so you can start running it right away. SQLite is God's latest gift to the world before he gave up on the humankind. Just kidding, it was Richard Kipp who created it, before he gave up on the humankind. BlabberCore project implements it in EF Core as in listing 5.5.

You may not be familiar with EF Core, Entity Framework, or ORMs (Object-relational Mapping) in general, but that's okay; you don't have to. It's pretty much straightforward as you can see. The `AddBlab` method just creates a new database record in memory, creates a pending insertion to the Blabs table and calls `SaveChanges` to write changes to the database. Similarly, the `GetAllBlabs` method simply gets all the records from the database, ordered by date in descending order. Notice how we need to convert our dates to UTC to make sure time zone information isn't lost, because SQLite doesn't support `DateTimeOffset` types. Regardless of how many best practices you learn, you'll always encounter cases that it just won't work. In those cases, you'll have to improvise, adapt, and overcome as in this example.

Listing 5.5 EF Core version of BlabDb

```
using Blabber.Models;
using System;
using System.Collections.Generic;
using System.Linq;

namespace Blabber.DB {
  public class BlabDb : IBlabDb {
    private readonly BlabberContext db;    #A

    public BlabDb(BlabberContext db) {   #B
      this.db = db;
    }

    public void AddBlab(Blab blab) {
      db.Blabs.Add(new BlabEntity() {
        Content = blab.Content,
        CreatedOn = blab.CreatedOn.UtcDateTime,    #C
      });
      db.SaveChanges();
    }

    public IEnumerable<Blab> GetAllBlabs() {
      return db.Blabs
        .OrderByDescending(b => b.CreatedOn)
        .Select(b => new Blab(b.Content,
          new DateTimeOffset(b.CreatedOn, TimeSpan.Zero)))    #D
        .ToList();
    }
  }
}
```

#A EF Core DB context
#B Receiving context through dependency injection
#C Converting our DateTimeOffset to DB-compatible type
#D Converting DB-time to DateTimeOffset

We managed to introduce a database storage backend to our project during our refactoring without disrupting the development workflow. We used dependency injection to avoid direct

dependencies. More importantly, our content is now persisted across sessions, and restarts as you can see in figure 5.7.

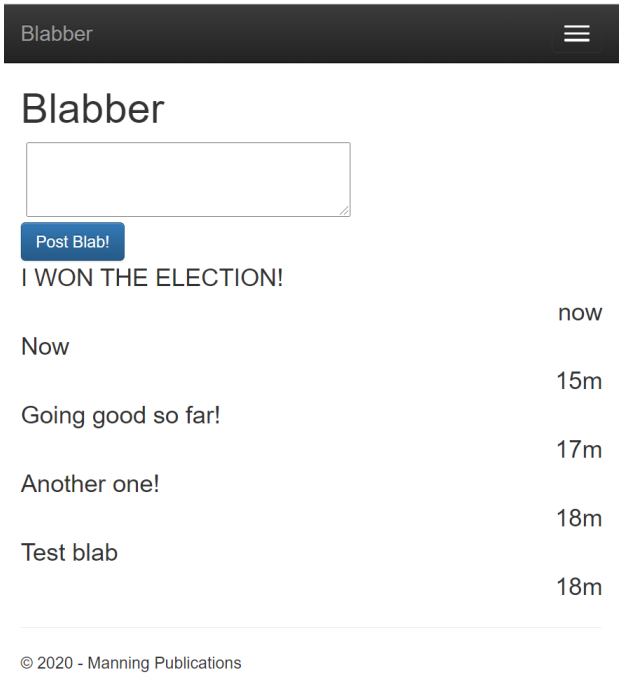


Figure 5.7 Screenshot of Blabber running on SQLite database

5.2.5 The final stretch

You can extract as many components as there can be shared between the old and the new project, but eventually, you'll hit a chunk of code that is not possible to share between two web projects. For example, our controller code doesn't need to change between ASP.NET and ASP.NET Core, because the syntax is the same, but it's impossible to share that piece of code between the two because they use entirely different types. ASP.NET MVC controllers are derived from `System.Web.Mvc.Controller` while ASP.NET Core controllers are derived from `Microsoft.AspNetCore.Mvc.Controller`. There are theoretical solutions to this, like abstracting away the controller implementation behind an interface and write custom classes that use that interface, instead of being direct descendants of `Controller` class, but that's just too much work. When you come up with a supposedly elegant solution to a problem, you should always ask yourself "is it worth it?". Elegance in engineering must always take the cost into account.

That means, at some point, you'll have to take the risk of conflicting with other developers and transfer the code to the new code base. I call that the final stretch, which will take a shorter time thanks to your previous preparatory work on refactoring. Because of your

work, the future refactor operations will take shorter because you'll end up with a compartmentalized design at the end of the process. It's a good investment.

In our example, Models component consists of an unusually small part of our project, therefore, makes our savings negligible. However, it's expected for large projects to have significant amount of shareable code, which might reduce your work factor considerably.

In the final stretch, you need to transfer all the code and the assets to your new project, then make everything work. I added a separate project to the code examples called BlabberCore that contains the new .NET code so you can see how some constructs translate to .NET Core.

5.3 Reliable refactoring

Your IDE tries really hard so you don't break the code simply by randomly choosing menu options. If you manually edit a name, any other code that references the name will break. If you use the rename function of your IDE, all references to the name will be renamed as well. That is not always a guarantee though. There are many ways you can refer to a name without compiler knowing. For example, it's possible to instantiate a class using a string too. In our example microblogging code, Blabber, we refer to every piece of content as blabs, and we have a class that defines a content called Blab as in listing 5.1.

Listing 5.6 Class representing a content

```
using System;

namespace Blabber
{
    public class Blab
    {
        public string Content { get; private set; }
        public DateTimeOffset CreatedOn { get; private set; }

        public Blab(string content, DateTimeOffset createdOn)    #A
        {
            if (string.IsNullOrWhiteSpace(content))
            {
                throw new ArgumentException(nameof(content));
            }
            Content = content;
            CreatedOn = createdOn;
        }
    }
}
```

#A Constructor ensures there are no invalid blabs.

We normally instantiate classes using the `new` operator, but it's also possible to instantiate the `Blab` class using reflection for certain purposes such as when you don't know what class you're creating during compile time:

```
var blab = Activator.CreateInstance("Blabber.Models",
    "Blabber", "test content", DateTimeOffset.Now);
```

Whenever we refer to a name in a string, we risk breaking the code after a rename as IDE cannot track the contents of strings. Hopefully, that'll stop being a problem when we start doing code reviews with our AI overlords. I don't know why in that fictional future, it's still us who's doing the work, and AI just grades our work though? Weren't they supposed to take over our jobs? It turns out they are much more intelligent than we give them credit for.

Until the AI takeover of the world, your IDE cannot guarantee a perfectly reliable refactoring. Yes, you have some wiggle room, like using constructs like `nameof()` to reference types instead of hard coding them into strings as we've seen in previous chapter, but that helps you only marginally.

The secret to the reliable refactoring is testing. If you can make sure that your code has good test coverage, you can have much more freedom in changing it. Therefore, it's usually a wise idea to start a long-term refactoring work with creating missing tests for the relevant piece of code first. If we take our architecture change example in the previous chapters, a more realistic roadmap would involve adding missing tests to the whole architecture. We skipped that step in our example because our code base was extremely small and trivial to test manually (e.g. run the app, post a blab, see if it appears).

You can see a modified version of our roadmap in figure 5.8 which includes the phase of adding tests to our project so it can be refactored reliably.

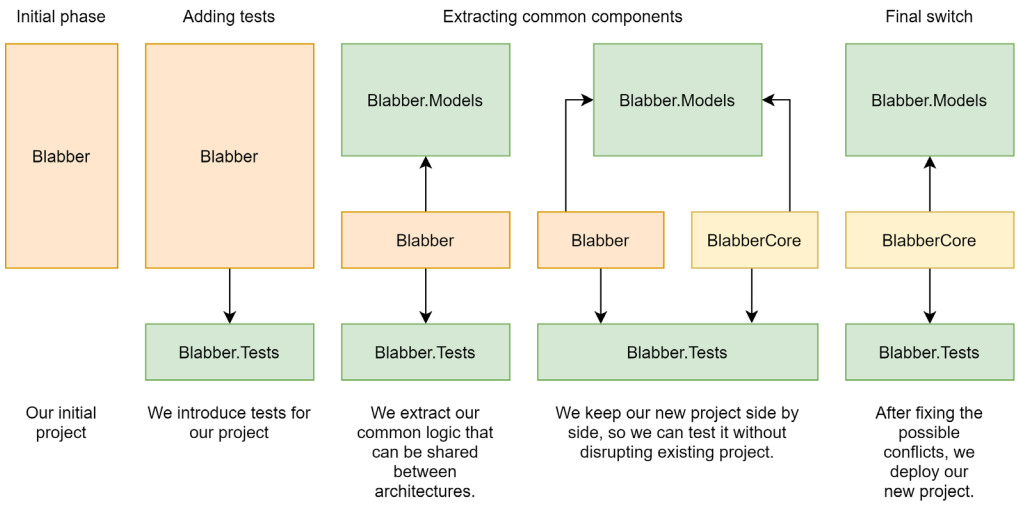


Figure 5.8 Reliable refactoring with tests

5.4 When not to refactor

The good thing about refactoring is that it makes you think about ways of improving code. The bad thing about refactoring is that, at some point, it might become an end rather than a means, pretty much like Emacs. For the uninformed, Emacs is a text editor, a development

environment, a web browser, an operating system, and a post-apocalyptic role-playing game because someone just couldn't hold their horses. The same can happen with refactoring. You start seeing every code as a place for a potential improvement. It becomes such an addiction that you create excuses to make change for the sake of making the change itself, but not consider its benefits. Not only does this waste your time, but it wastes your team's too as they need to adapt to every change you introduce.

You should essentially develop an understanding of good enough code and worthiness when working in the streets. Yes, code can rust away when left untouched, but a good enough code can bear that burden easily. The criteria you need to look for good enough code is:

- Is your only reason for refactoring is "this is more elegant?" That's a huge red flag as elegance is not only subjective, but also vague, therefore meaningless. Try to come up with solid arguments and solid benefits like "this will make this component easier to use by reducing amount of boilerplate we need to write every time we use it", "this will prepare us for migrating to the new library," "this will remove our dependency to the component X", and so forth.
- Does your target component depend on minimal set of components? That indicates that it can be moved or refactored easily in the future. Our refactoring exercises may not benefit us for identifying rigid parts of the code. You can postpone it, until you come up with a more solid improvement plan.
- Does it lack test coverage? That is an immediate red flag to avoid refactoring, especially if the component also has too many dependencies. Lack of testing for a component means that you don't know what you're doing, so, stop doing it.
- Is it a common dependency? That means, even with good amount of test coverage and good justification, you might be impacting the ergonomics of your team by creating a disruption in their workflow. You should consider postponing a refactor operation if the gains you seek isn't enough to compensate the cost.

If any of those criteria is met, you should consider avoiding refactoring, or at least postponing it. Prioritization work is always relative and there is always more fish in the sea.

5.5 Summary

- Embrace refactoring as it provides more benefits than on the surface.
- You can perform large architectural changes in incremental steps.
- Estimate not only costs, but the risks too.
- Always have an either mental or a written roadmap for incremental work when working on large architectural changes.
- Use dependency injection to remove roadblocks like tightly coupled dependencies when refactoring. Reduce code rigidity with the same technique.
- Consider not doing a refactor when it costs more than it brings.

6

Security by scrutiny

This chapter covers:

- Understanding security as a whole
- Leveraging threat models
- Avoiding common security pitfalls like SQL injection, CSRF, XSS, overflows
- Techniques to reduce attacker's capabilities
- Storing secrets correctly

Security is a commonly misunderstood problem throughout history as early as that unfortunate incident at Troy, an ancient city in today's western Turkey. Trojans thought their walls were impenetrable and they were secure, but like modern social platforms, they underestimated thesocial-engineering abilities of their adversaries. Greeks withdrew from battle and left a tall wooden horse figure as a gift. Trojans loved the gesture and let the horse inside their walls to cherish it. In the midnight, the Greek soldiers hidden in the hollow horse got out and opened the gates, letting the hidden Greek armies in, causing the downfall of the city. At least, that's what we know from the *post-mortem blog posts* of Homeros, possibly the first instance of *irresponsible disclosure* in the history.

Post-mortems and responsible disclosures

A post-mortem blog post is a long article usually written after a terribly embarrassing security incident to provide as many details as possible transparently to hide the fact that the management has screwed up.

Responsible disclosure is the practice of publishing a security vulnerability after providing the company, which didn't invest in finding the problem in the first place, an ample time to fix the problem. Companies invented the term to load the act with emotional burden so the researcher would feel guilty, while security vulnerabilities themselves are always called "incidents", and never "irresponsible". I believe that it should have been called something like timed disclosure from the get-go.

Security is both a broad and deep term like in the story of Trojans, which involves human psychology. That's the first perspective you need to embrace: security is never about only software or information; it's about people, and environment too. Because of the vastness of the subject, this chapter can never make you an expert on security. It will make you a better developer with better understanding of it, though.

6.1 Beyond hackers

Software security is usually thought in the terms of vulnerabilities, exploits, attacks, and hackers. But security can be breached because of other, seemingly irrelevant factors too. For example, you could be accidentally logging username and passwords in your web logs, which could be stored on much less secure servers than your database. It happened to billion-dollar companies like Twitter as they found out that they were storing plaintext passwords in their internal logs[1], and an adversary could immediately start using passwords they accessed as opposed to cracking hashed passwords.

Facebook provided an API for developers that let them to browse through users' friend list. A company used that information to generate political profiles of people to influence US elections with precision targeted ads back in 2016. It was a feature that worked exactly as it was designed. There was no bug, no security hole, no backdoors, or no hackers involved. Some people created it, and other people used it, yet the acquired data let people to be manipulated against their will, causing harm.

You'd be surprised to know how many companies leave their databases accessible on internet without any password. Database technologies like MongoDB and Redis don't authenticate users by default; you have to enable authentication manually. Obviously, many developers don't do that, causing massive data leaks.

There is a famous motto among developers and DevOps people: "Don't deploy on Fridays." The logic is simple. If you screw something up, there'll be no one to handle it during the weekend, so do high risk activities closer to the start of the week. Otherwise, it can get really bad both for the staff and the company. Existence of weekends isn't a security vulnerability either, yet it can still lead to catastrophic outcomes.

That brings us to relation between security and reliability. Security, like testing, is a subset of reliability. Reliability of your services, reliability of your data, and reliability of your business. When you look at security in the perspective of reliability, it becomes easier to make security related decisions as you master it along the way when looking at other aspects of reliability such as testing as we've discussed in the previous chapters.

Even if you have zero accountability for the security of the products you develop, taking reliability of your code into account helps you to make certain decisions to get less headaches in the future. Street coders optimize their future too, not just their now. The goal is to do minimal work to achieve great success in your lifetime. Seeing security-related decisions as technical debt for reliability helps you optimize your lifetime as a whole. I recommend this on every product, regardless of potential security impacts. For example, you could be developing an internal dashboard for your access logs, accessed by nobody other than trusted people. I still suggest you applying secure software best practices, like using

[1] "Twitter says bug exposed user plaintext passwords", https://www.zdnet.com/article/twitter-says-bug-exposed-passwords-in-plaintext/

parameterized queries for running SQL statements, which we will go into detail later. It might seem like slightly extra work, but it helps you develop the habit, helping you in the long run. It's not really a shortcut if it prevents you from improving yourself.

Since we've already established that developers are humans in previous chapters, you need to accept that you carry along the weaknesses of humans, primarily miscalculating the probabilities. I know this as a person who used "`password`" as my password almost on all platforms over several years in early 2000s. I'd thought nobody would think that I was that dumb. I turned out to be right; nobody noticed that I was that dumb. Luckily, I've never been hacked, at least by my password getting compromised, but I haven't been a target of many people around that time either. That means I correctly, or randomly, hit the nail on my *threat model*.

6.2 Threat modeling

A threat model is a clear understanding of what could possibly go wrong in the context of security. The assessment of threat model is commonly expressed like "nah, it'll be fine", or "hey, wait a second…." The goal of having a threat model is to prioritize the security measures you need to take, to optimize cost, and increase effectiveness. The term itself sounds very technical because the process can be intricate. But the understanding of threat model isn't.

A threat model effectively lays out what's not a security risk, or not worth protecting against. It's similar to not worrying about a catastrophic draught in Seattle, or not worrying about sudden emergence of affordable housing in San Francisco even though they are still legitimate possibilities.

We actually develop threat models unconsciously. For example, one of the most common threat models could be "I've got nothing to hide!" against threats like hacking, government surveillance, or an ex-partner who was supposed to have become an adult a decade ago. That means we don't really care if our data is compromised and used for whatever purpose. That's mostly because we lack imagination about how our data can be used. Privacy is like a seatbelt in that sense: you don't need it 99% of the time, but when you need it, it can save your life. When hackers find out our SSN and apply credit applications on our behalf and taking all your money leaving you in huge debt, you slowly start to realize that you might have one or two things to hide. When your cellphone data mistakenly matches a murder's time and coordinates, you become the greatest proponent of privacy.

Actual threat modeling is slightly more complicated. It involves analyzing actors, data flow, trust boundaries. There are formal methods developed to create threat models, but unless your primary role is a security researcher, and responsible for the security of the institution you're working at, you don't need a formal approach to threat modeling, but you need to have the basic understanding of it: prioritizing security.

First, you need accept the rule of the land: security problems will hit your app or platform sooner or later. There is no running away from it. "But this is just an internal web site," "but we're behind a VPN," "but this is just a mobile app on an encrypted device," "nobody knows about my site anyway," "but we use PHP"; especially the last one doesn't really help your case.

The inevitability of security problems also emphasizes the relativity of all the things. There is no perfectly secure system. Banks, hospitals, credit scoring companies, nuclear reactors, government institutions, cryptocurrency exchanges, and almost all other institutions have experienced a security incident with varying degrees of severity. You'd think your web site about rating the best cat picture would be exempt from that, but the thing is, your web site can be used as a leverage for sophisticated attacks too. The passwords you store for one of the users might contain the same login information to a nuclear research facility that person works at, because we're not really good at remembering passwords.

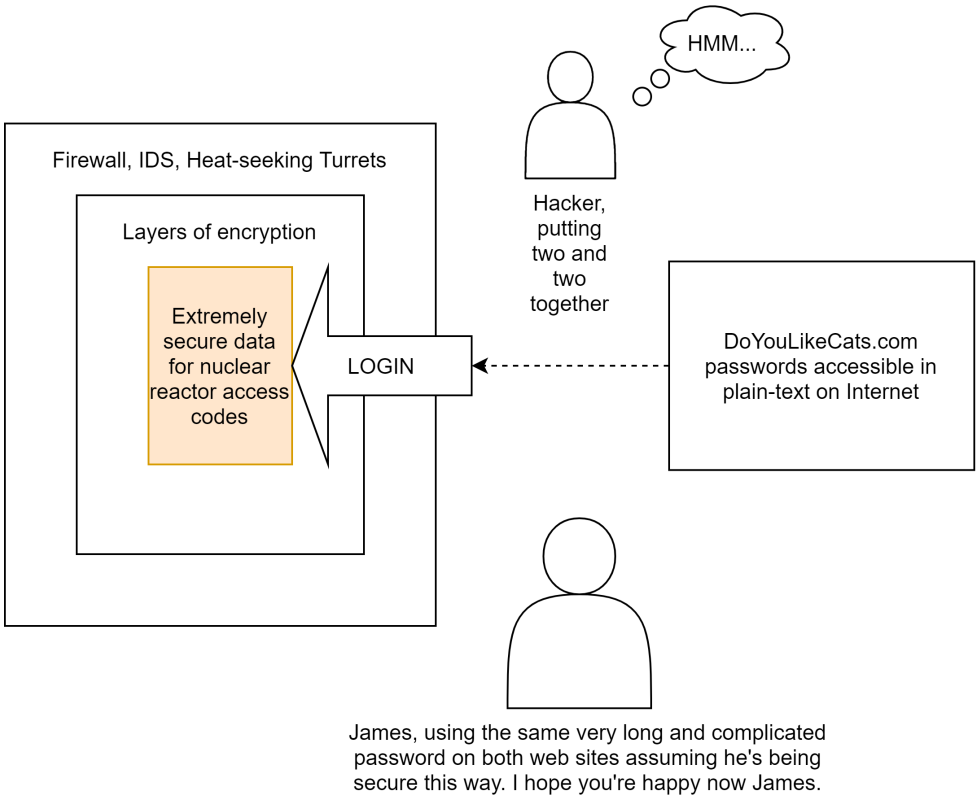


Figure 6.1 Security isn't always about software.

But mostly, hackers don't even know when they hack your web site, because they don't individually walk through all the web sites in the world. They just use bots, to scan for vulnerabilities, and hacker just collects the data afterwards, bots making all the hard work. Well, AI is taking our jobs after all.

6.2.1 Pocket-sized threat models

You may not be supposed to do all the threat modeling for your application. You may not be affected by security incidents either. But you're expected to write minimally secure code and that's not too hard if you follow certain principles. You basically need a mini threat model for your application. It basically encompasses these elements:

- The assets of your application. Basically, anything that you don't want to lose, or leak is an asset, including your source code, design documents, database, private keys, API tokens, server configurations, or your Netflix watchlist.
- Servers that assets reside on. Every server gets accessed by some parties and every server accesses some other servers. It's important for you to know these relationships in order to understand potential problems.
- Information sensitivity. You can assess this by asking yourself the questions "how many people and institutions would be harmed if this information became public?," "what's the seriousness of potential harm?," and "have I been in a Turkish prison?"
- Access paths to resources. Your application has access to your database. Is there any other way to access it? Who does have access? How secure are they? What happens if somebody tricks them into accessing the DB? Can they delete the production database by executing a simple [redacted][redacted][2]? Do they only have access to source code? Then, anyone who has access to source code also has effective access to production DB.

You can come up with a basic threat model on a piece of paper by using that information. It might look like as in figure 6.2 for anyone who uses your application or web site. You can see in the figure that everyone has access to only mobile app and web servers. On the other hand, web servers have access to most critical resources like the database, yet they are exposed to internet. That means your web servers are the riskiest assets exposed to outside world.

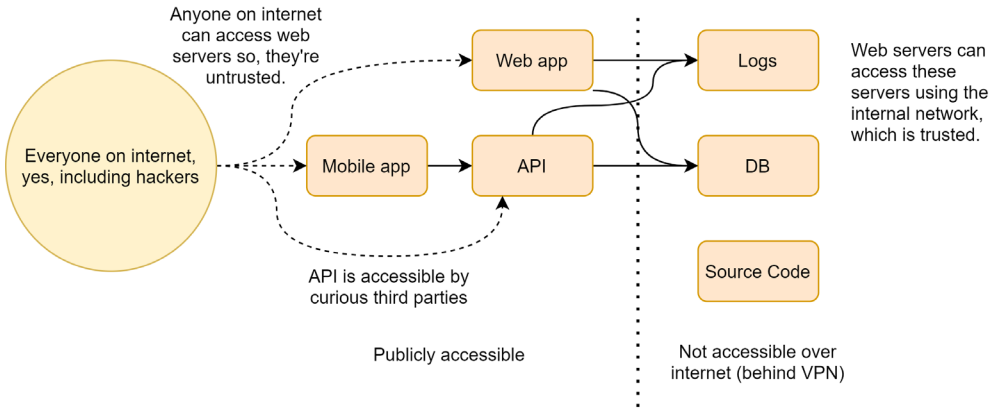


Figure 6.2 Accessibility of servers on a network

[2]Redacted. Classified information. Therefore, our databases are secure.

Besides regular users, you also have other types of users with different access privileges to your servers and assets they contain. In figure 6.3, you can see how different type of roles can access different servers they are allowed to. As you can see, because the CEO loves to access and have control over every little thing, the easiest way to penetrate this server is to send the CEO an email. You'd expect other roles to have limited access to only to the resources that they need access to but that's not usually the case.

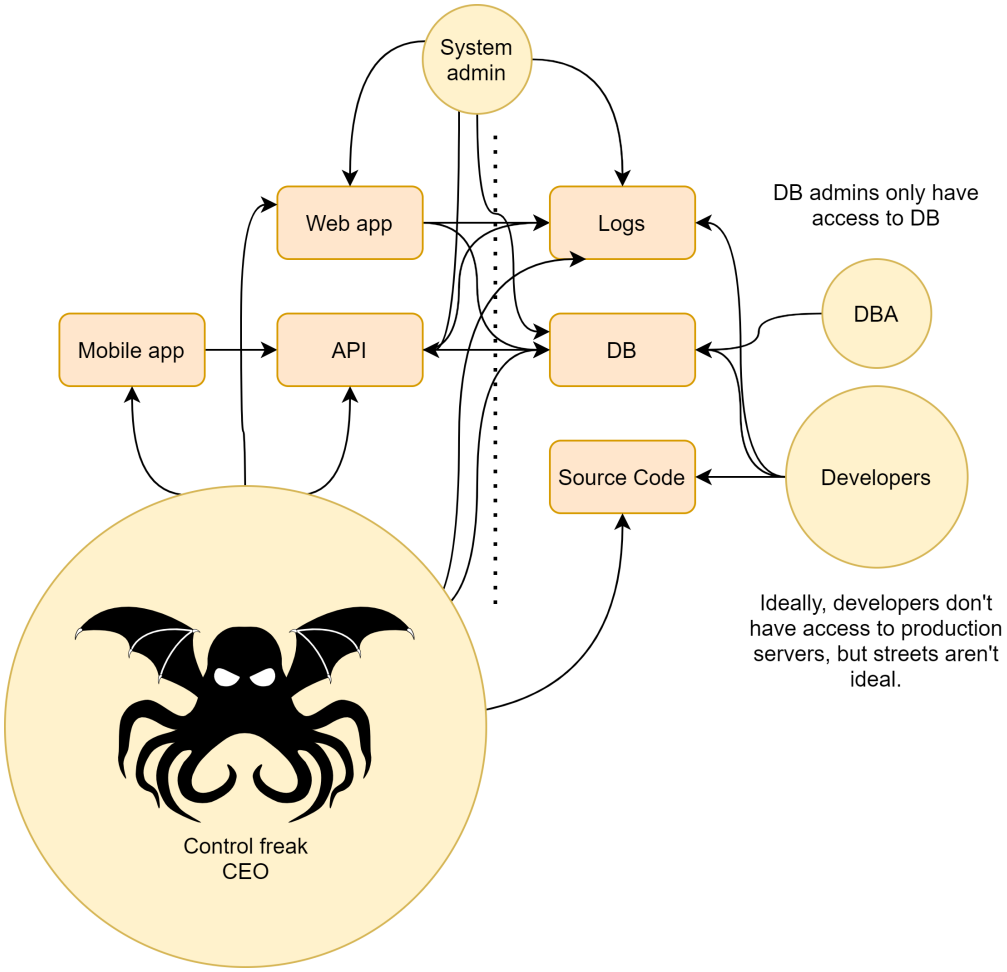


Figure 6.3 Server accessibility based on privileged user types

When you look at this model from 20,000 feet, it's obvious that sending an email to the CEO, asking them to log in to the VPN to check something and redirect them to your phishing

web site is the most obvious way to gain access to everything about a company. A threat model makes such things obvious and makes you understand the risk factors.

If your control freak CEO is the first candidate to harm your business, the code running on the web servers is the second. Not just your code either. You could have a delayed security update on the server, causing it to be taken over. But nothing's worse than just typing some text on a form on the web site to gain access or destroy the entire data on the database.

Your web application or API is one of the easiest entry points, after your CEO of course, for a hacker or a bot to attain their goal. That's because your application is unique. It only exists on your servers. It's only tested by you. All the third-party components on your servers have been through millions of iterations of testing, bug fixing, security audits. Even if you had the budget to do those, you wouldn't have time to do them in the short run.

The goal of a hacker or a bot can vary from simply stopping your service because it's a Rent-a-DoS (Denial of Service) hired by your competitor because they lack any other ways to compete with you, to extracting user data in order to acquire some other valuable resource somewhere with the same password, or just accessing a private data on your servers.

When you have a list of threats available, you can start addressing them by closing the holes. As your web app or API is one of the popular candidates, it's important that you know how to write secure code while writing web applications.

6.3 Write secure web apps

Every application is unique, but there are some really easy to apply practices you can use during your coding that would make your app more resilient to security problems. As Street coders, we'll also inquire why those practices are the best, and when they are not the best. Let's examine the popular attacks on web applications which we can prevent by changing how we write and design our programs.

6.3.1 Design with security in mind

Security is hard to retrofit. That's mostly because of all the design decisions that led you to writing insecure code in the first place. In order to change the security properties of an application, you might need to re-assess your design too. So, it's important to take security into account when designing it. Go over these steps:

- Go over your written or mental threat model. Understand the risks, the costs of making them secure, and the costs of making them secure later in the future.
- Decide on where you store your secrets (DB passwords, API keys) for your app. Make it a hard policy. Assume your source code is accessible by everyone. We will go over the best practices about storing secrets later in the chapter.
- Design for the least privilege. A code ideally shouldn't require any more privilege than it needs to accomplish its task. For example, don't give your app DB Administrator privileges if your app doesn't need to schedule a periodic DB recovery operation. If only a few tasks need higher privileges, consider compartmentalizing them into a separate, isolated entity, such as a separate app. Run web apps under least privileged accounts possible.

- Apply that principle to your entire organization. Employees shouldn't have access to resources that they don't need to perform their daily tasks. CEO shouldn't have access to DB, or any servers at all. That's not because nobody can be trusted, but because their access can also be compromised by external parties.

When you have these down before writing a single line of code for your new app, or even your new feature, you'll be much better in the long run.

In the following sections, some of the topics are only applicable for Web/API development, and the examples used are usually specific to a single library. If you're not doing anything remotely accessible, you can mostly skip to section about storing user secrets. Otherwise, keep on reading.

6.3.2 Usefulness of security by obscurity

Software security is a race against time. Despite that how secure you think your software is, it comes down to how secure people are, how secure everything that surrounds your software is. Every security measure can eventually be broken. It used to be estimated that it would take longer than the lifetime of universe to break a 4096-bit RSA key, but it turns out that it only takes until production of a quantum computer. That means the sole purpose of every security measure is to gain you time, make attacker's work hard.

Security by obscurity is a loathed practice by information security experts. As Benjamin Franklin once said, "Those who try to achieve security by obscurity, deserve neither security nor obscurity." Hmm, he may not have said that exactly, but close enough. The reason for the opposition against security by obscurity is that it doesn't buy you time, or perhaps it does but only marginally. What experts are against is the belief that obscurity is sufficient. It isn't enough, and it's never effective by itself. You should never prioritize it, and only employ it when you have available resources left. But, in the end, it may buy you marginal security.

Let's get this fact straight: marginal security isn't security. It's a temporary band-aid that might keep your project up while it gets to a certain level of growth. In the first year of Eksi Sozluk, I remember keeping administration interface behind an obscure URL with no authentication whatsoever. Let me put it in context though: it was 1999, the web site had 1000 users at most, and I didn't share the URL with anyone. Instead of investing a lot in an elaborate authentication and authorization mechanism I focused on the web site dynamics relevant to users. I definitely knew that it was only a matter of time before someone would find it out though, so I upgraded it to an authenticated system as soon as possible.

Similarly, the web has run over HTTP protocol for a long time and used "Basic" authentication scheme which didn't encrypt passwords, just encoded them in Base64.[3] It was the living testament to security by obscurity. Yes, no sane security expert recommended it, but many web sites used it, knowing the risks or not. If you were on the same network with the user, like a public Wi-Fi access point, you could easily extract passwords and web traffic from the sessions of users who used them. Eventually, Man-In-The-Middle (MITM) attacks and password skimming applications became so prevalent that there was this huge push in the last decade to switch over to HTTPS, HTTP/2, TLS 1.3, and more secure authentication protocols like OAuth2. Security by obscurity worked for decades right in front of us.

[3] Base64 is a binary encoding method that converts unprintable characters into unreadable characters.

That brings us to the point: prioritize security based on your threat model, and if your model permits it, security by obscurity can work for you like how putting a "Beware of the dog" sign can reduce the risk of robberies to a certain extent, even if you don't have any dog.

Perfect security isn't attainable, and you'll always encounter tradeoffs between user experience and security, like how chat app Telegram chose a worse security model than WhatsApp but provided much better usability, so people are switching to it even when they're aware of the consequences. It's really important that you have the same level of awareness for the consequences of the tradeoff decisions that you make. Simply rejecting every measure under the umbrella excuse of "hey, security by obscurity is bad" doesn't help you.

That said, real security is getting cheaper. You had to buy $500 SSL certificates in order to get your web site up and running with HTTPS, now you can do it completely free by using certificates from Let's Encrypt initiative[4]. Having a secure authentication system is now only about plugging a library into your project. Make sure that you're not exaggerating the requirements of getting good security, and not just making excuses to use security by obscurity to have a really bad security. Always prefer real security over security by obscurity when the effort difference is marginal, and risks are considerable.

Obscurity cannot buy you real security, but it can occasionally buy you time until you sort things out.

6.3.3 Don't implement your own security

Security is a complex subject. You should never write your own implementation of a security mechanism, be it hashing, encryption, or throttling. It's perfectly okay to write code as experimentation but don't use your own security code in production. That advice is also commonly called "don't roll your own crypto." Usually, security related specifications expect the reader to have necessary understanding of requirements of developing secure software, and a regular developer can miss critical details while implementing their own, essentially creating zero security.

Take hashing for example. Even a team of expert scientists on cryptography have a hard time creating a cryptographically secure hash algorithm without weaknesses. Almost any hash algorithm before SHA2 has serious security weaknesses.

I don't expect you become so adventurous that you'll try to write your own hashing algorithm but, would you have guessed that you shouldn't even implement your own string comparison function because it's insecure? We'll go into details of this in the section about storing secrets later in this chapter.

You can still create defenses against security simply by changing how you do daily work without implementing anything from scratch. We'll go over these common attack vectors, but that's not an extensive list, rather prioritized samples to show you that attaining decent security may not require huge effort on your part. You can be as effective as you were before and write much more secure software.

[4] Let's Encrypt, https://letsencrypt.org

6.3.4 SQL Injection attacks

SQL injection attack is a long-solved problem, yet it's still one of the popular ways to compromise a web site. It should have disappeared from the face of the earth about the same time as directing career of George Lucas, but somehow persevered, unlike George Lucas.

The attack is quite simple in fact. You have a SQL query running on your web site. Let's say, you want to find a user's id from the username given, like to view the profile of that user, a common scenario. Say, it looks like this:

```
SELECT id FROM users WHERE username='<username here>'
```

A straightforward approach to build this query with the given username as input is to embed the username into query using string manipulation. In listing 6.1, we have a simple `GetUserId` function that takes a username as a parameter and builds the actual query by concatenating string. This is usually the beginner approach to build SQL queries, but it may look okay at first. The code basically creates a command, sets its query to our query after substituting the given username, and executes it. It returns the result as a nullable integer because a record may not exist at all. Also, note that we concatenate strings, but we don't do it in a loop, so as we discussed in earlier chapters, which doesn't have redundant memory allocation overhead.

Optional return values

We specifically use a nullable return type in the GetUserId function in listing 6.1 instead of a pseudo identifier that denotes absence of value, like `-1` or `0`. That's because the compiler can catch unchecked nullable return values in the caller's code and find programming errors. Had we used a regular integer value like zero or minus one, the compiler wouldn't know if that's a valid value or not. In C# versions before 8.0, compiler didn't have these affordances. The future is now!

Listing 6.1 Naïve retrieval of user ID from the database

```
public int? GetUserId(string username) {
  var cmd = db.CreateCommand();
  cmd.CommandText = @"
    SELECT id
      FROM users
      WHERE name='" + username + "'";    #A
  return cmd.ExecuteScalar() as int?;    #B
}
```

#A We build actual query here.
#B Retrieve result or null if the record doesn't exist.

Let's run our function in our mind. Imagine running it with the value "`placid_turn`." If we clean up the extra whitespace, the executed SQL query would look like:

```
SELECT id FROM users WHERE username='placid_turn'
```

Now, consider if the value of username contains an apostrophe, something like "`hackin'`." Our query now would look like this:

```
SELECT id FROM users WHERE username='hackin''
```

Notice what happened there? We introduced a syntax error. That query would fail with a syntax error, the `SqlCommand` class would raise a `SqlException` and the user would see an error page. That doesn't sound so scary. Our hacker would only cause an error to happen. No impact to our service reliability, or the security of our data. Now, consider a username like "`' OR username='one_lame'`." It will throw a syntax error again, but it will look like this:

```
SELECT id FROM users WHERE username='' OR username='one_lame''
```

The first apostrophe closed the quote, and we could continue our query with additional expressions. It's getting scarier. You see, we can manipulate the query to see the records that we're not supposed to see by simply eliminating the syntax error by simply adding double dashes at the end of the username:

```
SELECT id FROM users WHERE username='' OR username='one_lame' --'
```

The double dashes mean an inline comment which assumes the rest of the line is a comment in SQL. It's similar to doubles slashes ("`//`") in all C-style languages, except C. Well, early versions of it at least. That means the query runs perfectly and returns the information for `one_lame` instead of `placid_turn`.

We're not limited to a single SQL statement either. We can run multiple SQL statements by separating them with a semicolon in most SQL dialects. With a long enough username, you can do this:

```
SELECT id FROM users WHERE username='';DROP TABLE users --'
```

That query would delete the table users along with all the records in the table *immediately*, unless there's a lock contention or an active transaction causing a timeout. Think about it, you can do this to a web application remotely, by simply typing a specially crafted username and clicking on a button. You can leak or lose your data. You might be able to recover the lost data from a backup depending on how you're good at it, but you can put the leaked data back into the bottle.

Backups and 3-2-1 backup rule

Remember how we discussed that regressions were the worst type of bugs that lose us time, like destroying a perfectly built building only to build it from scratch in earlier chapters? Having no backups is possibly worse than that. A regression makes you fix a bug again while a lost data makes you *create* the data from scratch. If it's not your data, your users will never bother creating it again. That's one of the first lessons I've learned in my development career. I was a very risk-taking (aka dumb) person in my early career. Back in 1992, I remember writing a compression tool and trying it on its own source code, replacing the original. The tool converted my whole source code into a single byte, and its contents were 255. I'm still confident that there'll be an algorithm in the future to extract those densely packed bits, but I was careless. Version control systems wasn't a thing in personal development back then either. I learned about the importance of having backups right there.

My second lesson in backups was in early 2000. A year had passed since I created Eksi Sozluk, luckily without Y2K issues. I was convinced about importance of backups, but I used to get my hourly backups on the same server and only copied those to a remote server once a week. One day, the disks on the server burned, literally, they spontaneously combusted, data on them was completely unrecoverable. That was when I understood the importance of backups on separate servers. Later in my career, I learned that there was an unspoken rule called "3-2-1 backup rule" in the wild which states that: "have three separate backups, two on separate media, and one at a separate location." Obviously, developing a sane backup strategy requires more thinking than that, and it might never be your job, but that's the minimum you might consider embracing.

WRONG SOLUTION TO SQL INJECTION

How would you consider fixing a SQL injection vulnerability in your app? The first thing comes to mind is escaping: replacing every single apostrophe character (`'`) with double apostrophes (`''`), so a hacker can't close the quote that your SQL query opens as double apostrophes are regarded as regular characters rather than syntactic elements.

The problem with this approach is that there isn't a single apostrophe in Unicode alphabet. The one you escape has the Unicode point value of `U+0027` (APOSTROPHE) while, for example, `U+02BC` (MODIFIED LETTER APOSTROPHE) also represents an apostrophe symbol albeit for a different purpose, and it's possible that the DB technology you're using might treat it is a regular apostrophe or translate all the other apostrophe-lookalikes to a character DB accepts. So, the problem comes down to that you cannot know the underlying technology enough to make the escaping on behalf of it correctly.

IDEAL SOLUTION TO SQL INJECTION

The safest way to solve SQL injection problem is to use *parameterized queries*. Instead of modifying the query string itself, you pass down an additional list of parameters, and the underlying DB provider handles it all. The previous code in listing 6.1 looks like as in listing 6.2 when applied with a parameterized query. Instead of putting the string as a parameter in the query we specify a parameter with "`@parameterName`" syntax and specify the value of this parameter in a separate `Parameters` object associated with that command.

Listing 6.2 Using parameterized queries

```
public int? GetUserId(string username) {
  var cmd = db.CreateCommand();
  cmd.CommandText = @"
    SELECT id
      FROM users
      WHERE username=@username";    #A
  cmd.Parameters.AddWithValue("username", username);    #B
  return cmd.ExecuteScalar() as int?;
}
```

#A Name of the parameter
#B We pass the actual value here.

Voila! Send whatever character you want in the username, there is no way you can change the query. There isn't even any escaping happening anymore as the query and the value of parameters are send in separate data structures.

Another advantage of using parameterized queries is to reduce *query plan cache* pollution. Query plans are execution strategies DBs develop when running a query for the first time. DB keeps this plan in cache and if you run the same query again, it reuses the existing query. It uses a dictionary-like structure, so lookups are O(1), really fast. But, like everything in universe, query plan cache has limited capacity. If you send these queries to the DB, they'll all have different query plan entries in the cache:

```
SELECT id FROM users WHERE username='oracle'
SELECT id FROM users WHERE username='neo'
SELECT id FROM users WHERE username='trinity'
SELECT id FROM users WHERE username='morpheus'
SELECT id FROM users WHERE username='apoc'
SELECT id FROM users WHERE username='cypher'
SELECT id FROM users WHERE username='tank'
SELECT id FROM users WHERE username='dozer'
SELECT id FROM users WHERE username='mouse'
```

Because the query plan cache is limited in size, if you run this query with enough different username values other useful query plan entries will be evicted from the cache and it will get filled with these possibly useless entries. That's what we call query plan cache pollution.

When you use parameterized queries instead, your executed queries will all look the same:

```
SELECT id FROM users WHERE username=@username
SELECT id FROM users WHERE username=@username
SELECT id FROM users WHERE username=@username
SELECT id FROM users WHERE username=@username
SELECT id FROM users WHERE username=@username
SELECT id FROM users WHERE username=@username
SELECT id FROM users WHERE username=@username
SELECT id FROM users WHERE username=@username
SELECT id FROM users WHERE username=@username
SELECT id FROM users WHERE username=@username
```

Since all queries have the same text, DB will be using only a single query plan cache entry for all the queries you run this way. Your other queries will have a better chance to find their spot in this place and you will overall get better performance with your queries in addition to being perfectly safe from SQL injections. All free!

As every recommendation in this book, you'll still have to keep in mind that parameterized query also isn't a silver bullet. You might be tempted to say "Hey, if it's that good, I'll make everything parameterized!," but you shouldn't unnecessarily parameterize, say, constant values as query plan optimizer can find better query plans for certain values. For example, you might want to write this query, although you always use "`active`" as the value for status:

```
SELECT id FROM users WHERE username=@username AND status=@status
```

Query plan optimizer will think that you can send any value as status and will pick a plan that works good enough for all possible values of `@status`. That might mean using the wrong index for "active" and getting a worse performing query. Hmm, maybe a chapter about databases is in order?

WHEN YOU CAN'T USE PARAMETERIZED QUERIES

Parameterized queries are very versatile. You can even use variable number of parameters by naming them `@p0`, `@p1`, `@p2` in code and add parameter values in a loop. Still, there might be cases where you can't really use parameterized queries, or you don't want to, such as to avoid polluting query plan cache again, or you might need certain SQL syntax like pattern matching (think of `LIKE` operators and characters like "`%`" and "`_`") which may not be supported by parameterized queries. What you can do in this case to aggressively sanitize the text, rather than escaping:

If the parameter is a number, parse it into a correct numeric type (`int`, `float`, `double`, `decimal`, etc.) and use that in the query instead of placing it in the string directly, even if that means unnecessarily converting between an integer and a string more than once.

If it's a string, but you don't need any special characters, or you need only a subset of special characters, remove everything else except the valid characters from the string. This is nowadays called "allow-listing" as in having a list of only allowed elements, instead of having a list of denied elements. This makes you avoid accidentally sneaking a malicious character in your SQL queries.

Some DB abstractions may not seem to support parameterized queries the common way. Those can have alternative ways to pass parameterized queries. For example, EF Core uses a `FormattableString` interface to perform the same operation. A similar query to what we've shown in listing 6.2 would look like listing 6.3 in EF Core. The `FromSqlInterpolated` function does a clever thing by using `FormattableString` and C#'s string interpolation syntax together. This way, the library can use the string template, replace arguments with parameters, and build a parameterized query behind the scenes without you knowing.

Interpolate me, complicate me, elevate me (courtesy of the band Rush)

In the beginning, there was `String.Format()`. You could substitute strings with it without dealing with the messy syntax of string concatenation. For example, instead of `a.ToString() + "+" + b.ToString() + "=" + c.ToString()`, you could just write `String.Format("{0}+{1}={2}", a, b, a + b)`. It's easier to understand how the resulting string will look like using `String.Format`, but it's not really straightforward which parameter corresponds to which expression. Then came string interpolation syntax with C# 6.0 which let you write the same expression as `$"{a}+{b}={a+b}"`. It's brilliant; it both lets you understand how the resulting string will look like yet it's straightforward to see which variable corresponds where in the template.

The thing is `$".."` is pretty much a syntactic sugar for `String.Format(..., ...)` syntax, which processes the string before calling the function. If we needed the interpolation arguments in our function itself, we had to write new function signatures similar to `String.Format`'s and call formatting ourselves, complicating our work.

Luckily, the new string interpolation syntax also allows automatic casting to `FormattableString` class which holds both the string template and its arguments. Your function can receive the string and arguments separately if you change the type of the string parameter to `FormattableString`. This leads to interesting uses like delaying

the text processing in logging libraries, or, as in our example in listing 6.3, parameterized queries without processing the string. `FormattableString` is pretty much the same thing in JavaScript's template literals which serve the same purpose.

Listing 6.3 Parameterized query with EF Core

```
public int? GetUserId(string username) {
  return dataContext.Users
    .FromSqlInterpolated(    #A
      $@"SELECT * FROM users WHERE username={username}")    #B
    .Select(u => (int?)u.Id)    #C
    .FirstOrDefault();
}
```

#A Uses string interpolation to create parameterized query.
#B Cast to `FormattableString` when passed to `FromSqlInterpolated`.
#C Make our default value null instead of zero for integers by typecasting to nullable.
#D Return the first value from the query, if there is any.

SUMMARY

Use parameterized queries, not too much, mostly for user input. Parameterization is powerful; it's perfect for keeping your app secure, and the query plan cache decent size simultaneously. Yet, understand the gotchas of parameterization like poor query optimization and avoid using it for constant values.

6.3.5 Cross-site scripting

I think cross-site scripting (I prefer XSS as a shorthand, as the other alternative CSS is also a popular styling language on the web) should have been called "JavaScript injection" for the dramatic effect. Cross-site scripting actually sounds like a competitive sports category in programming, like cross-country skiing. If I didn't know what it is, I could easily be sold to the concept. "Wow, cross-site scripting. That sounds nice. I'd love my scripting to work across sites."

XSS is a two phased attack. The first one is to ability to insert your JavaScript code in the page, and the second phase is to load a larger JavaScript code over the network and execute it on your web page. The advantage of this is multiple. You can capture user's actions, information, and even session by stealing session cookies from another session, which is called "session hijacking."

SORRY, I CAN'T INJECT THAT, DAVE

XSS mainly stems from poorly encoded HTML. It resembles SQL injection in that sense. Instead of providing apostrophe in the user input, we can provide angled brackets to manipulate HTML code. If we can modify the HTML code, we can manipulate it to have `<script>` tags and provide JavaScript code inside.

A simple example is the search feature of web sites. When you search for something the results are listed on the resulting page, but if no results were found, there is usually an error message that says "Your search query for 'flux capacitors for sale' didn't return any results."

So, what happens if we search for "`<script>alert('hello!');</script>`"? If the output isn't properly encoded, there is a chance that you can see something like in figure 6.3.

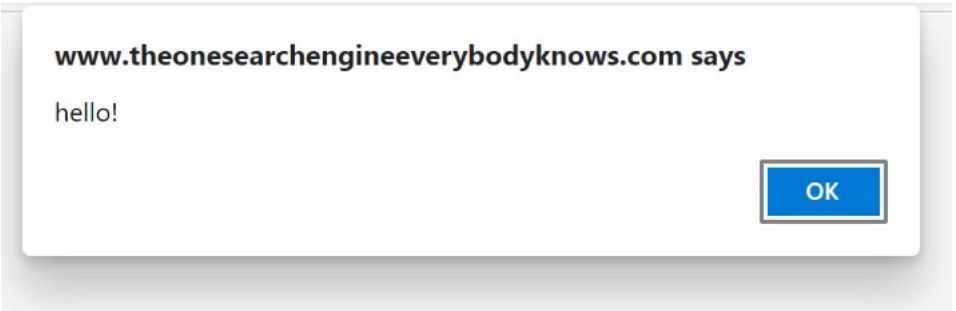


Figure 6.3 Your code runs on someone else's web site, what can go wrong?

If you can inject a simple alert command, you can certainly inject more. You can read the cookies and send them to another web page. You can even load whole JavaScript code from a remote URL and run it on the page. That's where "cross-site" comes from. Allowing JavaScript code to send requests to third party web sites is regarded as a cross-site request.

PREVENTING XSS

Easiest way to defend against XSS is to encode text so that special HTML characters are escaped. That way, they are represented with an equivalent HTML entity instead of their own character, as in table 6.1. Normally, you shouldn't need these tables and perform any encoding using existing, well-tested functions. This is just for your reference to recognize these entities when you see them in your HTML. When escaped like that, user input won't be regarded as HTML and will be shown as plain text as in figure 6.4.

Table 6.1 HTML entity equivalents of special characters

Character	Escaped HTML entity	Alternative
&	&amp;	&#38;
<	&lt;	&#60;
>	&gt;	&#62;
"	&quot;	&#34;
'	&apos;	&#39;

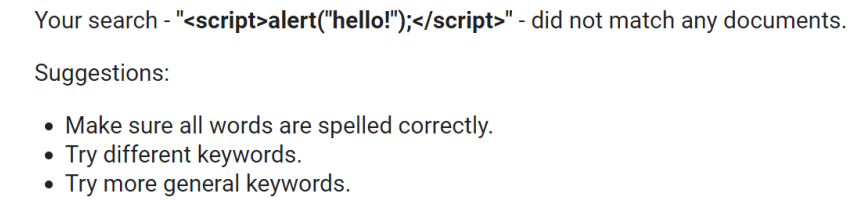


Figure 6.4 When properly escaped, HTML can be quite harmless.

Many modern frameworks actually encode HTML for regular text by default. Consider the Razor template code in our own search engine, Fooble, in listing 6.4. As you can see, we're using "`@`" syntax to directly include a value in our resulting HTML page without performing any encoding at all.

Listing 6.4 An excerpt from our search engine results page

```
<p>
  Your search for <em>"@Model.Query"</em>    #A
  didn't return any results.
</p>
```

#A We use no extra code for encoding.

Despite that we directly output the query string, there is no XSS error as shown in figure 6.5. If you view the source code of the generated web page, you'll see that it's quoted perfectly as in listing 6.5.

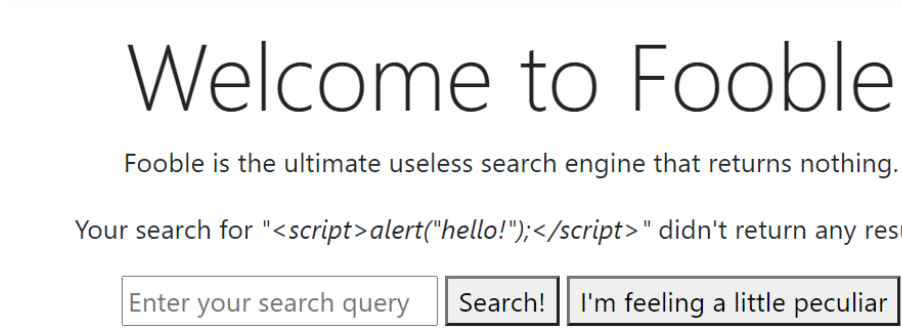


Figure 6.5 We perfectly avoid an XSS attack here.

Listing 6.5 Actual HTML source generated

```
<p>
    Your search for <em>"&lt;script&gt;alert(&quot;hello!&quot;);&lt;/script&gt;"</em>
    didn't return any results.
</p>
```

#A Perfectly escaped, as all things should be.

Then, why do we need to care about XSS at all? That's because, again, programmers are human. Despite the emergence of elegant templating technologies, there are cases you might still think that using raw HTML output can be a good case.

COMMON XSS PITFALLS

One of popular pitfalls is the ignorance of the separation of concerns, such as keeping HTML in your model. You might be tempted to return a string with some HTML embedded in it, because it's easier to integrate to logic into your page code. For example, you might want to return a plain text or a link in your get method, depending on if the text is clickable. For example, with ASP.NET MVC, it might feel easier to type this:

```
return View(isUserActive
    ? $"<a href='/profile/{username}'>{username}</a>"
    : username);
```

and then this in the view:

```
@Html.Raw(Model)
```

instead of creating a new class to hold active and username together, like this:

```
public class UserViewModel {
    public bool IsActive { get; set; }
    public string Username { get; set; }
}
```

and then creating that model in the controller:

```
return View(new UserViewModel()
{
    IsActive = isUserActive,
    Username = username,
});
```

and creating conditional logic in the template to render the username properly:

```
@model UserViewModel
… #A other code here
@if (Model.IsActive) {
    <a href="/profile/@Model.IsActive">
        @Model.Username
    </a>
} else {
    @Model.Username
}
```

It might look like a lot of work do things the right way when only perspective you have is about writing less code. There are ways to avoid a lot of overhead though. You can make your job much easier even by switching to Razor Pages from ASP.NET MVC, but if that's not possible, you can do a lot on existing code too. For example, you can eliminate a separate model by using a tuple instead:

```
return View((Active: isUserActive, Username:  username));
```

This way, you can keep the template code as is. That would save you from creating a new class although there are benefits to it, like reuse. You can get the same benefit from new C# records too, by declaring a view model with a single line of code, immutable too!

```
public record UserViewModel(bool IsActive, string Username);
```

A Razor Pages application already helps you shorten your code as you don't need a separate model class anymore. Controller logic is encapsulated in the ViewModel class created in the page.

If you have to include HTML code in your MVC Controller or Razor Pages ViewModel for some reason that cannot be avoided, consider using `HtmlString` or `IHtmlContent` types instead which let you define well-encoded HTML strings with explicit declarations. If you had to create the same scenario with `HtmlString`, it would look like as in listing 6.6. Since ASP.NET doesn't encode `HtmlString`'s, you wouldn't even need to wrap it with the `Html.Raw` statement.

In listing 6.6, you can see how we implement XSS-safe HTML output. We define Username as `IHtmlContent` instead of `string`. This way, Razor will directly use the content of the string without encoding. The encoding is handled by `HtmlContentBuilder` only for the parts you explicitly specified.

Listing 6.6 Using XSS-safe constructs for HTML encoding

```
public class UserModel : PageModel {
  public IHtmlContent? Username { get; set; }

  public void OnGet(string username) {
    bool isActive = isUserActive(username);
    var content = new HtmlContentBuilder();
    if (isActive) {
      content.AppendFormat("<a href='/user/{0}'>", username);    #A
    }
    content.Append(username);    #B
    if (isActive) {
      content.AppendHtml("</a>");    #C
    }
    Username = content;
  }
}
```

#A This only HTML encodes username.
#B This also encodes username.
#C No encoding applied here at all.

SUMMARY

XSS is easily avoided by not trying to cut corners like injecting HTML by bypassing encoding completely. If you have to inject HTML, be extra careful about encoding the values properly. If you think being XSS-conscious increases your code size, there are ways to reduce the code overhead.

6.3.6 Cross-site request forgery

There is a reason that operations that modify the content on web are performed with POST verb instead of GET on HTTP protocol. You cannot produce a clickable link to a POST address. It can only be posted once. If it fails, your browser warns you if you need to submit it again. Because of that, posting to a forum, logging in, making a significant change is usually denoted with a POST. There is also DELETE and PUT with a similar purpose, but they aren't as commonly used.

That nature of POST makes us trust it more than we give it credit for. The weakness of POST comes from that the original form doesn't have to reside on the same domain the POST request is made. It can be on any web page on the internet. That lets attackers to make POST submissions by tricking you into clicking a link on their web page. Let's just assume that Twitter's delete operation works as a POST operation at a URL like `https://twitter.com/delete/{tweet_id}`.

What if I put a web site on my domain, streetcoder.org/about, and put a form like this, not even using a single line of JavaScript:

```
<h1>Welcome to a super secret web site!</h1>
<p>Please click on the button to continue</p>
<form method="POST" action="https://twitter.com/delete">
    <input type="hidden" name="tweet_id" value="123" />
    <button type="submit">Continue</button>
</form>
```

Luckily, there is no tweet with the ID of 123, but if there was, and Twitter was just a simple startup who didn't know how to protect against CSRF, we would have the ability to delete of someone else's tweet only by asking them to visit our shady web site. If you can use JavaScript, you can even send POST requests without requiring any click to any web form element.

The way to avoid this kind of problem is to use a randomly generated number for every form generated that is replicated on both the form itself and on web site cookies. Since the shady web site can't know those numbers and can't manipulate the cookies of the other web site, they can't really make the request pretend that the request came from the user. The good thing is that usually the framework you use covers for you, all you need to do is to enable generation of tokens and verifying them on the client side. ASP.NET Core 2.0 automatically includes them into forms so you don't need to perform any action, but you need to make sure that those tokens are verified in case you're creating forms in a different way, such as in your own HTML helper. In that case, you need explicitly produce request forgery tokens in your template using a helper like this:

```
<form method="post">
    @Html.AntiForgeryToken()
```

```
    ...
</form>
```

You need to make sure that it's validated on server-side too. Again, this is normally automatic, but in case you have it disabled globally, you can selectively enable it on certain controller actions or Razor pages using the `RequestAntiForgeryToken` attribute.

```
[RequestAntiForgeryToken]
public class LoginModel: PageModel {
    ...
}
```

Since CSRF mitigation is already automatic in modern frameworks like ASP.NET Core, you only need to know the details in order to understand the benefits. But, in case that you need to implement it yourself, it's important that you know how and why it works.

6.4 Draw the first flood

Denial of Service (DoS) is the common name for making your service not work. It can simply be something that causes your server to stop, hang, or crash, or something that can spike the CPU usage, or saturate the available bandwidth. Sometimes the latter type of attacks is called a *flood*. We'll specifically look at floods and how we can resist them.

There isn't a complete solution for floods because regular users in greater numbers can also bring down a web site. It's hard to discern a legitimate user from an attacker. There are ways to mitigate DoS attacks though, so the attacker's capabilities are reduced. A popular one is captcha.

6.4.1 Don't use captcha

Captcha is the bane of the web. It's a popular way to separate wheat from chaff but it's a great friction for humans. The idea is basically asking a mathematically complex problem that a human can solve easily, but automated software used in attacks will have hard time to tackle such as "what shall we have for lunch?"

The problem with captcha is that it's hard for humans too. "Mark all squares with traffic lights," do I just mark the squares that the light bulb itself, or do I also mark the enclosure of the traffic light, do I trace the light pole too? How about those graffiti like art that we're supposed to read easily? Are those letters "rn" or just "m"? Is "5" a letter? Why do you make me suffer?

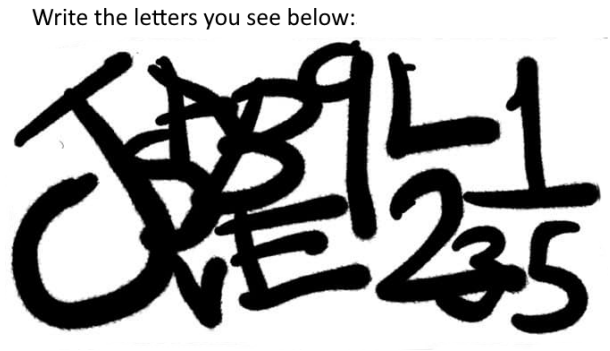


Figure 6.6 Am I human?

Captcha is useful but harmful at the same time as a denial-of-service measure. It's a great friction for humans and you don't want friction in your application in your growth phase. When I first released Eksi Sozluk in 1999, there wasn't even a login. Anyone could write anything on the web site immediately using whatever nickname they wanted. That caused problem shortly as people started to write using each other's nicknames, but that was after people started really loving it. Don't make your users suffer until you get popular enough. That's when bots will discover your web site and attack, and your users will tolerate slightly more pain as they already love your app.

That point applies to all kinds of solutions that involve UX friction for a technical problem. Cloudflare's "please wait for five seconds while we determine if you're an attacker or not" web page is similar. Fifty-three percent of visitors leave a web page when they wait for three seconds for page to load. You're effectively hurting users for the mere chance of someone finding your web site lucrative enough to attack and saturate. Do you want to lose your 53% of your visitors all the time, or all your visitors for an hour once a month?

6.4.2 Captcha alternatives

Write performant code, cache aggressively, and use throttling when necessary. We've already discussed the performance benefits of certain programming techniques up until this point, and we have an entire chapter ahead of us purely about performance optimization.

There is a gotcha to all this though. If you throttle based on an IP address, you'd be throttling everyone from the same IP address, such as a business or a company. When you grow beyond a certain extent, that might hinder your ability to serve requests fast enough to a significant portion of your users.

There is an alternative to throttling: proof of work. You might have heard about proof of work from cryptocurrencies. In order to make a request, your computer or your device is required to solve a really hard problem which is guaranteed to take a certain amount of time. One of the methods is integer factorization. Another proven method is asking the computer the meaning of life, universe, and everything. It's known to take some time.

Proof of work consumes client resources extensively though, which might impact battery life and performance on slower devices. That might also impact user experience badly, even worse than captcha.

You can present more user-friendly challenges, such as requirement of login after your web site passes the barrier of popularity. Checking authentication is cheap but registering to your web site and confirming an email address definitely takes time. That's, again, a user friction. If you ask your users to do something before accessing the content on your web site, such as registering or installing the mobile app, there is a high chance that the user will just swear away and leave your web site. When deciding about reducing an attacker's capability you need to consider those pros and cons.

6.4.3 Don't implement a cache

Dictionary is possibly the most popular structure used in web frameworks. HTTP request and response headers, cookies, and cache entries are all kept in dictionaries. That's because, as we've seen in chapter two, dictionaries are blazingly fast as they have $O(1)$ complexity. Lookups are instantaneous.

The problem with dictionaries is that they're so practical that we might decide to just fire up one to keep a cache of something. There is even a `ConcurrentDictionary` in .NET which is thread-safe, making it an attractive candidate for a hand-rolled cache.

Regular dictionaries included in a framework aren't usually designed for key values based on user input. If an attacker knows which runtime you use, they can cause what we call a *hash collision attack*. They can send requests with many different keys that correspond to the same hash code, causing collisions as we've seen in chapter two, which causes lookups to get closer to $O(N)$ instead of $O(1)$, bringing the application to its knees.

Custom dictionaries developed for web facing components usually use a different hash code algorithm with better distribution properties and therefore less collision probability, such as SipHash. Such algorithms can be slightly slower than regular hash functions in average, but because of their resistance against collision attacks, they perform better for worst case.

Dictionaries also don't have an eviction mechanism by default. They grow indefinitely. That might look okay when you test it locally but can fail spectacularly in production. Ideally, a cache data structure should be able to evict older entries to keep memory usage in check.

Because of all these factors, consider leveraging an existing cache infrastructure preferably provided by the framework, whenever you think of "Hey, I know, I'll just cache these in a dictionary."

6.5 Storing secrets

Secrets (passwords, private keys, API tokens) are the keys to your kingdom. They are small pieces of data, yet they provide disproportional amount of access. You've got the password of the production DB? Then, you've got access to everything. You've got an API token? You can do whatever that API permits you to do. That's why secrets have to be part of your threat model.

Compartmentalization is one of the best mitigations against security threats. Storing secrets safely is one of the ways to achieve it.

6.5.1 Keeping secrets in source code

Programmers are great at finding the shortest path to a solution. That includes taking shortcuts and cutting corners. That's why putting a password in the source code is our default tendency. We love rapid prototyping and that's because we hate anything that causes friction to our flow.

You might think that keeping secrets in source code is okay, like because nobody other than you have access to the code, or because developers have already access to the production DB passwords, therefore keeping the secret in source code wouldn't hurt.

The problem is that you don't take time dimension into account. In the long run, all source code gets hosted on GitHub. Source code doesn't get treated with as the same level of sensitivity as your production DB but contains the keys to it. Your customers can request the source code for contractual purposes. Your developers can keep local copies of source code to review it, and their computer can get compromised. Developers can't keep production DB the same way because it's usually too big to handle, and they associate a higher level of sensitivity to it.

RIGHT STORAGE

If you don't have your secrets in your source code, how would source code know the secret? You can keep it in the DB itself, but that creates a paradox. Where do you store the password to the DB, then? It's also a bad idea because it unnecessarily puts all protected resources in the same trust group with DB. If you have the password to the database, you have everything. Say, you're running Pentagon's IT, and you keep nuclear launch codes in the employee database, because employee database is well-protected. That creates an awkward situation when an accountant accidentally opens the wrong table in the database. Similarly, your app might have API access to more valuable resources than your database. You need to consider that disparity in your threat model.

The ideal way is to store these in a separate storage that's designed for that purpose, such as a password manager as cold storage and a cloud key vault (Azure Key Vault, AWS KMS). If your web servers and DB are in the same trust boundary in your threat model, you can simply add those secrets into environment variables on your server. Cloud services let you set up environment variables through their administration interface.

Modern web frameworks support various storage options for secrets, backed by the operating system's or cloud provider's secure storage facilities in addition to environment variables that can directly map into your configuration. Say, you have this configuration for your application:

```
{
    "Logging": {
        "LogLevel": {
            "Default": "Information"
        }
    },
    "MyAPIKey": "somesecretvalue"
}
```

You don't want to keep `MyAPIKey` in your configuration because anyone with source access would have access to the API key. So, you go ahead and remove the key there, and pass it as an environment variable in the production. On a developer machine, instead of using the environment variable, you can use user secrets instead. Using .NET you can initialize and set up user secrets by running `dotnet` command:

```
dotnet user-secrets init –id myproject
```

That initializes the project to use "myproject" id as an access identifier to relevant user secrets. You can then add user secrets for your developer account by running this command:

```
dotnet user-secrets set MyAPIkey somesecretvalue
```

Now, when you set up user secrets to be loaded in your configuration, the secrets will be loaded from user secrets file and will override the configuration. You can access your secret API key the same way you access the configuration:

```
string apiKey = Configuration[“MyAPIKey”];
```

Cloud services like Azure or AWS let you configure the same secrets through their environment variables, or key vault configurations.

6.5.2 Data shall be leaked

The popular web site, Have I Been Pwned?[5] is a notification service for leaked passwords against your email. As of writing this section, I seem to have been "pwned[6]" 16 times in different data leaks. Data leaks. Data have leaked, and data shall leak. You should always assume the risk of data going public and design against it.

DON'T COLLECT THE DATA YOU DON'T NEED

Your data cannot be leaked if it doesn't exist in the first place. Be aggressive about saying no to collecting a data except which you don't think your service could function without. There are side benefits like less storage requirements, higher performance, and less data management work, and less friction to the user. For example, many web sites require first name and last name when registering to a web site. Do you really need that data?

There is some data that you may not be able to do without, like passwords. However, responsibility of having someone's password is great because people tend to use the same password across multiple services. That means, if your password data leaks, the user's bank

[5] Have I Been Pwned, https://haveibeenpwned.com

[6] "Pwned" is a modified form of "owned" as in being dominated by a hacker. It's slang for having your ass handed to you. Example: "I got pwned because I chose my birthdate as my PIN."

accounts might get compromised too. You might say that that's on the user for not using a password manager and not using accounts, but you're dealing with humans here. There are simple things that you can do that prevent this from happening.

THE RIGHT WAY OF PASSWORD HASHING

The most common way to prevent passwords from being leaked is to use a hashing algorithm. Instead of storing passwords, you store a cryptographically secure hash of the password. We can't use any hashing algorithm, like `GetHashCode()` from chapter two, because regular hash algorithms are trivial to break or cause collisions with. Cryptographically secure hash algorithms are deliberately slow and resistant to several other forms of attacks.

Cryptographically secure hash algorithms vary in their characteristics. For password hashing, the preferred method is to use an algorithm that uses multiple iterations of the same algorithm many times in order to slow down the execution. Similarly, modern algorithms may also require a lot of memory relative to the work they're doing in order to prevent attacks by custom manufactured chips specifically designed to crack a certain algorithm.

Never use single iteration hash functions, even if they are cryptographically secure, such as SHA2, SHA3, and God forbid never MD5 or SHA1 as they are long broken. Cryptographic security property only ensures that the algorithm has exceptionally low collision probability; it doesn't ensure that they are resistant to brute force attacks. In order to get brute force resistance, you need to ensure that the algorithm will work really slow.

One of the common hash functions that are designed to work slow is PBKDF2 which sounds like a Russian secret service subdivision but stands for *Password-Based Key Derivation Function Two*. It can work with any hash function as it only runs them in a loop and combines the results. It uses a variant of SHA1 hash algorithm which is now considered a weak algorithm and shouldn't be used in any application anymore as it's getting more trivial to create a collision with SHA1 every day.

Unfortunately, PBKDF2 can be cracked relatively quickly today as it can be processed in parallel on GPU, and there are specialized ASIC (custom chip) and FPGA (programmable chip) designs for cracking it. You don't want an attacker to try combinations too fast when they're trying to crack your data that just leaked. There are newer hash algorithms like *bcrypt*, *scrypt*, and *Argon2* which are resistant to GPU or ASIC-based attacks too.

All modern brute-force resistant hash algorithms take either a difficulty coefficient as a parameter, or a number of iterations. You should make sure that your difficulty settings aren't too high that it becomes a DoS attack to attempt login on the web site. You probably shouldn't aim any difficulty that takes more than 100ms on your production server. I strongly recommend benchmarking your password hashing difficulty to make sure it doesn't hurt you because changing hash algorithms on the road is difficult.

Modern frameworks like ASP.NET Core provide password hashing functionality out of the box, and you don't really even need to know how it works, but its current implementation relies on PBKDF2 which is a bit behind in security as we've discussed. It's important make conscious decisions about proper hashing.

When picking an algorithm, I recommend favoring one that's supported by the framework that you use. If that's not available, then you should go for the most tested one. Newer algorithms usually aren't tested and verified as much as the older ones.

COMPARE STRINGS SECURELY

So, you've picked an algorithm, and you store hashes of the passwords instead of the passwords themselves. Now, all you need to do is to read the password from the user, hash it, and compare it with the password on the DB. Sounds simple, right? That could easily be a simple loop comparison as in listing 6.8. You can see that we implement a straightforward array comparison. We first check the lengths, and then we iterate in a loop to see if every element is equal. If we find a mismatch, we return immediately, so we don't bother to check the rest of the values.

Listing 6.8 A naïve comparison function for two hash values

```
private static bool compareBytes(byte[] a, byte[] b) {
  if (a.Length != b.Length) {
    return false;    #A
  }
  for (int n = 0; n < a.Length; n++) {
    if (a[n] != b[n]) {
      return false;    #B
    }
  }
  return true;    #C
}
```

#A Length mismatch check, just in case
#B Value mismatch
#C Success!

How can that code be not secure? The problem comes from our mini optimization of bailing out early when we find mismatched values. That means we can find out how long is the match by measuring how fast the function returns as in figure 6.7, and we can find the correct hash if we know the hash algorithm, by producing passwords that correspond to a certain first value of the hash, then first two values, and it goes on. Yes, the timing differences will be little, milliseconds, maybe nanoseconds, but they can still be measured against a baseline. If it can't be measured, measurements can be repeated to get more accurate results. It's way faster than to try every possible combination.

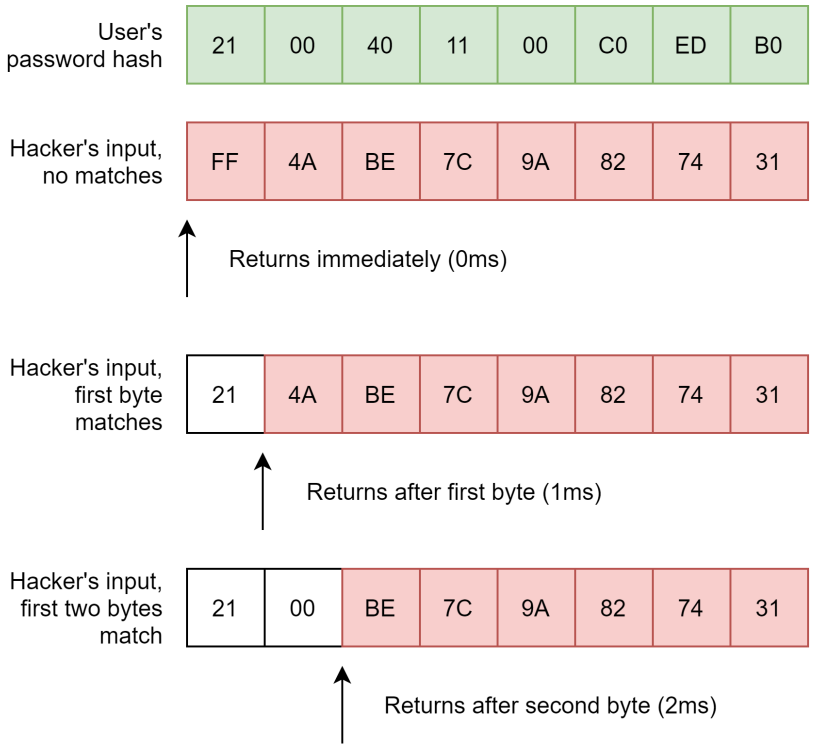


Figure 6.7 How fast comparison can help attackers to figure out your hash

To solve this, you need a comparison function that takes a constant time as in listing 6.9. Instead of returning early, we keep a result value and keep comparisons going even if the comparison fails. So, all our comparisons take constant value, avoiding leaking the hash values of users.

Listing 6.9 Secure hash comparison

```
private static bool compareBytesSafe(byte[] a, byte[] b) {
  if (a.Length != b.Length) {
    return false;    #A
  }
  bool success = true;
  for (int n = 0; n < a.Length; n++) {
    success = success && (a[n] == b[n]);    #B
  }
  return success;    #C
}
```

#A This is an exceptional case. It will never be hit ideally, so we keep it.
#B We constantly update our result variable without finishing early.
#C We return the final result.

DON'T USE FIXED SALTS

Salts are additional values introduced into password hashing algorithm in order to make values deviate even though they are for the same hash values. The reason for that is, you don't want the attacker to figure out all the same passwords by just guessing only hash value of one. This way even if every user's password is `hunter2`, all users will have different hash values, making attacker's life harder.

Developers can find using well-known values for hash salts, like hash of user's name, or user's identifier secure enough because they're usually easier to generate than generating an array of random values, but that's a completely unnecessary shortcut for way less security. You should always use random values for salts, but not just regular pseudorandom values either, you need values generated by a *cryptographically secure pseudorandom generator,* or a *CSPRNG*.

OH RANDOM, OH CHANCE!

Regular random values are generated with simple and predictable algorithms. Their goal isn't to create real unpredictability, but just an imitation of it. They're okay if you're writing an unpredictable enemy in your game, they're okay to pick today's featured post on your web site. They're fast, but they aren't secure. They can either be predicted, or the search space for valid random values can be narrowed down as they tend to repeat themselves in relatively shorter intervals. People managed to figure out the random value generator algorithms of slot machines in casinos in Las Vegas in the old times when the designers of those machines didn't know any better.

You need cryptographically secure pseudorandom numbers, because they're extremely hard to predict, they use multiple strong entropy sources like machine's hardware components that can provide such information and use more complex algorithms. As a result of that, they're naturally slower. So, they should usually only be used in the context of security.

Many cryptographically secure hash libraries provide a hash generation function that only receives length of the salt, not the salt itself, so the library takes care of generating that random salt for you, you can retrieve it from the results as in listing 6.10, which uses PBKDF2 as an example. We create an implementation of RFC2898 key derivation function. It's a PBKDF2 with HMAC-SHA1 algorithm. We use the `using` statement there because security primitives can use operating system's unmanaged resources, and it's good to have them cleaned up when they leave the scope. We leverage a simple record to return both hash and the newly generated salt in a single package.

Listing 6.10 Generating cryptographically secure random values

```
public record PasswordHash(byte[] Hash, byte[] Salt);    #A

private PasswordHash hashPassword(string password) {
  using var pbkdf2 = new Rfc2898DeriveBytes(password,
    saltSizeInBytes, iterations);    #B
  var hash = pbkdf2.GetBytes(keySizeInBytes);    #C
  return new PasswordHash(hash, pbkdf2.Salt);
}
```

#A Our record that holds hash and salt values
#B Creating an instance of hash generator.
#C We generate the hash value here.

GUIDs AREN'T RANDOM

GUIDs (Globally Unique Identifiers), or UUIDs (Universally Unique Identifiers), are random looking numbers like `14e87830-bf4c-4bf3-8dc3-57b97488ed0a`. They used to be generated based on obscure data like a network adapter's MAC address, or system date/time. Nowadays, they're all random, but they're aimed to be unique, not necessarily secure. They can still be predicted as there is no guarantee that they would be using a cryptographically secure random generator. You shouldn't rely on randomness of GUIDs, for, let's say, generating an "activation token" when sending out confirmation email to your newly registered users. Always use CSPRNGs for generating security-sensitive tokens.

6.6 Summary

- Use either mental or paper threat models to prioritize security measures and identify weaknesses.
- Design with security in mind first, as retrofitting security can be hard.
- Security by obscurity isn't real security, but it can be a real detriment; prioritize it as such.
- Don't implement your own security primitives, even when it comes to comparing two hash values. Trust well-tested and well-implemented solutions.
- User input is evil.
- Use parameterized queries against SQL injection attacks. If you can't use parameterized queries for any reason, validate and sanitize user input aggressively.
- Make sure user input is properly HTML encoded when being included in the page to avoid XSS vulnerabilities.
- Avoid captcha especially in your growth phase to deter DoS attacks; try other methods like throttling and aggressive caching first.
- Store secrets in separate secret stores rather than the source code.
- Store password hashes in your database with strong algorithms that are designed for the purpose.
- Use cryptographically secure pseudorandom numbers in security related contexts, never GUIDs.

7

Opinionated optimization

This chapter covers:

- Embracing premature optimization
- Taking a top-down approach to performance problems
- Optimizing CPU and I/O bottlenecks
- Making safe code faster and unsafe code safer

Programming literature on optimization always starts with a famous quote by the famous computer scientist Donald Knuth: "Premature optimization is the root of all evil." Not only is the statement wrong, it's also always misquoted. First, it's wrong because everybody knows that the root of all evil is Object Oriented Programming since it leads to bad parenting and class struggle. Second, it's misquoted because the actual quote is more nuanced. This is almost another case of "lorem ipsum" which is gibberish because it is quoted from the middle of an otherwise meaningful Latin text. Knuth's actual quote is "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%."[1]

I claim that premature optimization is the root of all learning. Don't hold yourself back for something you are so passionate about. Optimization is problem solving, and premature optimization is creating non-existent, hypothetical problems to solve, like how chess players set up pieces to set themselves up for a challenge. It's a good exercise. You can always throw away your work as we've already discussed in chapter three and keep the wisdom you gained. Exploratory programming is a legitimate way to improve your skills as long as you're in control of the risks and the time. Don't deprive yourself of learning opportunities.

There is a reason that people try to discourage you from premature optimization though. Optimization can bring rigidness to the code, making it harder to maintain. Optimization is an

[1] Donald Knuth let me know that his quote in the original article had been revised and reprinted in his book Literate Programming. Getting a personal response from him was one of the greatest highlights of my writing process.

investment, and its return heavily relies on how long you can keep it. If specifications change in the future, the optimizations you've performed can have you dug into a hole, painful to get out. More importantly, you could be trying to optimize a problem that doesn't exist in the first place and making the reliability of your code worse.

For example, you could have a file copying routine and you might know that the larger buffer sizes you read and write at once, the faster the whole operation becomes. You might be tempted to just read everything in memory and write it, so that you have the maximum possible buffer size. That might make your app consume unreasonable amounts of memory or cause it to crash when it tries to read an exceptionally large file. You need to understand the tradeoffs you're making when you're optimizing, which means you must correctly identify the problem you need to solve.

7.1 Solve the right problem

Slow performance can be fixed in many ways, and depending on the exact nature of the problem, its effectiveness and how much time you spend implementing it can vary drastically. The first step to understand the true nature of a performance problem is to find out if there is a performance problem in the first place.

7.1.1 Simple benchmarking

Benchmarking is the act of comparing performance metrics. It may not help you identify the root cause of a performance problem, but it can help you identify its existence. There are libraries like BenchmarkDotNet[2] which makes it extremely easy to implement benchmarks, with safety measures to avoid statistical errors. But, even if you don't use any library, you can simply use a timer just to understand the execution time of the pieces of your code.

Something I have always wondered is how much faster the `Math.DivRem()` function can be than regular division and remainder operation. It's been always suggested to use `DivRem` if you're going to need the result of the division and the remainder at the same time, but I've never had the chance to test if the claim holds up, until now:

```
int division = a / b;
int remainder = a % b;
```

That code looks very primitive, and therefore it's easy to assume that the compiler can optimize it just fine while `Math.DivRem()` version looks like an elaborate function call:

```
int division = Math.DivRem(a, b, out int remainder);
```

> **KNOW IT RIGHT** You might be tempted to call `%` operator the modulus operator, but it's not. It's the remainder operator in C or C#. There is no difference between the two for positive values, but negative values produce different results. For example, `-7 % 3` is -1 in C#, while it's 2 in Python.

You can create a benchmark suite right away with BenchmarkDotNet, and it's great for *micro-benchmarking*, a type of benchmarking where you measure small and fast functions

[2] BenchmarkDotNet, https://github.com/dotnet/BenchmarkDotNet

because either you're out of options or your boss is on vacation. BenchmarkDotNet can eliminate the measurement errors related to fluctuations or the function call overhead. You can see the code that uses BenchmarkDotNet in listing 7.1 that tests the speed of `DivRem` versus manual division/remainder operations. We basically create a new class that describes the benchmark suite with benchmarked operations marked with `[Benchmark]` attributes. BenchmarkDotNet itself figures out how many times it needs to call those functions to get accurate results because a one-time measurement or running benchmarks only a few iterations is susceptible to errors. We use multi-tasking operating systems, and other tasks running in the background can impact the performance of the code you're benchmarking on these systems. We mark the variables used in calculation with `[Params]` attribute to prevent the compiler from eliminating the operations it deems unnecessary. Compilers are easily distracted but they're smart.

Listing 7.1 Example BenchmarkDotNet code

```
public class SampleBenchmarkSuite {
  [Params(1000)]    #A
  public int A;

  [Params(35)]    #A
  public int B;

  [Benchmark]    #B
  public int Manual() {
    int division = A / B;
    int remainder = A % B;
    return division + remainder;    #C
  }

  [Benchmark]     #B
  public int DivRem() {
    int division = Math.DivRem(A, B, out int remainder);
    return division + remainder;    #C
  }
}
```

#A We're avoiding compiler optimizations.
#B Attributes mark the operations to be benchmarked.
#C We return values, so the compiler doesn't throw away computation steps.

You can run these benchmarks simply by creating a console application and adding a using line and Run call in your `Main` method:

```
using System;
using System.Diagnostics;
using BenchmarkDotNet.Running;

namespace SimpleBenchmarkRunner {
  public class Program {
    public static void Main(string[] args) {
      BenchmarkRunner.Run<SampleBenchmarkSuite>();
    }
  }
}
```

If you run your application, the Benchmark results would be shown after a minute of running:

```
| Method |    a |  b |     Mean |     Error |    StdDev |
|------- |----- |--- |---------:|----------:|----------:|
| Manual | 1000 | 35 | 2.575 ns | 0.0353 ns | 0.0330 ns |
| DivRem | 1000 | 35 | 1.163 ns | 0.0105 ns | 0.0093 ns |
```

It turns out `Math.DivRem()` is two times faster than performing division and remainder operations separately. Don't be alarmed by Error column as it's only a statistical property to help the reader assess accuracy when BenchmarkDotNet doesn't have enough confidence in the results. It's not the standard error but half of the 99.9% confidence interval.

Although BenchmarkDotNet is dead simple and comes with features to reduce statistical errors, you may not want to deal with an external library for simple benchmarking. In that case, you can just go ahead and write your own benchmark runner using a Stopwatch as in listing 7.2. You can simply iterate in a loop long enough to get a vague idea about relative differences in performance of different functions. We're reusing the same suite class we created for BenchmarkDotNet but using our own loops and measurements for the results.

Listing 7.2 Homemade benchmarking

```
private const int iterations = 1_000_000_000;

private static void runBenchmarks() {
  var suite = new SampleBenchmarkSuite {
    A = 1000,
    B = 35
  };

  long manualTime = runBenchmark(() => suite.Manual());
  long divRemTime = runBenchmark(() => suite.DivRem());

  reportResult("Manual", manualTime);
  reportResult("DivRem", divRemTime);
}

private static long runBenchmark(Func<int> action) {
  var watch = Stopwatch.StartNew();
  for (int n = 0; n < iterations; n++) {
    action();    #A
  }
  watch.Stop();
  return watch.ElapsedMilliseconds;
}

private static void reportResult(string name, long milliseconds) {
  double nanoseconds = milliseconds * 1_000_000;
  Console.WriteLine("{0} = {1}ns / operation",
    name,
    nanoseconds / iterations);
}
```

#A We call the benchmarked code here.

And, when we run it the result is relatively the same:

```
Manual = 4.611ns / operation
DivRem = 2.896ns / operation
```

Note that our benchmarks don't try to eliminate function call overhead, or the overhead of the for loop itself, so they seem to be taking longer, but we still successfully observe that `DivRem` is still twice as fast as manual division and remainder operations.

7.1.2 Performance vs responsiveness

Benchmarks can only report relative numbers. They cannot tell you if your code is fast or slow, but they can tell you if it's slower or faster than some other code. A general principle about slowness in the eyes of a user is that any action that takes more than 100ms feels delayed, and any action that takes more than 300ms is considered sluggish. Don't even think about taking a full second. Most users will leave a web page or an app if they wait for more than three seconds. If a user's action takes more than five seconds to respond, it might as well take about the lifetime of the universe — it doesn't matter at that point. Figure 7.1 illustrates that point.

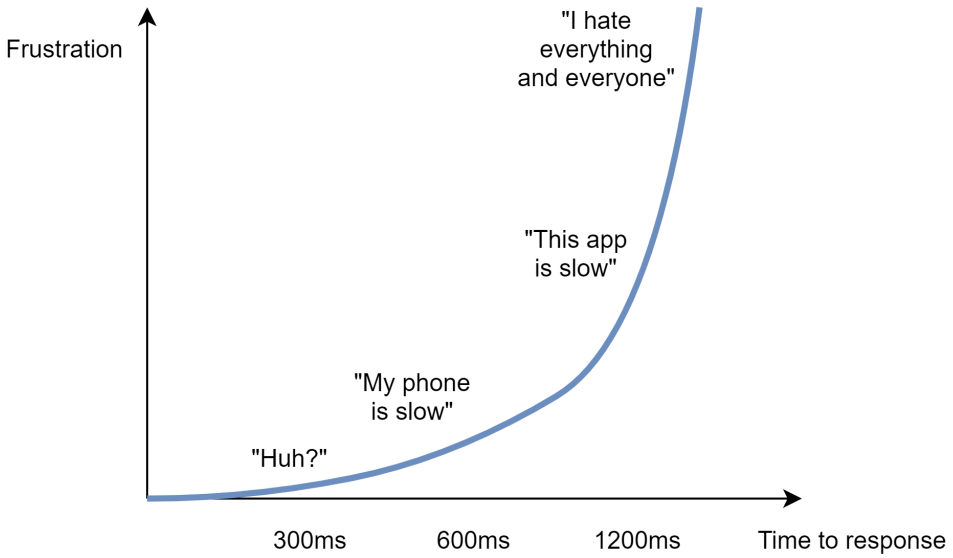


Figure 7.1 Response delays vs frustration

Obviously, performance isn't always about responsiveness. In fact, being a responsive app might require performing an operation slower. For example, you might have an app that replaces faces in a video with your face using machine learning. Because such a task is computationally intensive, the fastest way to calculate it is to do nothing else until the job's done. But that would mean a frozen UI, making the user think that something's wrong and quit the app. So, instead of doing the computation as fast as you can, you instead spare

some of the computational cycles to show a progress bar, perhaps calculate estimated time remaining, show a nice animation that can entertain the users while waiting. In the end, you have slower code but more successful code.

That means, even if benchmarks are relative, you can still have some understanding of slowness. Peter Norvig came up with the idea in his blog[3] of listing latency numbers to have a context of how things can be slower by orders of magnitude in different contexts. I'd like to create a similar table with my own back of the envelope calculations in table 7.1. You can come up with your own numbers by looking at this:

Table 7.1 Latency numbers in various contexts

Read a byte from	Time
A CPU register	1ns
CPU's L1 cache	2ns
RAM	50ns
NVMe disk	250,000ns
Local network	1,000,000ns
Server on the other side of the world	150,000,000ns

Latency affects performance too, not just user experience. Your database resides on a disk, and your database server resides on a network. That means, even if you write the fastest SQL queries, and define the fastest indexes on your database, you're still bound to the laws of physics and can't get any result faster than a millisecond. Every millisecond you spend eats away from your total budget which is ideally less than 300ms.

7.2 Anatomy of sluggishness

In order to understand how to improve performance, you must first understand how performance fails. As we've seen, not all performance problems are about speed — some of them are about responsiveness. The speed part, though, is related to how computers work in general, so it's a good idea to get yourself acquainted with some low-level concepts. This would help you understand the optimization techniques later in the chapter too.

CPUs are chips that process instructions they read from RAM and perform them repetitively in a never-ending loop. You can imagine it like a wheel turning, and every rotation of the wheel would typically perform another instruction as depicted in figure 7.2. Some operations can take multiple turns, but the basic unit is a single turn, popularly known as a *clock cycle*, or a *cycle* for short.

[3] "Teach Yourself Programming in Ten Years", Peter Norvig, http://norvig.com/21-days.html#answers

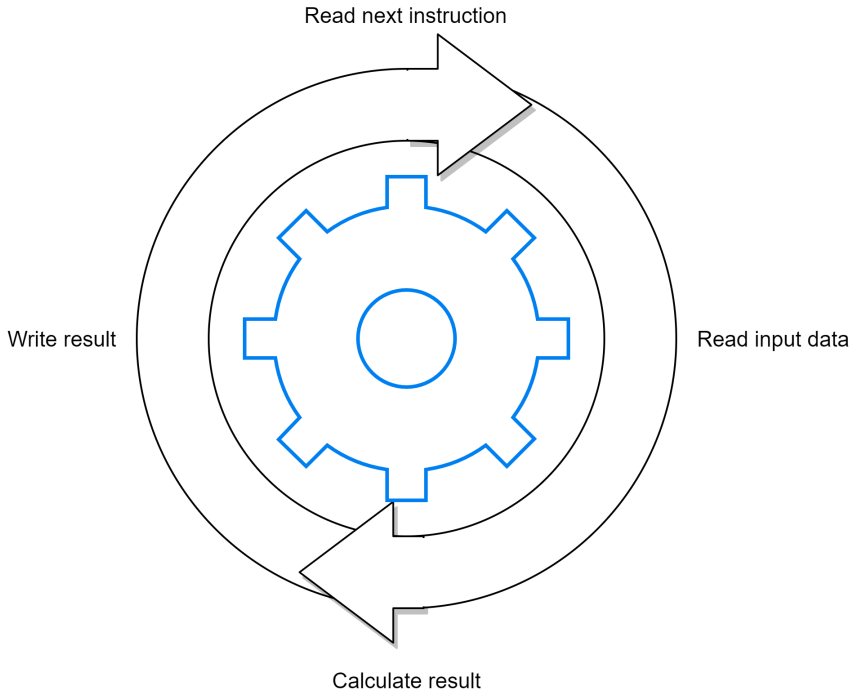


Figure 7.2 The 20,000 feet anatomy of a single CPU cycle

The speed of a CPU, typically expressed in Hertz, indicates how many clock cycles it can process in a second. The first electronic computer ENIAC could process 100,000 cycles a second, shortened as 100KHz. An antique 4Mhz Z80 CPU which belonged to my 8-bit home computer back in the 80's could only process 4 million cycles per second. A modern 3.4Ghz AMD Ryzen 5950X CPU can process 3.4 *billion* cycles in a second on each of its cores. That doesn't mean CPUs can process that many instructions because, first, some instructions take more than one clock cycle to complete; and second, modern CPU's can process multiple instructions in parallel on a single core too. So, sometimes CPUs can even run more instructions than what their clock speed allows them to.

There are also some CPU instructions that can take arbitrary amount of time depending on their arguments, such as block memory copy instructions. Those take O(N) time based on how large the block is.

Basically, every performance problem related to code speed comes down to how many instructions are executed and for how many times. When you optimize code, what you're trying to do is either reduce the number of instructions executed or use a faster version of an instruction. The `DivRem` function runs faster than division and remainder because it gets converted into instructions that take fewer cycles.

7.3 Start from the top

The second-best way to reduce the number of instructions executed is to choose a faster algorithm. The best way is obviously to delete the code entirely, and I'm serious: delete the code you don't need. Don't keep unneeded code in the codebase. Even if it doesn't degrade the performance of the code, it degrades the performance of developers, which eventually degrades the performance of the code too. Don't even keep commented out code. Use the history features of your favorite source control system like Git or Mercurial to restore old code. If you need the feature occasionally, put it behind configuration instead of commenting it out. This way, you won't get surprised when you finally blow the dust on the code and it won't compile at all because everything's changed since. It'll remain current and working.

As we've seen in chapter two, a faster algorithm can make a tremendous difference, even if it's implemented in a poorly optimized way. So, ask yourself "is this the best way to do this?" first. There are ways to make a badly implemented code faster too, but nothing beats solving the problem at the *top* as in the broadest scope, the scenario itself, and delve deeper until you figure out the actual location of the problem. This way is usually faster and the result ends up way more maintainable.

Consider an example where users complain that viewing their profile on the app is slow and you can reproduce the problem yourself too. The performance problem can come from either the client or the server. So, you start from the top: you first identify which major layer the problem appears in by eliminating one of the two layers the problem can possibly be in. If a direct API call has the same problem, the problem must be in the client application, or otherwise in the server. You continue this path until you identify the actual problem. In a sense, you're doing a binary search as shown in figure 7.3.

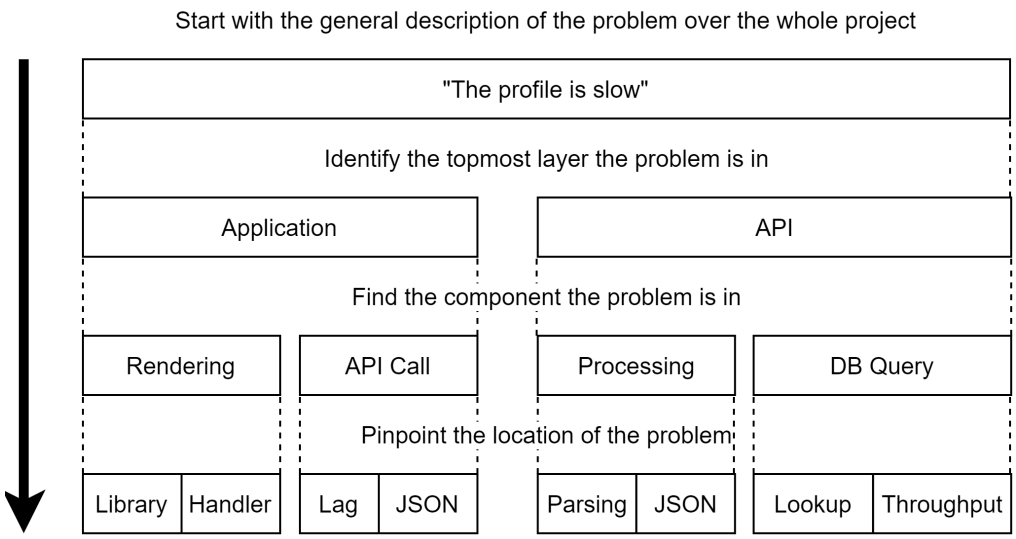


Figure 7.3 Top-down approach for identifying the root cause

When you follow a top-down approach, you're guaranteed to find the root cause of the problem in an efficient way, instead of trying guesswork. Since you do a binary search manually here, you're now using algorithms in real life too to make your life easier, so good job! When you find where the problem happens, check any red flags for immediate code complexity. There are patterns that you can identify that might be causing more complex code to execute when a simple one could suffice. Let's go over some of them.

7.3.1 Nested loops

One of the easiest ways to slow down the code is to put it inside another loop. When writing code in nested loops, we underestimate the effects of multiplication. Nested loops aren't always visible either. Continuing our example about the slow user profile, consider that you found the problem in the backend code that generates the profile data. There is a function that returns the badges a user has and shows them on their profile. A sample code might look like this:

```
public IEnumerable<string> GetBadgeNames() {
  var badges = db.GetBadges();
  foreach (var badge in badges) {
    if (badge.IsVisible) {
      yield return badge.Name;
    }
  }
}
```

No apparent nested loops. As a matter of fact, it's possible to write the same function with LINQ without any loops at all, but with the same slowness problem:

```
public IEnumerable<string> GetBadgesNames() {
  var badges = db.GetBadges();
  return badges
    .Where(b => b.IsVisible)
    .Select(b => b.Name);
}
```

Where is the inner loop? That's something you'll have to ask yourself over the course of your programming career. The culprit is the `IsVisible` property because we just don't know what it's doing underneath.

Properties in C# were invented because the developers of the language were tired of writing "get" in front of every function name despite how simple they might be. As a matter of fact, property code is converted to functions when compiled, with "`get_`" and "`set_`" prefixes added to their name. The upside of using properties is that they allow you to change how a field-looking member in a class functions without breaking compatibility. The downside of properties is that they conceal potential complexity. They look like simple fields, basic memory access operations, which might make you assume that calling a property may not be expensive at all. You should never put computationally intensive code inside properties ideally, but it's impossible for you to know whether someone else has done it or not, at least not without looking.

When we look at the source of the `IsVisible` property of the `Badge` class we can see it's more expensive than it looks:

```
public bool IsVisible {
  get {
    var visibleBadgeNames = db.GetVisibleBadgeNames();
    foreach (var name in visibleBadgeNames) {
      if (this.Name == name) {
        return true;
      }
    }
    return false;
  }
}
```

This property, without any shame, dares to call the database to retrieve the list of visible badge names and compares them in a loop to check if our supposed badge is one of the visible ones. There are too many sins in that code to explain, but your first lesson is to beware of properties. They contain logic, and their logic may not be always simple.

There are many optimization opportunities in the `IsVisible` property; the first and foremost one is not to retrieve the list of visible badge names every time the property is called. You could keep them in a static list retrieved only once, assuming the list rarely changes, and you can afford a restart when it happens. You can also do caching but we'll come to that later. That way, you could reduce the property code to this:

```
private static List<string> visibleBadgeNames = getVisibleBadgeNames();

public bool IsVisible {
  get {
    foreach (var name in visibleBadgeNames) {
      if (this.Name == name) {
        return true;
      }
    }
    return false;
  }
}
```

The good thing about keeping a list is that it already has a `Contains` method so you can eliminate the loop in IsVisible:

```
public bool IsVisible {
  get => visibleBadgeNames.Contains(this.Name);
}
```

The inner loop has finally disappeared, but we still haven't destroyed its spirit. We need to salt and burn its bones. Lists in C# are essentially arrays, and they have $O(N)$ lookup complexity. That means our loop hasn't gone away, but has only moved inside another function, in this case, `List<T>.Contains()`. So, we can't just reduce complexity by eliminating the loop — we have to change our lookup algorithm too.

We can sort the list and do a binary search to reduce lookup performance to $O(\log N)$, but luckily, we've been over chapter two and we know how the `HashSet<T>` data structure can provide a much better $O(1)$ lookup performance, thanks to looking up an item's location using its hash. Our property code finally started to look sane:

```
private static HashSet<string> visibleBadgeNames = getVisibleBadgeNames();
```

```
public bool IsVisible {
  get => visibleBadgeNames.Contains(this.Name);
}
```

We haven't done any benchmarking on this code, but looking at computational complexity pain points can provide you good insight, as you can see in that example. You should still always test if your fix performs better, because code will always contain surprises, and dark corners that might surprise you.

The story of `GetBadgeNames()` method doesn't end here. There are other questions to ask, like why the developer keeps a separate list of visible badge names instead of a single bit flag in the Badge record on the database, or why they don't simply keep them in a separate table and join them while querying the database. But as far as nested loops are concerned, it has probably become orders of magnitude faster now.

7.3.2 String-oriented programming

Strings are extremely practical. They are readable, they can hold any kind of text, and they can be manipulated easily too. We have already discussed how using the right type can improve performance over strings, but there are more subtle ways that strings can seep into your code.

One of those ways is to assume every collection is a string collection. For example, if you want to keep a flag in `HttpContext.Items` or `ViewData` container, it's common to find someone writing something like:

```
HttpContext.Items["Bozo"] = "true";
```

You find them later checking the same flag like this:

```
if ((string)HttpContext.Items["Bozo"] == "true") {
…
}
```

The typecast to string is usually added after the compiler warns you with the message, "hey, are you sure you want to do this, this isn't a string collection?" But the whole picture that the collection is actually an object collection is usually missed. You could, in fact, fix the code by simply using a Boolean variable instead.

```
HttpContext.Items["Bozo"] = true;
```

Check the value with:

```
if ((bool?)HttpContext.Items["Bozo"] == true) {
…
}
```

This way, you avoid storage overhead, parsing overhead, and even occasional typos like typing `True` instead of `true`.

The actual overhead of these simple mistakes is miniscule but when they become a habit, it can accumulate significantly. It's impossible to fix the nails on a leaking ship but nailing them the right way when building it can help you stay afloat.

7.3.3 Evaluating 2b || !2b

Boolean expressions in if statements are evaluated in the order they're written. The C# compiler generates smart code for evaluation so that it avoids evaluating unnecessary cases altogether. For example, remember our awfully expensive `IsVisible` property? Consider this check:

```
if (badge.IsVisible && credits > 150_000) {
```

An expensive property gets evaluated before a simple value check. If you're calling this function mostly with values of `x` less than 150,000, `IsBadge` wouldn't be called most of the time. You can simply swap the places of expressions:

```
if (credits > 150_000 && badge.IsVisible) {
```

This way, you wouldn't be running an expensive operation unnecessarily. You can also apply this with logical OR operations (`||`). In that case, the first expression that returns `true` would prevent the rest of the expression from being evaluated. Obviously, in real life, having that kind of expensive property is rare, but I recommend sorting expressions based on operand types:

1. d
2. Fields
3. Properties
4. Method calls

Not every Boolean expression can be safely moved around the operators though. Consider this example:

```
if (badge.IsVisible && credits > 150_000 || isAdmin) {
```

You can't simply move `isAdmin` to the beginning, because it would change the evaluation. So, make sure you don't accidentally break the logic in the if statement while optimizing Boolean evaluation.

7.4 Breaking the bottle at the neck

There are three types of delays in software: CPU, I/O, and human. You can optimize each category by either finding a faster alternative, parallelizing the tasks, or removing them from the equation.

When you're sure you're using an algorithm or a method suitable for the job, it finally comes down to how you can optimize the code itself. To evaluate your options for optimizations, you need to be aware of the luxuries that CPUs provide you.

7.4.1 Don't pack data

Reading from a memory address, say 1023, can take more time to read from memory address 1024 because CPUs can incur a penalty when reading from unaligned memory addresses. Alignment in that sense means a memory location being on the multiples of 4, 8,

16, and so forth, at least the *word size* of the CPU as seen in figure 7.4. On some older processors, the penalty for accessing unaligned memory is death by a thousand small electric shocks. Seriously, some CPUs don't let you access unaligned memory at all.

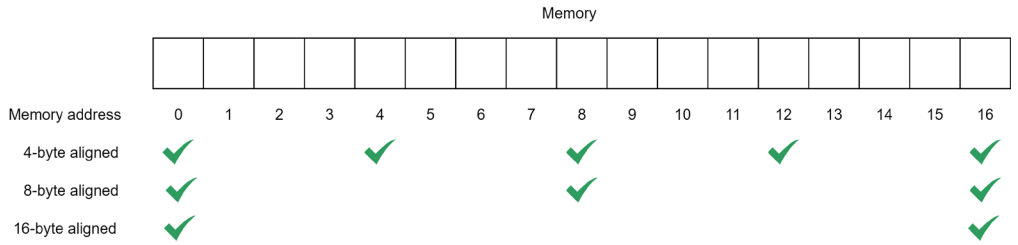


Figure 7.4 Memory address alignment

CPU Word size

Word size is typically defined by how many bits of data the CPU can process at a time. The concept is closely related to how a CPU is called a 32-bit or 64-bit. Word size mostly reflects the size of a CPU's accumulator register. Registers are like CPU-level variables, and the accumulator is the most commonly used register. Take the Z80 CPU for example. It has 16-bit registers, and it can address 16-bit memory, but it's considered an 8-bit processor because it has an 8-bit accumulator register.

Thankfully, we have compilers, and they take care of this stuff. But it's possible to override the behavior of the compiler, and it might incorrectly register this way: You're storing more stuff in a small space, there is less memory to read, so it should be faster. Consider the data structure in listing 7.4. Because it's a struct, C# will apply alignment only based on some heuristics, and that can mean no alignment at all. You might be tempted to keep the values in bytes so that it becomes a small packet to pass around as in listing 7.3.

Listing 7.3 A packed data structure

```
struct UserPreferences {
  public byte ItemsPerPage;
  public byte NumberOfItemsOnTheHomepage;
  public byte NumberOfAdClicksICanStomach;
  public byte MaxNumberOfTrollsInADay;
  public byte NumberOfCookiesIAmWillingToAccept;
  public byte NumberOfSpamEmailILoveToGetPerDay;
}
```

But, since memory accesses to unaligned boundaries are slower, your storage savings are offset by the access penalty to each member in the struct. If you change the data types in the struct from byte to int and create a benchmark to test the difference, you can see that byte access is almost twice as slow despite that it occupies a quarter of memory as shown in table 7.2.

Table 7.2 Difference between aligned and unaligned member access

Method	Mean
ByteMemberAccess	0.2475 ns
IntMemberAccess	0.1359 ns

The moral of the story is, avoid optimizing storage unnecessarily. There are benefits of doing it in certain cases: For example, when you want to create an array of a billion numbers, the difference between byte and int can become three gigabytes. Smaller sizes can also be preferable for I/O, but trust the memory alignment otherwise. The immutable law of benchmarking is, "measure twice, cut once, then measure again, and you know what, let's go easy on the cutting for a while."

SHOP LOCAL

Caching is about keeping frequently used data at a location that can be accessed faster than where it usually resides. CPUs have their own cache memories with different speeds but all faster than RAM itself. I won't go into the technical details of how cache is structured, but basically, CPUs can read memory in their cache much faster than regular memory in RAM. That means, for example, sequential reads are faster than random reads around the memory. For example, reading an array sequentially can be faster than reading a linked list sequentially, although both take O(N) time to read end-to-end and arrays can perform better than linked lists. The reason is that there is a higher chance of the next element being in the cached area of the memory. Linked lists, on the other hand, get their elements scattered around in the memory because they are separately allocated.

Suppose you have a CPU with a 16-byte cache, and you have both an array of three integers and a linked list of three integers. In figure 7.5, you can see that reading the first element of the array would also trigger loading the rest of the elements into the CPU cache, while traversing the linked list would cause a cache miss and force the new region to be loaded into the cache.

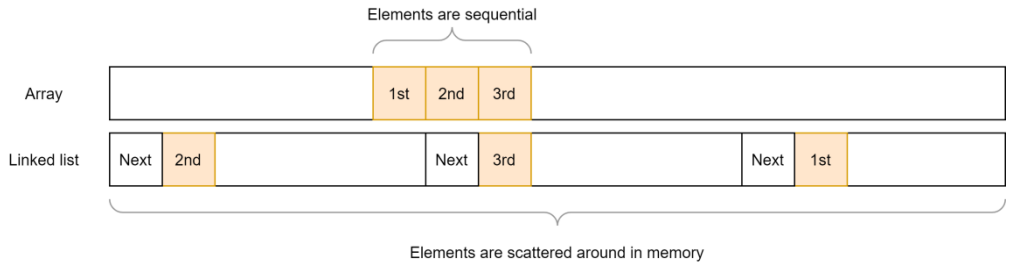


Figure 7.5 Array vs linked list cache locality

CPUs usually bet that you'd be reading data sequentially. That doesn't mean linked lists don't have their uses. They have excellent insert/delete performance and less memory overhead when growing. Array-based lists need to reallocate and copy buffers when growing which is

terribly slow, so they allocate more than they need which can cause disproportionate memory use in large lists. In most cases, though, a list would serve you fine and it might as well be faster for reading.

7.4.2 Keep dependent work away from each other

A single CPU instruction is processed by discrete units on the processor. For example, one unit is responsible for decoding the instruction while another is responsible for memory access. But, since a decoder unit needs to wait for an instruction to complete, it can do other decoding work for the next instruction while memory access is running. That technique is called *pipelining.* That means the CPU can execute multiple instructions in parallel on a single core as long as the next instruction doesn't depend on the result of the previous one.

Consider an example where you need to calculate a checksum where you simply add together the values of a byte array to get a result, as in listing 7.4. Normally checksums are used for error detection, and adding numbers together can be the worst implementation, but we'll assume that it was a government contract. When you look at the code, it constantly updates the value of `result`, therefore every calculation depends on `i` and `result`. That means CPU cannot parallelize any work, because it depends on an operation.

Listing 7.4 A simple checksum

```
public int CalculateChecksum(byte[] array) {
  int result = 0;
  for (int i = 0; i < array.Length; i++) {
    result = result + array[i];    #A
  }
  return result;
}
```

#A Depends on both `i` and the previous value of `result`.

There are ways to reduce dependencies or at least make them block the instruction flow less than they do. One way is to reorder instructions to increase the gap between dependent code, so an instruction doesn't block the following one in the pipeline due to the dependency on the result of the first operation.

Since addition can be done in any order, we can split the addition into four parts in the same code and let the CPU parallelize the work. See how we can implement such a task in listing 7.5. This code contains more instructions, but there are now four different result accumulators that can complete the checksum separately and later be summed together. We later sum the remaining bytes in a separate loop.

Listing 7.5 Parallelizing work on a single core

```
public static int CalculateChecksumParallel(byte[] array) {
  int r0 = 0, r1 = 0, r2 = 0, r3 = 0;    #A
  int len = array.Length;
  int i = 0;
  for (; i < len - 4; i += 4) {
    r0 += array[i + 0];    #B
    r1 += array[i + 1];    #B
    r2 += array[i + 2];    #B
```

```
    r3 += array[i + 3];    #B
  }
  int remainingSum = 0;
  for (; i < len; i++) {    #C
    remainingSum += i;
  }
  return r0 + r1 + r2 + r3 + remainingSum;    #D
}
```

#A The four accumulators!
#B These calculations are independent of each other.
#C Calculate the sum of the remaining bytes.
#D Bring everything together.

We're doing a lot more work than in the simpler code in listing 7.4, yet this one turns out be 15% faster on my machine. Don't expect magic from such a micro-optimization, but you'll love it when it helps you in tackling CPU intensive code. The main takeaway is that reordering code — and even removing dependencies between code — can help your code's speed because dependent code can cause the pipeline to get clogged.

7.4.3 Be predictable

The most upvoted, the most popular question in the history of StackOverflow is "Why is processing a sorted array faster than processing an unsorted array?"[4]

In order to optimize execution time, CPUs try to act preemptively ahead of the running code in order to make preparations before there is a need. One of the techniques CPUs employ is called *branch prediction*. Such code is just a sugarcoated version of comparisons and branches:

```
if (x == 5) {
  Console.WriteLine("X is five!");
} else {
  Console.WriteLine("X is something else");
}
```

The if statement and curly braces are the elements of structured programming. They are a sugarcoated version of what the CPU processes. Behind the scenes, the code gets converted into a low-level code like this during the compilation phase:

```
  compare x with 5
  branch to ELSE if not equal
  write "X is five"
  branch to SKIP_ELSE
ELSE:
  write "X is something else"
SKIP_ELSE:
```

I'm just paraphrasing here as the actual machine code is more cryptic, but this isn't entirely inaccurate. Regardless how elegant the design you create with your code, it eventually becomes a bunch of comparison, addition, and branch operations. Listing 7.6 shows the

[4] The StackOverflow question can be found at https://stackoverflow.com/questions/11227809/why-is-processing-a-sorted-array-faster-than-processing-an-unsorted-array

actual assembly output for x86 architecture for the same code. It might feel more familiar after you've seen the pseudocode. There is an excellent online tool at sharplab.io that lets you see the assembly output of your C# program; hopefully it will outlive this book's lifetime.

Stop worrying, and learn to love assembly

Machine code, the native language of a CPU, is just a series of numbers. Assembly is a human readable syntax for machine code. Assembly syntax differs between CPU architectures, so I recommend getting familiar with at least one. It's a humbling experience, and it reduces your scare about what's going on under the hood. It may seem complicated, but it's simpler than the languages we write programs in, even primitive if you will. An assembly listing is a series of labels and instructions like:

```
    let a, 42
some_label:
    decrement a
    compare a, 0
    jump_if_not_equal some_label
```

That's a basic decrementing loop counting from 42 to zero written in a pseudo-assembly syntax. In real assembly, instructions are shorter to make them easier to write and excruciating to read. For example, the same loop would be written like this on an x86 CPU:

```
   mov al, 42
some_label:
    dec al
    cmp al, 0
    jne some_label
```

On an ARM processor architecture, it can look like this instead:

```
    mov r0, #42
some_label:
    sub r0, r0, #1
    cmp r0, #0
    bne some_label
```

These can be written shorter than shown here with different instructions, but as long as you're familiar with the structure of assembly, you can take a peek at what kind of machine code JIT compiler generates and understand the actual behavior. It especially does wonders when understanding CPU intensive tasks.

Listing 7.6 Actual assembly code for our comparison

```
        cmp ecx, 5    #A
        jne ELSE    #B
```

```
        mov ecx, [0xf59d8cc]    #C
        call System.Console.WriteLine(System.String)
        ret    #E
ELSE:   mov ecx, [0xf59d8d0]    #D
        call System.Console.WriteLine(System.String)
        ret    #E
```

#A Compare instruction
#B Branching instruction (Jump if Not Equal)
#C Pointer for the string "X is five"
#D Pointer for the string "X is something else"
#E "Return" instruction

A CPU cannot know if a comparison will be successful or not before it's executed, but thanks to branch prediction, it can have a strong guess based on what it observes. Based on its guesses, the CPU makes a bet and starts processing instructions from that branch it predicted, and if it's successful in its prediction, everything becomes already in place, boosting the performance.

That's why processing an array with random values can be slower if it involves comparisons of values: branch prediction fails spectacularly in that case. A sorted array performs better because the CPU can predict the ordering properly and predict the branches correctly.

Keep that in mind when processing data. The less surprises you give the CPU, the better it will perform.

7.4.4 SIMD

CPUs also support specialized instructions that can perform computation on multiple data at the same time with a single instruction. That technique is called SIMD (Single Instruction, Multiple Data). If you want to perform the same calculation on multiple variables, SIMD can boost its performance significantly on supported architectures.

SIMD works pretty much like multiple pens taped together. You can draw whatever you want, but the pens will all perform the same operation on a different coordinate on the paper. A SIMD instruction will perform an arithmetic computation on multiple values, but the operation will remain constant.

C# provides SIMD functionality via `Vector` types in System.Numerics namespace. Since every CPU's SIMD support is different, and some CPUs don't support SIMD at all, you first check whether it's available on the CPU:

```
if (!Vector.IsHardwareAccelerated) {
  … non-vector implementation here …
}
```

Then, you need to figure out how many of a given type the CPU can process at the same time. That changes from processor to processor, so you have to query it first:

```
int chunkSize = Vector<int>.Count;
```

In this case, we're looking to process `int` values. The number of items the CPU can process can change based on the data type. When you know the number of elements you process at a time, you can go ahead and process the buffer in chunks.

Consider that we'd like to multiply values in an array. Multiplication of a series of values is a common problem in data processing, whether it's changing the volume of a sound recording or adjusting the brightness of an image. For example, if you multiply pixel values in an image by two, it becomes twice as bright. Similarly, if you multiply voice data by two, it becomes twice as loud. A naïve implementation would look as shown in listing 7.7. We simply iterate over the items and replace the value in place with the result of the multiplication.

Listing 7.7 Classic in-place multiplication

```
public static void MultiplyEachClassic(int[] buffer, int value) {
  for (int n = 0; n < buffer.Length; n++) {
    buffer[n] *= value;
  }
}
```

When we use `Vector` type to make these calculations instead, our code becomes more complicated, and it looks slower, to be honest. You can see the code in listing 7.8. We basically check for SIMD support and query the chunk size for integer values. We later go over the buffer at the given chunk size and copy the values into vector registers by creating instances of `Vector<T>`. That type supports standard arithmetic operators, so we simply multiply the vector type with the number given. It will automatically multiply all of the elements in the chunk in a single go.

Listing 7.8 "We're not in Kansas anymore" multiplication

```
public static void MultiplyEachSIMD(int[] buffer, int value) {
  if (!Vector.IsHardwareAccelerated) {
    MultiplyEachClassic(buffer, value);    #A
  }

  int chunkSize = Vector<int>.Count;    #B
  int n = 0;
  for (; n < buffer.Length - chunkSize; n += chunkSize) {
    var vector = new Vector<int>(buffer, n);    #C
    vector *= value;    #D
    vector.CopyTo(buffer, n);    #E
  }

  for (; n < buffer.Length; n++) {    #F
    buffer[n] *= value;    #F
  }    #F
}
```

#A Call the classic implementation if SIMD's not supported.
#B Query how many values SIMD can process at once.
#C Copy array segment into SIMD registers.
#D Multiply all values at once.
#E Replace the results.
#F Process remaining bytes the classic way.

It looks like too much work, doesn't it? Yet, benchmarks are impressive as seen in table 7.3. Our SIMD-based code is twice as fast as the regular code in this case. Based on the data types you process and operations you perform on the data, it can be much higher.

Table 7.3 The SIMD difference

Method	Mean
MultiplyEachClassic	5.641 ms
MultiplyEachSIMD	2.648 ms

You can consider SIMD when you have a computationally intensive task at hand and you need to perform the same operation on multiple elements at the same time.

7.5 Ones and zeros of I/O

I/O encompasses everything a CPU communicates with the peripheral hardware, be it disk, network adapter, or even GPU. I/O is usually the slowest link on the performance chain. Think about it: a hard drive is actually a rotating disk with a spindle seeking over the data. It's basically a robotic arm constantly moving around. A network packet can travel at the speed of light, yet it would still take more than a hundred milliseconds for it to rotate the earth. Printers are especially designed to be slow, inefficient, and anger-inducing.

You can't make I/O itself faster most of the time as its slowness comes from physics, but the hardware can run independently of CPU, so, it can work while CPU is doing other stuff. That means you can overlap CPU and I/O work and complete an overall operation in a smaller timeframe.

7.5.1 Make I/O faster

Yes, I/O is slow due to inherent limitations of hardware, but it can be made faster too. For example, every read from a disk incurs an operating system call overhead. Consider a file copy code as in listing 7.9. It's pretty much straightforward. It copies every byte read from the source file and writes those bytes to the destination file.

Listing 7.9 Simple file copy

```
public static void Copy(string sourceFileName,
  string destinationFileName) {

  using var inputStream = File.OpenRead(sourceFileName);
  using var outputStream = File.Create(destinationFileName);
  while (true) {
    int b = inputStream.ReadByte();    #A
    if (b < 0) {
      break;
    }
    outputStream.WriteByte((byte)b);    #B
  }
}
```

#A Read the byte.
#B Write the byte.

The problem is that every system call implies an elaborate ceremony. The `ReadByte()` function here calls the operating system's read function. Operating system calls switch to kernel mode. That means CPU changes its execution mode. Operating system routine looks up the file handle and necessary data structures. It checks whether the I/O result is already in cache. If it's not, it calls relevant device drivers to perform actual I/O operation on the disk. The read portion of the memory gets copied to a buffer in the process' address space. These operations happen lightning fast, and it can become significant when you just read one byte.

Many I/O devices read/write in blocks. Those are called block devices. Network and storage devices are usually block devices. The keyboard is a character device as it sends a character at a time. Block devices cannot read less than the size of a block, so it doesn't make sense to read anything less than a typical block size. For example, a hard drive can have a sector size of 512 bytes, making it a typical block size for disks. Modern disks can have larger block sizes but let's see how much performance can be improved simply by reading 512 bytes. Listing 7.10 shows the same copy operation that takes a buffer size as a parameter and reads and writes using that chunk size.

Listing 7.10 File copy using larger buffers

```
public static void CopyBuffered(string sourceFileName,
  string destinationFileName, int bufferSize) {

  using var inputStream = File.OpenRead(sourceFileName);
  using var outputStream = File.Create(destinationFileName);
  var buffer = new byte[bufferSize];
  while (true) {
    int readBytes = inputStream.Read(buffer, 0, bufferSize);    #A
    if (readBytes == 0) {
      break;
    }
    outputStream.Write(buffer, 0, readBytes);    #B
  }
}
```

#A Read `bufferSize` **bytes at once.**
#B Write `bufferSize` **bytes at once.**

If we write a quick benchmark that tests against byte-based copy function and buffered variant with different buffer sizes, we can see the difference that reading large chunks at a time makes. You can see the results in table 7.4.

Table 7.4 Effect of buffer size on I/O performance

Method	Buffer size	Mean
Copy	1	1,351.27 ms
CopyBuffered	512	217.80 ms
CopyBuffered	1024	214.93 ms

CopyBuffered	16384	84.53 ms
CopyBuffered	262144	45.56 ms
CopyBuffered	1048576	43.81 ms
CopyBuffered	2097152	44.10 ms

Even using a 512 byte buffer makes a tremendous difference — the copy operation becomes six times faster. Yet, increasing it to 256KB makes the most difference and making anything above that is marginal. I ran these benchmarks on a Windows machine, and Windows I/O uses 256KB as the default buffer size for its I/O operations and cache management. That's why the returns suddenly become marginal after 256KB. In the same way that it says "actual contents may vary" on food packages, your actual experience on your operating system may vary.

Consider finding the ideal buffer size when working with I/O and avoid allocating more memory than you need.

7.5.2 Make I/O non-blocking

One of the most misunderstood concepts in programming is asynchronous I/O. It's often confused with multi-threading. Multi-threading is a parallelization model to make any kind of operation faster by letting a task run on separate cores. Asynchronous I/O (or async I/O for short) is a parallelization model for I/O-heavy operations only and it can work on a single core. Multi-threading and async I/O can also be used together as they address different use cases.

I/O is naturally asynchronous because external hardware is almost always slower than the CPU, and the CPU doesn't like waiting and doing nothing. Mechanisms like interrupts and DMA (Direct Memory Access) were invented to allow hardware to signal the CPU when an I/O operation is complete, so CPU can transfer the results. That means, when an I/O operation is issued to the hardware, CPU can continue executing other stuff while hardware is doing its work, and the CPU can check back when the I/O operation is complete. This mechanism is the foundation of async I/O.

Figure 7.6 gives an idea about how both types of parallelization work. In both figures, the second computational code (CPU Op #2) is dependent on the result of the first I/O code (I/O Op #1). Because computational code cannot be parallelized on the same thread, they execute in tandem, and therefore take longer than multi-threading on a four core machine. On the other hand, however, you still gain significant parallelization benefits without consuming threads or occupying cores.

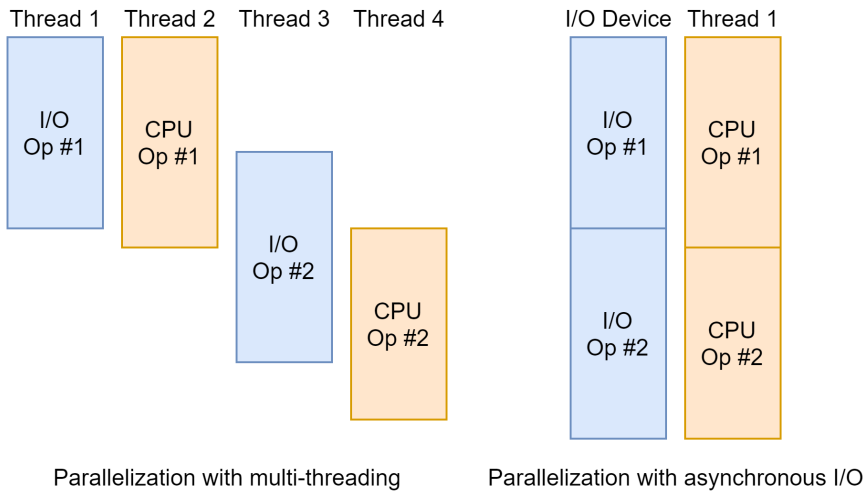


Figure 7.6 Difference between multi-threading and async I/O

The performance benefit of async I/O comes from its providing natural parallelization to the code without your doing extra work for it. You don't even need to create an extra thread. It's possible to run multiple I/O operations in parallel and collect the results without suffering through the problems multi-threading brings, like race conditions. It's practical and scalable.

Asynchronous code can also help with responsiveness in event-driven mechanisms, especially user interfaces, without consuming threads. It might seem like UI has nothing to do with I/O but user input also comes from I/O devices like a touchscreen, a keyboard, or a mouse, and user interfaces are triggered by those events. They constitute perfect candidates for async I/O, and asynchronous programming in general. Even timer-based animations are hardware driven because of how a timer on a device operates, and are therefore ideal candidates for async I/O.

THE ARCHAIC WAYS

Until the early 2010s, async I/O used to be managed with callback functions. Async operating system functions would require you to pass them a callback function and OS would execute your callback function when the I/O operation was completed. Meanwhile, you could perform other tasks. If we wrote our file copy operation in old asynchronous semantics, it would look pretty much as shown in listing 7.11 Mind you, this is a very cryptic and ugly code and it's probably why boomers don't like async I/O so much. Actually, I had so much trouble writing this code myself that I had to resort to some modern constructs like `Task` in order to finish it. I'm just showing you this so that you'll love and appreciate the modern constructs and how much time they save us.

The most interesting thing about this ancient code is that it returns immediately, which is magical. That means I/O is working in the background, the operation continues, and you can do other work while it's being processed. You're still on the same thread too. No multi-

threading involved. In fact, that's one of the greatest advantages of async I/O, as it conserves OS threads, so it becomes more scalable, which we will see in the chapter about scalability. If you don't have anything else to do, you can always wait for it to complete but that's only a preference.

In listing 7.11, we define two handler functions. One is an asynchronous `Task` called `onComplete()`, which we want to run when the whole execution finishes, but not right away. Another is a local function called `onRead()` which is called every time a read operation is complete. We pass this handler to `stream`'s `BeginRead` function, so it initiates an asynchronous I/O operation and registers `onRead` as a callback to be called when the block is read. In the `onRead` handler, we start the write operation of the buffer we just read completely and make sure another round of read is called with the same `onRead` handler set as a callback. This goes on until the code reaches the end of the file, and that's when the `onComplete` Task gets started. It's a very convoluted way to express asynchronous operations.

Listing 7.11 Old-style file copy code using async I/O

```
public static Task CopyAsyncOld(string sourceFilename,
  string destinationFilename, int bufferSize) {

  var inputStream = File.OpenRead(sourceFilename);
  var outputStream = File.Create(destinationFilename);

  var buffer = new byte[bufferSize];
  var onComplete = new Task(() => {    #A
    inputStream.Dispose();
    outputStream.Dispose();
  });

  void onRead(IAsyncResult readResult) {    #B
    int bytesRead = inputStream.EndRead(readResult);    #D
    if (bytesRead == 0) {
      onComplete.Start();    #C
      return;
    }
    outputStream.BeginWrite(buffer, 0, bytesRead,    #E
      writeResult => {
        outputStream.EndWrite(writeResult);    #F
        inputStream.BeginRead(buffer, 0, bufferSize, onRead,    #G
          null);
      }, null);
  }

  var result = inputStream.BeginRead(buffer, 0, bufferSize,    #H
    onRead, null);
  return Task.WhenAll(onComplete);    #J
}
```

#A Called when the function finishes.
#B Called whenever a read operation is complete.
#C Start the final Task.
#D Get the number of bytes read.
#E Start the write operation.

#F Acknowledge completion of the write.
#G Start the next read operation.
#H Start the first read operation.
#J Return a waitable Task for onComplete.

The problem with this approach is that the more async operations you start, the easier you lose track of the operations. Things would easily turn into *callback hell*, a term coined by NodeJS developers.

MODERN ASYNC/AWAIT

Luckily, brilliant designers at Microsoft found a great way to write async I/O code using async/await semantics. The mechanism, first introduced in C#, became so popular and proved itself so practical that it got adopted by many other popular programming languages such as C++, Rust, JavaScript, and Python.

You can see the async/await version of the same code in listing 7.12. What a breath of fresh air! We declare the function with the `async` keyword so that we can use `await` in the function. Await statements define an anchor but they don't really wait for the expression following them to be executed. They just signify the points of return when the awaited I/O operation completes in the future, so we don't have to define a new callback for every continuation. We can write code like regular synchronous code. Because of that, the function still returns immediately as in listing 7.11. Both `ReadAsync` and `WriteAsync` functions are also functions that return a `Task` object like `CopyAsync` itself. By the way, the Stream class already has a `CopyToAsync` function to make copying scenarios easier but we're keeping read and write operations separate here to align the source with the original code.

Listing 7.12 Modern async I/O file copy code

```
public async static Task CopyAsync(string sourceFilename,    #A
  string destinationFilename, int bufferSize) {

  using var inputStream = File.OpenRead(sourceFilename);
  using var outputStream = File.Create(destinationFilename);
  var buffer = new byte[bufferSize];
  while (true) {
    int readBytes = await inputStream.ReadAsync(buffer, 0, bufferSize);    #B
    if (readBytes == 0) {
      break;
    }
    await outputStream.WriteAsync(buffer, 0, readBytes);    #B
  }
}
```

#A The function is declared with the async keyword and it returns Task.
#B Any operation following await is converted to a callback behind the scenes.

When you write code with async/await keywords, the code behind the scenes gets converted to something similar to listing 7.11 during compilation, with callbacks and everything. Async/await saves you from a lot of legwork.

GOTCHAS OF ASYNC I/O

Programming languages don't require you to use async mechanisms just for I/O. You can declare an async function without calling any I/O related operations at all and perform only CPU work on them. In that case, you'd have created an unnecessary level of complexity without any gains. The compiler usually warns you of that situation, but I've seen many examples of compiler warnings getting ignored in a corporate setting because nobody would want to deal with the fallout of any problems a fix would cause. The performance issues would pile up, and then you'd get tasked with fixing all those problems at once, dealing with a larger fallout. Bring this up in code reviews, and make your voice heard.

One rule of thumb that you need to keep in mind with async/await is that "await" doesn't "wait." If your async code waits for something to complete, you're doing it wrong.

7.6 If all else fails, cache

Caching is one of the most robust ways to improve performance immediately. Cache invalidation might be a hard problem, but it's not a problem if you only cache things that you don't worry about invalidating. You don't need an elaborate caching layer residing on a separate server like Redis, or Memcached either. You can use an in memory cache like the one provided by Microsoft in the `MemoryCache` class in the `System.Runtime.Caching` package. True, it cannot scale beyond a certain point, but scaling may not be something you'd be looking for at the beginning of a project. Ekşi Sözlük serves 10 million requests per day, on a single DB server, and on four web servers, yet it still uses in-memory cache today.

Avoid using data structures that are not designed for caching. They usually don't have any eviction or expiration mechanism, becoming the source of memory leaks and, eventually, crashes. Use things designed to be cached.

Your database can also be a great persistent cache.

Don't be afraid of infinite expiration in cache, as either a cache eviction or an application restart will come before the end of the universe.

7.7 Summary

- Use premature optimizations as exercises, and learn from them.
- Avoid painting yourself into a corner with unnecessary optimizations.
- Always validate your optimizations with benchmarking.
- Keep optimization and responsiveness in balance.
- Make a habit of identifying problematic code like nested loops, string-heavy code, inefficient Boolean expressions.
- When building data structures, consider the benefits of memory alignment to get better performance.
- When you need micro-optimizations, know how a CPU behaves, and have cache locality, pipelining, and SIMD in your toolbelt.
- Increase I/O performance by using correct buffering mechanisms.
- Use asynchronous programming to run code and I/O operations in parallel without wasting threads.
- In case of emergency, break the cache.

8

Palatable scalability

This chapter covers:

- Scalability vs performance
- Progressive scalability
- Breaking database rules
- Smoother parallelization
- The truth in monolith

> "It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness."
>
> **Charles Dickens on scalability**

I've had my share of experience with scalability because of the technical decisions I've made for Ekşi Sözlük back in 1999. The whole database for the web site was just a single text file at first. The writes held locks on the text file, causing the everything to freeze for all visitors. Reads weren't very efficient either; retrieving a single record would be in $O(N)$ time, requiring scanning the whole database. It was the worst of the worst technical designs possible.

It wasn't because the server was so slow that the code froze. It was the data structures and the parallelization decisions that all contributed to the sluggishness. That's the gist of scalability itself. Performance by itself can't make a system scalable. You need all aspects of your design cater to increasing number of users.

More importantly, that terrible design wasn't more important than how quickly I released the web site, in hours. Initial technical decisions didn't matter in the long run because I was able to pay back most of the technical debts on the way. I changed the database technology as soon as it started causing too many problems. I wrote the code from scratch when the

technology I used didn't work out anymore. There is a Turkish proverb that goes like "a caravan is prepared on the road", which means "make it up as you go."

I also recommended measuring twice and cutting once at several places in the book which seemingly conflict with *que sera, sera*[1] motto. That's because there is no single prescription for all our problems. We need to keep these methods in our toolbelts and apply the right one for the problem at hand.

From a systems perspective, scalability means ability to make a system faster by throwing more hardware at it. From programming perspective, a scalable code is the one that can keep its responsiveness constant at the face of increasing demand. There is obviously an upper limit of how some code can keep up with the load, and the goal of writing scalable code is to push that upper limit as far as possible.

Like refactoring, scalability is best addressed progressively: in tangible, smaller steps towards a bigger goal. It's possible to design a system to be fully scalable from scratch, but the amount of effort and time required to achieve it and the returns you get are overshadowed by the importance of getting a product released as soon as possible.

Some things don't scale at all. As Fred Brooks put eloquently in his marvelous book The Mythical Man Month, "the bearing of a child takes nine months, no matter how many women are assigned." Mr. Brooks said this about how assigning more people to an already delayed project can only add to the delays, but it's also applicable to certain factors of scalability. For example, you can't make a CPU core run more instructions in a second than its clock frequency. I mean, yes, we've discussed that we can surpass it slightly by appealing to SIMD, branch prediction, and so forth but there is still an upper limit on the performance you can achieve on a single CPU core.

The first step to achieve scalable code is to remove the bad code that prevents it from scaling. Such code can create bottlenecks, causing the code to remain slow even after you've added more hardware resources. Some may even sound anti-intuitive to you. Let's go over these potential bottlenecks, and how we can remove them.

8.1 Don't use locks

Locking in programming is a feature that lets you write thread-safe code. Thread-safe means that a piece of code can work consistently even when called by two or more threads simultaneously. Consider a class that's responsible to generate unique identifiers for entities created in your application, and let's assume that it needs to generate sequential numeric identifiers. That's usually not a good idea as we've discussed in the chapter on security, as incremental identifiers can leak information about your application. You may not want to expose how many orders you receive in a day, how many users you have, and so forth. Let's assume that there's a legitimate business reason behind having consecutive identifiers, say, to ensure there are no missing items. A simple implementation would look like this:

```
class UniqueIdGenerator {
  private int value;
  public int GetNextValue() => ++value;
```

[1] After a popular song from 1950's by Doris Day, my father's favorite singer, "Que sera, sera" means "whatever will be, will be" in Italian. It's the official mantra for code deploys on Fridays, usually followed by 4 Non Blondes hit "What's Up?" on Saturday which ends with Aimee Mann's "Calling It Quits" on Monday.

```
}
```

When you have multiple threads using the same instance of this class, it's possible for two threads to receive the same value, or non-sequential values. That's because the expression `++value` translates to multiple operations on the CPU, one that reads `value`, one that increments it, one that stores the incremented value back in the field, and finally the one that returns the result as can be seen clearly in the x86 assembly output of the JIT compiler[2]:

```
UniqueIdGenerator.GetNextValue()
    mov eax, [rcx+8]    #A
    inc eax    #B
    mov [rcx+8], eax    #C
    ret    #D
```

#A Move field's value in memory into EAX register (read).
#B Increment the value of EAX register (increment).
#C Move incremented value back into the field (store).
#D Return the result in EAX register (return).

Every line is an instruction that a CPU runs one after the other. When you try to visualize multiple CPU cores running the same instructions at the same time, it's easier to see how it can cause conflicts in the class as seen in figure 8.1. There you can see that three threads return the same value, one, despite that the function was called three times.

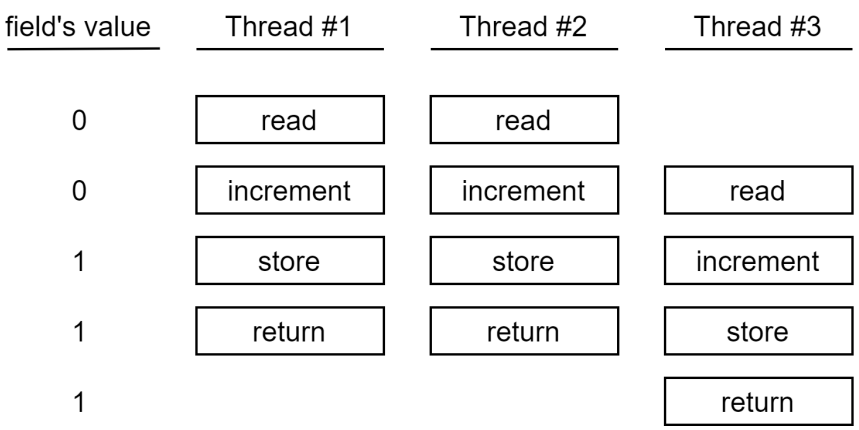


Figure 8.1 Multiple threads running simulatenously causing state to break

That code isn't thread-safe and the way all threads try to manipulate the data themselves without respecting other threads is called a *race condition*. CPUs, programming languages and operating systems provide you variety of features that can help you deal with that

[2] A JIT (Just In Time) compiler is a compiler that converts compiled code (called IL, Intermediate Language) to the native instruction set of the CPU architecture it's running on in order to make it faster.

problem. They usually all come down to blocking other CPU cores from reading from or writing to the same memory region at the same time, and that folks, is called locking.

In this example, the most optimized way would be to use an atomic increment operation that increments the value in memory location directly and prevents other CPU cores from accessing the same memory region while doing that, so no thread would read the same value, or incorrectly skip values. It would look like this:

```
using System.Threading;
class UniqueIdGeneratorAtomic {
  private int value;
  public int GetNextValue() => Interlocked.Increment(ref value);
}
```

In this case, the locking is implemented by the CPU itself and it would behave as shown in figure 8.2 when executed. The CPU's lock instruction only holds the execution on parallel cores at that location during the lifetime of the instruction that immediately follows it, so the lock automatically gets released when every atomic in-memory add operation is executed. Notice that the return instructions don't return the field's current value but the result of "memory add" operation instead, but the field's value stays sequential regardless.

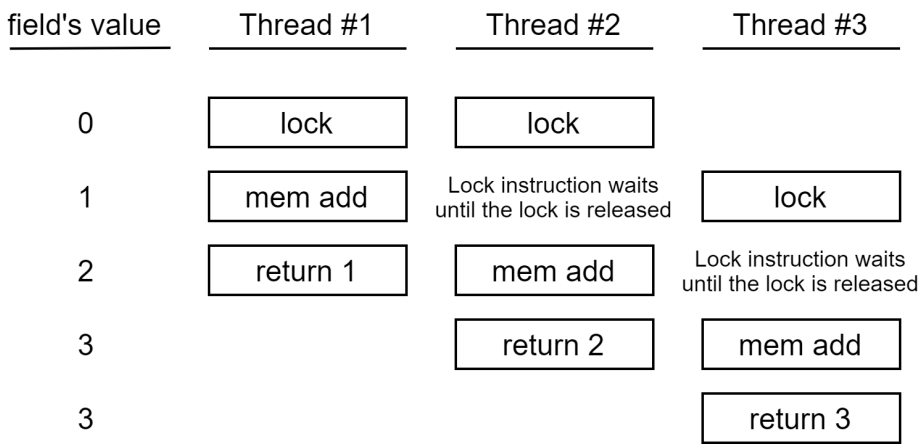


Figure 8.2 CPU cores wait for each other when atomic increment is used.

There will be many cases where a simple atomic increment operation isn't enough to make your code thread-safe. For example, what if you needed to update two different counters in sync? In cases where you can't ensure consistency with atomic operations, you can use C#'s `lock` statement as shown in listing 8.1. For simplicity, we stick to our original counter example, but locks can be used to serialize any state change on the same process. We allocate a new dummy object to use as a lock, as .NET uses an object's header to keep lock information.

Listing 8.1 Thread-safe counter with C#'s lock statement

```
class UniqueIdGeneratorLock {
  private int value;
  private object valueLock = new object();    #A
  public int GetNextValue() {
    lock (valueLock) {    #B
      return ++value;    #C
    }
  }
}
```

#A Our lock object, specific for our purpose
#B Other threads wait until we're done.
#C Exiting the scope automatically releases the lock.

Why do we allocate a new object? Couldn't we just use "`this`" so our own instance would also act like a lock? That would save us some typing. The problem is that your instance can also be locked by some code outside of your control. That can cause unnecessary delays, or even *deadlocks* because your code might be waiting on that other code.

Deadlocks go brrr

A deadlock is when two threads wait on the resources acquired by the other. It's quite easy to hit: Thread 1 acquires resource A, and waits for resource B to be released, while thread 2 acquires resource B, and waits for resource A to be released as shown in figure 8.3.

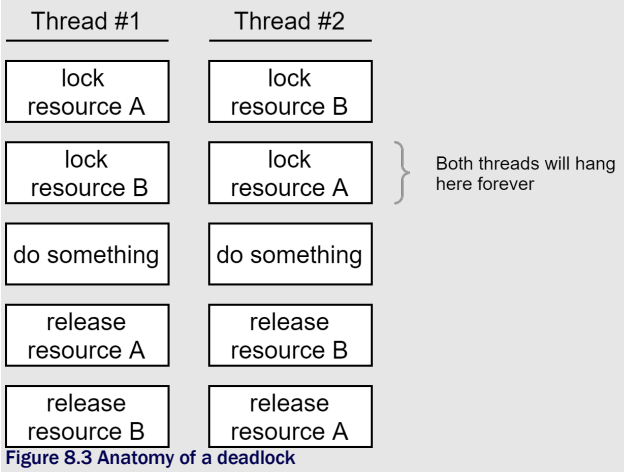


Figure 8.3 Anatomy of a deadlock

The result is like an infinite loop, waiting for a condition that will never be satisfied. That's why it's important to be explicit about which lock we use for what purpose in the code. That's why having a separate object for our locks is always a good idea so you can trace the code that uses the certain locks and make sure they're not shared by other code. That's not possible with `lock(this)`.

Some of the application hangs you encounter are results of a deadlock, and contrary to the popular belief, they can't be fixed by hitting your table with your mouse, screaming at the monitor, or rage quitting.

There is no magical solution to deadlocks other than clear understanding of the locking mechanisms in your code, but a good rule of thumb is always to first the most recently acquired lock first, and release locks as soon as possible. Some programming constructs may make it easier to avoid using locks, like channels in Go programming language, but it's still possible to have deadlocks with those too, only less likely.

Our own implemented locking code would behave like in figure 8.4. As you can see, it's not as efficient as atomic increment operation, but it's still perfectly thread-safe.

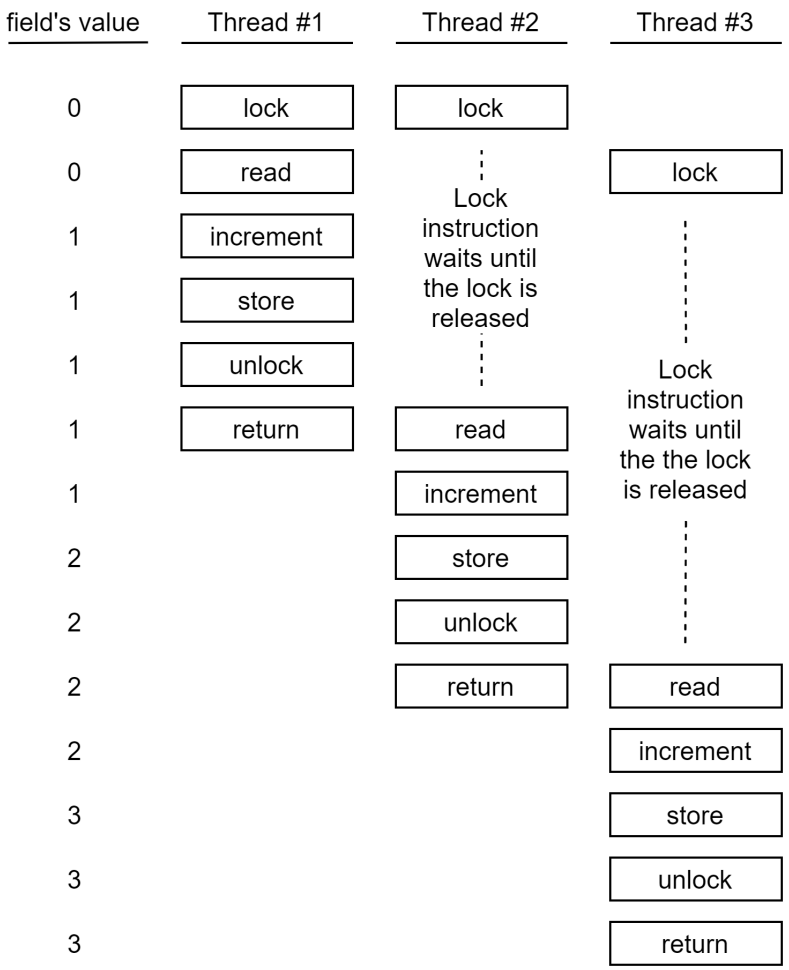


Figure 8.4 Using C#'s lock statement to avoid race conditions

As you can see, locks can make other threads stop and wait for a certain condition. While providing consistency, it can be one of the greatest challenges against scalability. There's nothing worse than wasting valuable CPU time waiting. You should strive for waiting as little as possible. How do we achieve that?

First, make sure that you really need locks. I've seen code written by smart programmers that can simply be fine without acquiring any locks at all, but unnecessarily wait for a certain condition to be met. If an object instance won't be manipulated by other threads, that means you may not need locks at all. I don't say "you won't" because it's hard to assess side effects of code. Even a locally scoped object can use shared objects, and therefore might require locks in place. You need to be clear about your intent, and the side effects of your code. Don't use locks because it magically makes the code it surrounds thread-safe. Understand how locks work, be explicit about what you're doing.

Second, find out if the shared data structure you use has a *lock-free* alternative. Lock-free data structures, as the name implies, provide data structures that can be directly accessed by multiple threads without requiring any locks. The implementation of lock-free structures can be complicated though. They can even be slower than their locked counterparts, but they can be more scalable! A common scenario where a lock-free structure can be beneficial is shared dictionaries, or as they're called in some platforms, maps. You might need a dictionary of something shared by all threads, like certain keys and values, and the usual way to handle that is to use locks.

Consider an example where you need to keep API tokens in memory, so you don't need to query the database for their validity every time they're accessed. A correct data structure for this purpose would be a cache, and cache data structures can have lock-free implementations too, but the developers tend to use the tool that is closest when they try to solve a problem, in this case, a dictionary:

```
public Dictionary<string, Token> Tokens { get; } = new();
```

Notice the cool `new()` syntax in C# 9.0? Finally, the dark days of writing the same type twice when declaring class members is over. The compiler can now assume its type based on its declaration.

Anyway, we know that dictionaries aren't thread-safe if they're manipulated. That's an important point: if you have a data structure that you initialize at the start of your application and you never change it, you don't need it to be locked; all read-only structures without *side-effects* are thread-safe.

Side-effects

What does it mean code having side-effects, apart from the occasional headache and nausea you get in a code review session? It's a term coming from the domain of functional programming. If a function changes anything outside its scope, that's considered a side-effect. Not just variables, or fields, but anything. For example, if a function writes a log message, it causes an irreversible change in the log output, that's considered a side effect too. A function without any side-effects can be run any number of times, and nothing in the environment would change. Those functions without side-effects are called *pure functions*. A function that calculates the area of a circle and returns the result is a pure function:

```
class Circle {
```

```
  public static double Area(double radius) => Math.PI * Math.Pow(radius, 2);
}
```

That's a pure function not just because it has no side effects, but also because the members and functions it accesses are also pure. Otherwise, those could cause side-effects too, and that would also render our function impure. The greatest benefit of pure functions is that they are guaranteed to be thread-safe, so they can be run in parallel with other pure functions without any problems.

Because we need to manipulate our data structure in our example, we need to have a wrapper interface to provide locking as shown in listing 8.2. You can see in the Get method that if the token can't be found in the dictionary, it's rebuilt by reading related data from the database. Reading from the database can be time consuming, and that means all requests would be put on-hold for that read operation to finish.

Listing 8.2 Lock-based thread-safe dictionary

```
class ApiTokens {
  private Dictionary<string, Token> tokens { get; } = new();    #A

  public void Set(string key, Token value) {
    lock (tokens) {
      tokens[key] = value;    #B
    }
  }

  public Token Get(string key) {
    lock (tokens) {
      if (!tokens.TryGetValue(key, out Token value)) {
        value = getTokenFromDb(key);    #C
        tokens[key] = value;
        return tokens[key];
      }
      return value;
    }
  }

  private Token getTokenFromDb(string key) {
    … a time consuming task …
  }
}
```

#A This is the shared instance of the dictionary
#B A lock is still needed here because it's a multi-step operation.
#C This call can take a long time, blocking everyone else.

That's not scalable at all, and a lock-free alternative would be great here. .NET provides two different sets of thread-safe data structures, one is whose names start with `Concurrent*` where short-lived locks are used. They're not all "lock-free" as we discussed before; they still use locks, but they're optimized to hold them for brief periods of time, making them quite fast, and possibly simpler than a true lock-free alternative. The other set of alternatives is `Immutable*` where the original data is never changed but every modify operation creates a

new copy of the data with the modifications. It's slow as it sounds, but there are cases where they might be desirable over Concurrent flavors.

If we use ConcurrentDictionary instead, our code suddenly becomes way more scalable as shown in listing 8.3. You can now see that lock statements aren't needed anymore, so the time-consuming query can run better in parallel with other requests, blocking as little as possible.

Listing 8.3 Lock-free thread-safe dictionary

```
class ApiTokensLockFree {
  private ConcurrentDictionary<string, Token> tokens { get; } = new();

  public void Set(string key, Token value) {
    tokens[key] = value;
  }

  public Token Get(string key) {
    if (!tokens.TryGetValue(key, out Token value)) {
      value = getTokenFromDb(key);    #A
      tokens[key] = value;
      return tokens[key];
    }
    return value;
  }

  private Token getTokenFromDb(string key) {
    … a time consuming task …
  }
}
```

#A This will run parallel now!

A minor downside of this change is that multiple requests can run such an expensive operation like `getTokenFromDb` for the same token in parallel because there no locks preventing that from happening anymore. So, at worst case, you'd be running the same time-consuming operation in parallel for the same token unnecessarily, but even in that case, you wouldn't be blocking any other requests, so it's likely to beat the alternate scenario. Not using locks might be worth it.

8.1.1 Double-checked locking

There is another simple technique that lets you avoid using locks for certain scenarios. For example, ensuring only a single instance of an object is created when multiple threads are requesting it can be hard. What if two threads make the same request at once? For example, let's say we have a cache object. If we accidentally provide two different instances, different parts of code would have a different cache, causing inconsistencies or wastefulness at best. To avoid this problem, you protect your initialization code inside a lock to make sure as shown in listing 8.4. The static Instance property would hold a lock before creating object so it makes sure that no other instances would create the same instance twice.

Listing 8.4 Ensuring only one instance is created

```
class Cache {
  private static object instanceLock = new object();    #A
  private static Cache instance;    #B
  public static Cache Instance {
    get {
      lock(instanceLock) {    #C
        if (instance is null) {
          instance = new Cache();    #D
        }
        return instance;
      }
    }
  }
}
```

#A The object used for locking.
#B Cached instance value.
#C All other callers wait here if there is another running in the block.
#D The object gets created, and only once too!

The code works okay, but every access to `Instance` property will cause a lock to be held. That can create unnecessary waits. Our goal is to reduce locking. So, you can add a secondary check for the value of instance: return its value before acquiring the lock if it's already initialized, and acquire the lock only if it hasn't yet, as shown in listing 8.5. It's a simple addition yet eliminates 99.9% of lock contentions in your code, making it more scalable. We still need the secondary check inside the lock statement because there is a small possibility that another thread to have already initialized the value and released the lock just before we acquired it.

Listing 8.5 Double-checked locking

```
public static Cache Instance {
  get {
    if (instance is not null) {    #A
      return instance;    #B
    }
    lock (instanceLock) {
      if (instance is null) {
        instance = new Cache();
      }
      return instance;
    }
  }
}
```

#A Notice the pattern-matching based "not null" check in C# 9.0
#B Return the instance without locking anything.

Double-checked locking may not be possible with all data structures. For example, you can't do it for members of a dictionary because you can read from a dictionary in a thread-safe manner outside of a lock while it's being manipulated.

C# has come a long way and made safe singleton initializations much easier with helper classes like `LazyInitializer`. You can write the same property code in a simpler way like in

listing 8.6. It employs double-checked locking already behind the scenes, saving you from extra work.

Listing 8.6 Safe initialization with LazyInitializer

```
public static Cache Instance {
  get {
    return LazyInitializer.EnsureInitialized(ref instance);
  }
}
```

There are other cases where double-checked locking might be beneficial, for example, if you want to make sure, say, a list only contains certain number items at most, you can safely check its `Count` property as you're not accessing any of the list items during the check. Count is usually just a simple field access and is "mostly thread-safe" unless you use the number you read for iterating through the items. An example would be like as in listing 8.7, and it would be fully thread-safe.

Listing 8.7 Alternative double-checked locking scenarios

```
class LimitedList<T> {
  private List<T> items = new();

  public LimitedList(int limit) {
    Limit = limit;
  }

  public bool Add(T item) {
    if (items.Count >= Limit) {    #A
      return false;
    }
    lock (items) {
      if (items.Count >= Limit) {    #B
        return false;
      }
      items.Add(item);
      return true;
    }
  }

  public bool Remove(T item) {
    lock (items) {
      return items.Remove(item);
    }
  }

  public int Count => items.Count;
  public int Limit { get; }
}
```

#A First check outside the lock.
#B Second check inside the lock.

You might have noticed that the code in listing 8.7 doesn't contain any indexer property to access list items with their index. That's because it's impossible to provide thread-safe

enumeration on direct index access other than fully locking the list before enumerating. Our class is only useful for counting items, not accessing them. But accessing counter property itself is quite safe, so we can employ it in our double-checked locking so we get better scalability.

8.2 Embrace inconsistency

Databases provide vast number of features to avoid inconsistencies: locks, transactions, atomic counters, transaction logs, page checksums, snapshots, and so forth. That's because they're designed for the systems in which you can't afford retrieving the wrong data, like banks, nuclear reactors, and matchmaking apps.

Reliability isn't a black and white concept. There are levels of reliability that you can survive at with significant gains in performance and scalability. NoSQL is such a philosophy that foregoes certain consistency affordances of traditional relational database systems like foreign keys and transactions while gaining performance, scalability, and obscurity in return.

You don't need to go full NoSQL to get the benefits of such an approach. You can achieve similar gains on a traditional database like MySQL, or SQL Server.

8.2.1 The dreaded NOLOCK

As a query hint, NOLOCK dictates the SQL engine that reads can be inconsistent and they can contain data from not yet committed transactions. That might sound scary, but is it really? Think about it. Let's consider our microblogging platform we discussed in previous chapters, Blabber. When you post every time, another table that contains post counts would be updated too. If a post fails to be posted, the counter shouldn't get incremented either. A sample code would look like as in listing 8.8. You can see in the code that we wrap everything in a transaction, so if the operation fails at any point, we don't get inconsistent numbers in post counts.

Listing 8.8 A tale of two tables

```
public void AddPost(PostContent content) {
  using (var transaction = db.BeginTransaction()) {    #A
    db.InsertPost(content);    #B
    int postCount = db.GetPostCount(userId);    #C
    postCount++;
    db.UpdatePostCount(userId, postCount);     #D
  }
}
```

#A Encapsulate everything in a transaction.
#B Insert post to its own table.
#C Retrieve post count.
#D Update incremented post count.

The code might have reminded you of our unique ID generator example in the previous section; remember how threads worked in parallel with steps like (read, increment, store) and we had to use a lock to ensure that we kept consistent values? The same thing's

happening here. Because of it, we sacrifice scalability. But do we need this kind of consistency? Can I entertain you with the idea of eventual consistency?

Eventual consistency is that you ensure certain consistency guarantees but only after a delay. In our example here, you can update the incorrect post counts at certain time intervals. The best thing about that is that such operation doesn't need to hold any locks. Users will rarely see their post counts not reflecting the actual post count, until it gets fixed by the system. You gain scalability, because the less locks you hold, more parallel requests can be run on the database.

A periodic query that updates a table would still hold locks on that table, but they would be more granular locks, probably on a certain row, or at worst case, on a single page on disk. You can alleviate that problem with double-checked locking: you can first run a read-only query that just queries which rows that needs to be updated, and you can only run your update query after. That would make sure that the database doesn't get nervous about locking stuff because you simply executed an update statement on the database. A similar query for that would look like as in listing 8.9. First, we execute a SELECT query to find out mismatched counts, which doesn't hold locks. We then later update post counts based on our mismatched records. We can also batch these updates, but running them individually would hold more granular locks, possibly at the row-level, so it would allow more queries to be run on the same table without holding a lock longer than necessary. The drawback is, updating every individual row will take longer, but it will finish eventually.

Listing 8.9 Code running periodically to achieve eventual consistency

```
public void UpdateAllPostCounts() {
  var inconsistentCounts = db.GetMismatchedPostCounts();    #A
  foreach (var entry in inconsistentCounts) {
    db.UpdatePostCount(entry.UserId, entry.ActualCount);    #B
  }
}
```

#A No locks are hold while running this query.
#B A lock is held only for a single row when running this.

A SELECT query in SQL doesn't hold locks on the table, but it can still be delayed by another transaction. That's where NOLOCK as a query hint comes into the picture. NOLOCK query hint lets a query to read *dirty data*, but in return it doesn't need to respect locks held by other queries, or transactions. It's easy to you, for example, in SQL server instead of `SELECT * FROM customers` you instead use `SELECT * FROM customers (NOLOCK)` which applies NOLOCK to the `customers` table.

What does dirty data mean? If a transaction starts to write some records to the database but isn't finished yet, those records are regarded as dirty at that moment. That means, a query with a NOLOCK hint can return rows that may not exist on the database yet or that will never exist. In many scenarios, that can be a level of inconsistency your app can live with. For example, don't use NOLOCK when authenticating a user, as that might be a security issue, but there shouldn't be a problem using it on, say, showing posts. At worst, you'll see a post that seemingly exists only a brief period, and it will have gone away in the next refresh anyway. You might have experienced this already with the social platforms you're using. Users delete their content, yet those posts keep showing up in your feed although you

usually get an error trying to interact with them. That's because the platform is okay with some level of inconsistency for the sake of scalability.

You can apply NOLOCK to everything in a SQL connection by running first a SQL statement that sounds unnecessarily profound: `SET TRANSACTION ISOLATION LEVEL READ_UNCOMMITTED`. I think Pink Floyd released a song with a similar title at some point. Anyway, the statement makes more sense and conveys your intent better too.

Don't be afraid of inconsistencies if you're aware of the outcomes. Prefer intentional inconsistency to allow space for more scalability.

8.3 Don't cache database connections

It's a rather common malpractice to open a single connection to database and share it in the code. The idea is sane on paper: avoid the overhead of connection and authentication for every query so they become faster. It's also a bit cumbersome to write open and close commands everywhere. But the truth is, when you only have a single connection to database, you can't run parallel queries against the database. You can effectively only one query at a time. That's a huge scalability blocker as can be seen in figure 8.5.

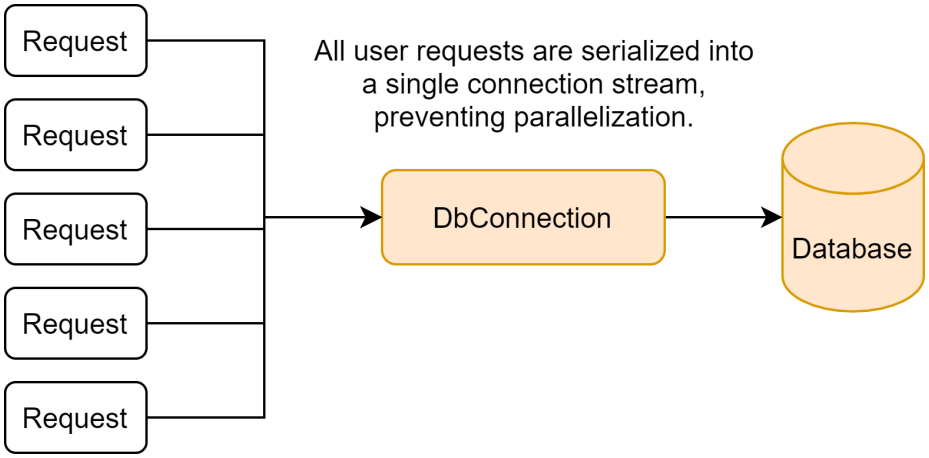


Figure 8.5 Bottleneck created by sharing a single connection in the application.

Having a single connection isn't a good idea for other reasons too. Queries might require different transaction scopes when running and they may conflict when you try to reuse a single connection for multiple queries at once.

I must agree, part of the problem comes from naming these things "connections" while they're in fact, not. You see, most client-side database connectivity libraries don't really open a connection when you create a connection object. They instead maintain certain number of already open connections and just retrieve one for you. When you think you're opening a connection, you're in fact retrieving an already open connection from what's famously called *the connection pool*. When you close the connection, the actual connection isn't closed either.

It's put back in the pool, and its state gets reset, so, any leftover work from previously running query wouldn't affect the new queries.

I can hear "I know what to do! I'll just keep a connection for every request and close the connection when the request ends!" That would allow parallel requests to be able to run without blocking each other as shown in figure 8.6. You can see in the figure that every request gets a separate connection and thanks to that they can run in parallel.

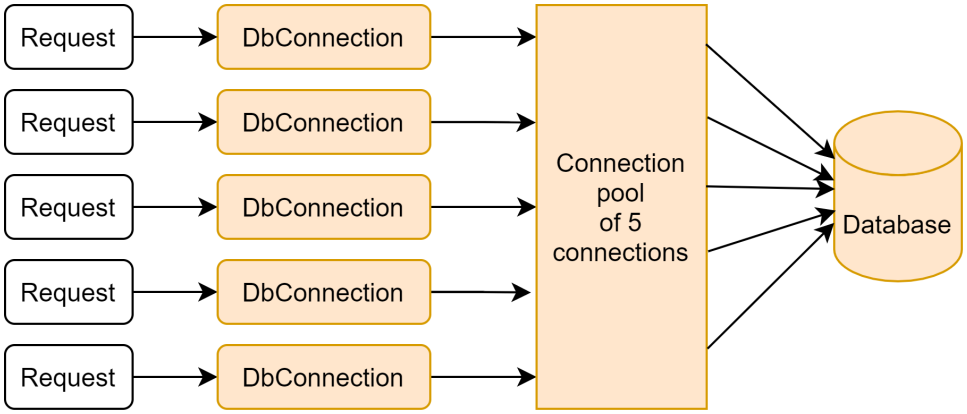


Figure 8.6 Keeping a single connection per HTTP request.

The problem with that approach is that when there are more than five requests, the connection pool must make the client wait until it can serve an available connection to them. Those requests wait in the queue, killing the ability to scale more requests, even though the request may not be in use at the time, as the connection pool has no way of knowing if the connection requested is in use or not, unless it's closed explicitly. This situation is shown in figure 8.7.

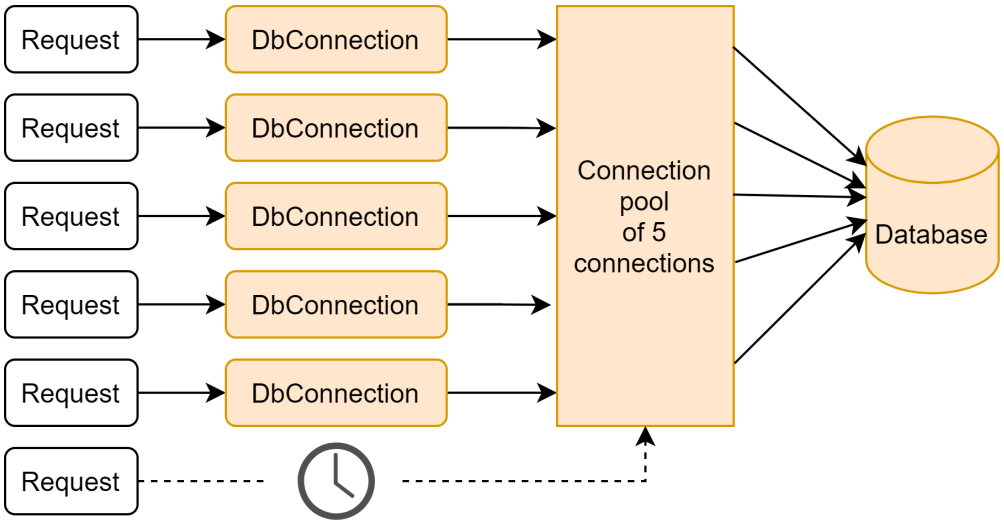


Figure 8.7 Per-request connection objects blocking additional request.

What if I told you that there is an even better approach, completely unintuitive but will make the code as scalable as possible? The secret solution is to maintain connections only for the lifetime of the queries. This would return the connection to the pool as soon as possible, allowing other requests to grab the available connection, leading to maximum scalability. How it works is shown in figure 8.8. You can see how connection pool serves no more than three queries at once, leaving room for another request or two.

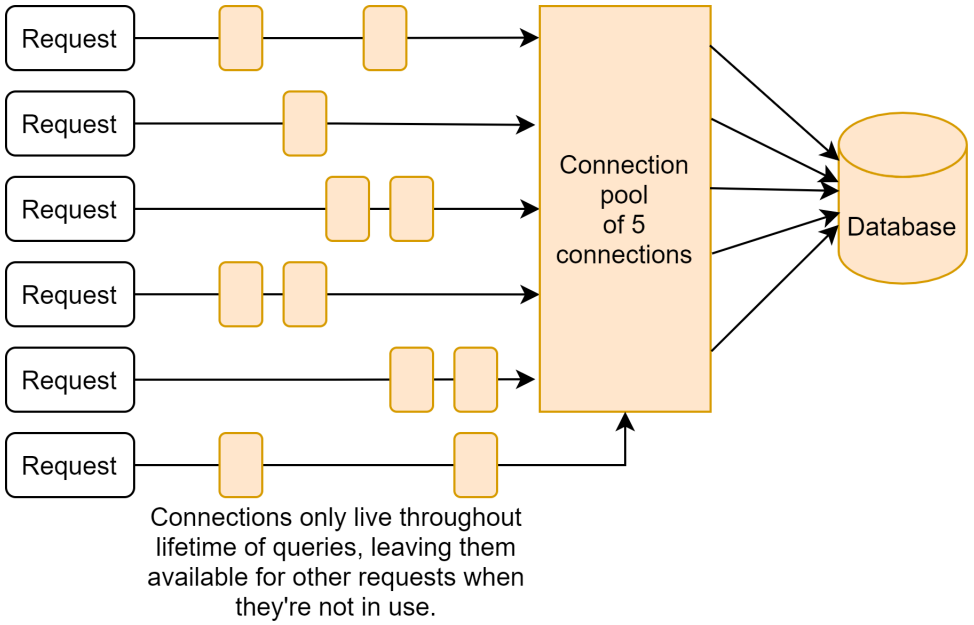


Figure 8.8 Per query connections to the database.

The reason that works is because a request is never just about running a query. There is usually some processing happening other than queries themselves. That means that the time you hold a connection object while something irrelevant running is wasted. By keeping connections open as short as possible, you leave maximum number of available connections for other requests.

The problem is that it's more work. Consider an example where you need to update preferences of a customer based on their name. A query execution is pretty much like in listing 8.10 normally. You run the queries right away, without considering connection lifetime.

Listing 8.10 A typical query execution with a shared connection instance

```
public void UpdateCustomerPreferences(string name, string prefs) {
  int? result = MySqlHelper.ExecuteScalar(customerConnection,    #A
      "SELECT id FROM customers WHERE name=@name",
      new MySqlParameter("name", name)) as int?;
  if (result.HasValue) {
    MySqlHelper.ExecuteNonQuery(customerConnection,    #A
        "UPDATE customer_prefs SET pref=@prefs",
        new MySqlParameter("prefs", prefs));
  }
}
```

#A Using a shared connection

That's because you have an open connection that you can reuse. Had you added the connection open and close code, it becomes a little bit more involved like in listing 8.11. You might think that we should close and open the connection between two queries so the connection can be returned to the connection pool for other requests, but that's completely unnecessary for such a short period of time. You would even be adding more overhead instead.

Listing 8.11 Opening connections for each query

```
public void UpdateCustomerPreferences(string name, string prefs) {
  using var connection = new MySqlConnection(connectionString);    #A
  connection.Open();    #A
  int? result = MySqlHelper.ExecuteScalar(customerConnection,
      "SELECT id FROM customers WHERE name=@name",
      new MySqlParameter("name", name)) as int?;
  //connection.Close();    #B
  //connection.Open();    #B
  if (result.HasValue) {
    MySqlHelper.ExecuteNonQuery(customerConnection,
        "UPDATE customer_prefs SET pref=@prefs",
        new MySqlParameter("prefs", prefs));
  }
}
```

#A Ceremony to open a connection to database
#B This is just silly.

You can wrap the connection open ceremony in a helper function and avoid writing it everywhere like this:

```
using var connection = ConnectionHelper.Open();
```

That saves you some keystrokes but it's prone to mistakes. You might forget putting the using statement before the call, and compiler might forget to remind you about it. That means you can forget closing connections this way.

8.3.1 In the form of an ORM

Luckily, modern *ORMs* (Object Relational Mapping tools are libraries that hides the intricacies of a database by providing entirely different set of intricate abstractions) like Entity Framework do this automatically for you, so you don't need to care about when the connection would be opened or closed. It opens the connection when necessary and closes it when it's done with it. You can use a single, shared instance of a `DbContext` with Entity Framework throughout the lifetime of a request. You may not want to use a single instance of it for the whole app though, as `DbContext` isn't thread-safe.

A similar query to listing 8.11 can be written like in listing 8.12 with Entity Framework. You can write the same queries using LINQ's syntax but I find this functional syntax easier to read and more composable.

Listing 8.12 Multiple queries with Entity Framework

```
public void UpdateCustomerPreferences(string name, string prefs) {
  int? result = context.Customers
    .Where(c => c.Name == name)
    .Select(c => c.Id)
    .Cast<int?>()
    .SingleOrDefault();    #A
  if (result.HasValue) {
    var pref = context.CustomerPrefs
      .Where(p => p.CustomerId == result)
      .Single();    #A
    pref.Prefs = prefs;
    context.SaveChanges();    #A
  }
}
```

#A Connection will be opened before and closed after each of these lines automatically.

You can have more space to scale your application over when you are aware of the lifetime semantics of Connection classes, connection pools, and actual network connections established to the database.

8.4 Don't use threads

Scalability isn't only about more parallelization, but it's also about conserving resources too. You can't scale beyond a full memory, nor can you scale beyond 100% CPU usage. ASP.NET Core uses a thread pool structure to keep a certain number of threads to serve web requests in parallel. The idea is quite similar to connection pool: having a set of already initialized threads let you avoid the overhead of creating them every time. Thread pools usually have more threads than the number of CPU cores on the system because threads frequently wait for something to complete, mostly I/O. This way, other threads can be scheduled on the same CPU core while certain threads are waiting for I/O to complete. You can see how more threads than number of CPU cores can help utilizing CPU cores better in figure 8.9. CPU can use the time a thread is waiting for something to complete to run another thread on the same core, by serving more threads than number of available CPU cores.

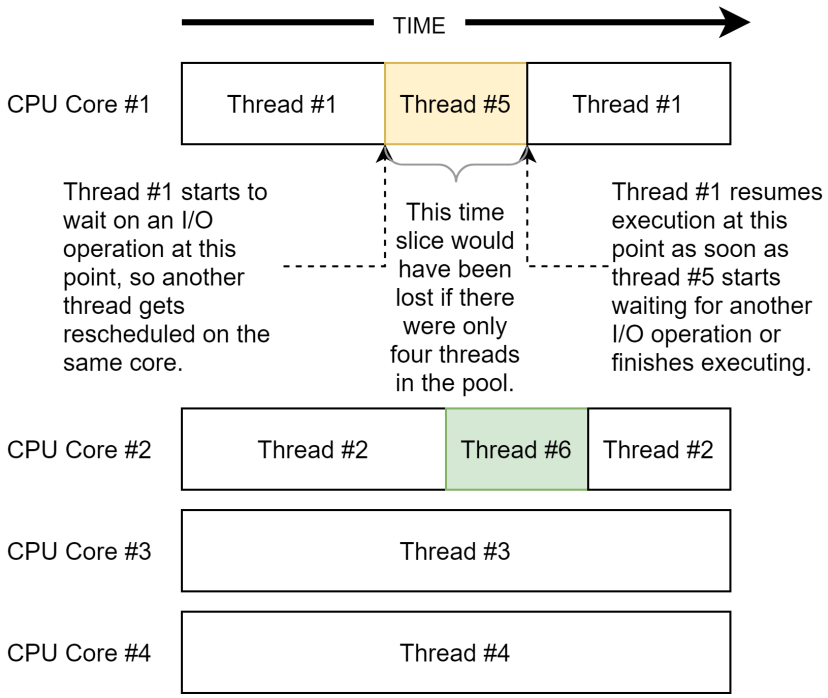


Figure 8.9 Optimizing CPU usage by having more threads than number of CPU cores.

This is better than having the same number of threads as CPU cores, but it's not precise enough to make the most use of your precious CPU time. The operating system gives threads a short amount of time to execute, then relinquishes the CPU core to other threads, to make sure every thread gets some chance to run in a reasonable time. That technique is called *preemption*, and it's how multitasking used to work with single core CPUs. Operating system juggled all the threads on the same core, creating the illusion of multitasking. Luckily, since most threads wait for I/O, users wouldn't notice that threads took turns to run on the single CPU you have, unless you ran a CPU intensive application. Then you'd feel its effects.

Because how operating systems schedule threads, having greater number of threads in the thread pool than the number of CPU cores is just a ballpark way of getting more utilization, but as a matter of fact, it can even harm scalability. If you have too many threads, they all start to get a smaller slice of CPU time, therefore they would take longer to run, bringing your web site or API to a crawl.

A more precise way to leverage time spent waiting for I/O is to use asynchronous I/O as we've seen in the previous chapter. Asynchronous I/O is explicit: wherever you have an `await` keyword, that means the thread will wait for a result of a callback, so the same thread can be used by other requests while the hardware is working on the I/O request itself. You

can serve multiple requests on the same thread in parallel this way as you can see in figure 8.10.

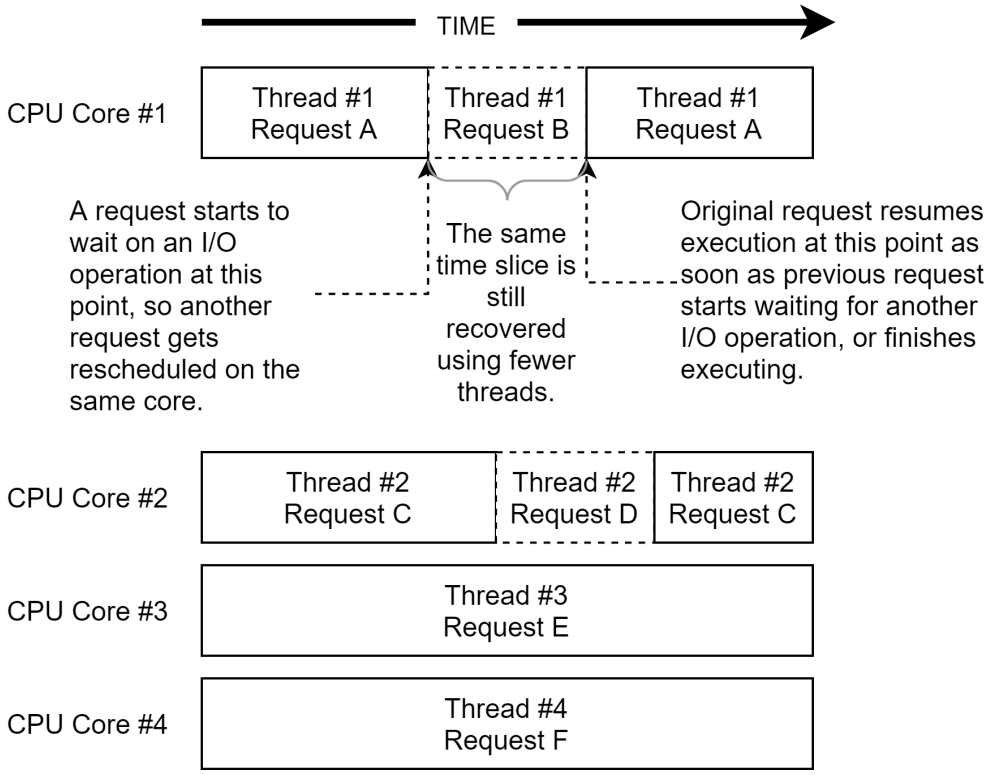


Figure 8.10 Achieving better concurrency with fewer threads and async I/O.

Asynchronous I/O is very promising. Upgrading an existing code to asynchronous I/O is straightforward too, as long as you have a framework that supports async calls at the root. For example, on ASP.NET Core, controller actions or Razor page handlers can either be written as regular methods or as asynchronous methods, as the framework builds necessary scaffolding around them. All you need to do is to rewrite the function using asynchronous calls and mark the method as `async`. Yes, you still need to make sure that your code works properly, your tests pass, but it's still a straightforward process.

Let's revise our example back in listing 8.6 and convert it to async here in listing 8.13. You don't need to go back and see the original code, as the differences are highlighted in the listing in bold here. Take a look at the differences and we'll break them down right after. It's important that you know what all these are for, so you use them consciously and correctly.

Listing 8.13 Converting blocking code to async code

```
public async Task UpdateCustomerPreferencesAsync(string name,
  string prefs) {
  int? result = await MySqlHelper.ExecuteScalarAsync(
    customerConnection,
    "SELECT id FROM customers WHERE name=@name",
    new MySqlParameter("name", name)) as int?;
  if (result.HasValue) {
    await MySqlHelper.ExecuteNonQueryAsync(customerConnection,
        "UPDATE customer_prefs SET pref=@prefs",
        new MySqlParameter("prefs", prefs));
  }
}
```

- Async functions don't actually need to be named with the suffix `Async`, but the convention helps you see that it's something you need to await. You might think that "but `async` keyword is right there!" but the keyword only affects the implementation, and it isn't part of the function signature. You must navigate the source code to find out of an async function is really async. If you don't await an async function, it returns immediately while you may incorrectly assume it has finished running. Try to stick to convention unless you can't afford it when you need specific names for your functions, such as with the names of controller actions as they can designate the URL routes as well. It also helps if you want to have two overloads of the same function with the same name, as return types aren't considered as a differentiator for overloads. That's why almost all async methods are named with an `Async` suffix in .NET.
- The `async` keyword at the beginning of the function declaration just means you can use `await` in the function. Behind the scenes, the compiler takes those async statements and generates necessary handling code and converts them into a series of callbacks.
- All async functions must return a `Task` or `Task<T>`. An async function without a return value could also have a `void` return type, but that's known to cause problems. For example, exception handling semantics change, and you lose composability. Composability in async functions lets you define an action that will happen when a function finishes in a programmatical way using `Task` methods like `ContinueWith`. Because of all that, async functions that don't have a return value should always use `Task` instead. When you decorate a function with `async` keyword, values after `return` statements are automatically wrapped with a `Task<T>`, so you don't need to deal with creating a `Task<T>` yourself.
- The `await` keyword ensures that the following line will only be executed after the expression it precedes has finished running. If you don't put `await` in front of multiple async calls, they will start running in parallel, and that can be desirable at times, but you need to make sure that you wait for them to finish, otherwise the tasks may be interrupted. On the other hand, parallel operations are prone to bugs; for example, you can't run multiple queries in parallel by using the same `DbContext` in Entity Framework Core because DbContext itself isn't thread-safe. However, you can

parallelize other I/O this way, like reading a file. Think of an example where you want to make two web requests at once. You may not want them to wait for each other. You can make two web requests concurrently and wait all of them to finish like it's shown in listing 8.14. We define a function that receives a list of URLs and starts a download task for each URL without waiting for previous one to complete so that downloads run in parallel on a single thread. We can use a single instance of HttpClient object because it's thread-safe. The function waits for all tasks to complete and builds a final response out of the results of all tasks.

Listing 8.14 Downloading multiple web pages in parallel on a single thread

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;

namespace Connections {
  public static class ParallelWeb {
    public static async Task<Dictionary<Uri, string>>    #A
      DownloadAll(IEnumerable<Uri> uris) {
      var runningTasks = new Dictionary<Uri, Task<string>>();    #B
      var client = new HttpClient();    #C
      foreach (var uri in uris) {
        var task = client.GetStringAsync(uri);    #D
        runningTasks.Add(uri, task);    #E
      }
      await Task.WhenAll(runningTasks.Values);    #F
      return runningTasks.ToDictionary(kp => kp.Key,
        kp => kp.Value.Result);    #G
    }
  }
}
```

#A Resulting type
#B Temporary storage to keep track of running tasks
#C Single instance is enough.
#D Start the task but don't await it.
#E Store the task somewhere.
#F Wait until all tasks are complete.
#G Build a new result Dictionary out of results of completed Tasks.

8.4.1 Gotchas of async code

There are things that you need to keep in mind when converting your code to async. It's easy to think that "make everything async!" and make everything worse in the process. Let's go over some of those pitfalls.

NO I/O MEANS NO ASYNC

If a function doesn't call an async function, it doesn't need to be async. Asynchronous programming only helps with scalability when you use it with I/O-bound operations. Using async on a CPU bound operation won't help scalability because those operations will need separate threads to run on, unlike I/O operations which can run in parallel on a single

thread. Compiler might also warn you when you try to use `async` keyword on a function that doesn't with other async operations. If you choose to ignore warnings, you'll just get unnecessarily bloated, perhaps slower code, due to async related scaffolding added to the function. Here is an example of unnecessary use of async keyword:

```
public async Task<int> Sum(int a, int b) {
  return a + b;
}
```

I know this happens because I've seen it in the wild, where people just decorated their functions async for no good reason. Always be explicit and clear about why you want to make a function async.

DON'T MIX SYNC AND ASYNC

It's extremely hard to call an async function in a synchronous context safely. People will say, "hey, just call `Task.Wait()`, or call `Task.Result` and you'll be fine." No, you won't be fine. That code will haunt you in your dreams, it will cause problems at the most unexpected times, and in the long run, you'll wish you can get some sleep only to have the nightmares.

The greatest problem with waiting for async functions in synchronous code that it can cause a deadlock due to other functions in the async function depending on the caller code to complete. Exception handling can also be non-intuitive as they would be wrapped inside a separate `AggregateException`.

Try not to mix asynchronous code inside a synchronous context. It's a complicated setup; that's why only frameworks do it usually. C# 7.1 added support for async Main functions, which means you can start running async code right away, but you can't call an async function from your synchronous web action. The opposite is fine though, you can, and you will, have synchronous code in your async functions as not every function is suitable for async.

MULTI-THREADING WITH ASYNC

Asynchronous I/O provides better scalability characteristics than multi-threading on I/O heavy code because it conserves less resources. But multi-threading and async are not mutually exclusive. You can have both. You can even use asynchronous programming constructs to write multi-threaded code. For example, you can run a long running CPU work in an async fashion like this:

```
await Task.Run(() => computeMeaningOfLifeUniverseAndEverything());
```

It will still run the code in a separate thread, but the await mechanism simplifies the synchronization of work completion. If you wrote the same code using traditional threads, it would look a little bit more involved. You need to have a synchronization primitive such as an event:

```
ManualResetEvent completionEvent = new(initialState: false);
```

Notice the new 'new'?

For a long time, programmers had to write `SomeLongTypeName something = new SomeLongTypeName();` to initialize an object. Typing the same type had always been a chore, even with the help of the IDE. That problem was remediated a bit after the introduction of `var` keyword in the language, but it doesn't work with class member declarations.

C# 9.0 brought a great quality of life improvement: you don't have to write the type of class after new if the type is declared before. You can go ahead and just write `SomeLongTypeName something = new();`. Brought to you by the awesome C# design team.

The event object you declare also needs to be accessible from the point of synchronization as well which creates additional complexity. The actual code becomes more involved too:

```
ThreadPool.QueueUserWorkItem(state => {
  computeMeaningOfLifeUniverseAndEverything();
  completionEvent.Set();
});
```

So, async programming can make some multi-threaded work easier to write, but it's neither a replacement for multi-threading completely, nor it helps scalability. Those are still regular threads, conserving resources.

8.5 Respect the monolith

There should be a post-it on your monitor that you'll only remove when you become rich from your vested startup stocks. It should say "no microservices."

The idea behind microservices is simple: if we split our code into separate self-hosted projects, it will be easier in the future to deploy those projects to separate servers, so, free scaling! The problem here, like many issues in software development we've discussed over the chapters, is added complexity. Do you split all the shared code? Do they really not share anything? What about their dependencies? How many projects will you need to update when you just change the database? How do you share context, like authentication, authorization? How do you ensure security? There'll be added roundtrip delay caused by the millisecond level delays between servers. How do you preserve compatibility? What if you deploy one first, and the other one breaks because of the new change? Do you have the capacity to handle this level of complexity?

I use the term monolith as the opposite of microservices, where the components of your software reside in a single project, or at least tightly coupled multiple projects deployed altogether to the same server. Because the components are interdependent, how do you move some of them to another server to make your app scale?

We've seen how we can achieve better scalability even on a single CPU core, let alone a single server in this chapter. Monolith can scale. It can work fine a long way until you find yourself in a situation where you must split your app. At that point, the startup you're working for is already rich enough to hire more developers to do the work. Don't complicate a new project with microservices where authentication, coordination, synchronization can become troublesome at such an early stage in the lifetime of the product. Ekşi Sözlük, more

than twenty years later, is still serving forty million users every month on a monolithic architecture. Monolith is the natural next step to switch to from your local prototype too. Go with the flow and consider adopting a microservice architecture only when its benefits overweigh its drawbacks.

8.6 Summary

- Approach scalability as a multi-step diet program. Small improvements can eventually lead you to a better scalable system.
- One of the greatest blockers of scalability is locks. You can't live with them; you can't live without them. Understand that they're sometimes dispensable.
- Prefer lock-free or concurrent data structures over acquiring locks yourself manually to make your code more scalable.
- Use double-checked locking whenever it's safe.
- Learn to live with inconsistencies for better scalability. Choose which types of inconsistencies would your business be okay with and use the opportunity to create more scalable code.
- ORMs, while usually seen as a chore, can also help you creating more scalable apps by employing optimizations that you may not think of.
- Use asynchronous I/O in all the I/O bound code that needs to be highly scalable to conserve available threads and optimize CPU usage.
- Use multi-threading for parallelizing CPU-bound work, but don't expect scalability benefits of asynchronous I/O, even when you use multi-threading with async programming constructs.
- A monolith architecture will complete a full tour around the world before the design discussion over a microservice architecture is finished.

9

Living with bugs

This chapter covers:

- Error handling best practices
- Living with bugs
- Intentional error handling
- Avoiding debugging
- Advanced rubber-duck debugging

The most profound work of literature on bugs is the book *Metamorphosis,* written by Franz Kafka. It depicts Gregor Samsa, a software developer, who wakes up one day to find out that the only bug is actually himself. Well, he isn't actually a software developer in the book as the entire practice of programming back in 1915 only consisted of a couple of pages of code Ada Lovelace wrote seventy years before Kafka. But Gregor Samsa's profession in the book was the next best thing to a software developer: a traveling salesperson.

The traveling salesperson problem

The traveling salesperson problem is a cornerstone subject in computer science because calculating the optimal route for a traveling salesperson is *NP-complete*. NP-complete is a completely unintuitive acronym that stands for "Nondeterministic Polynomial-time complete." Because many words are missing in the actual acronym, I had believed for a long time that it stood for "Non-Polynomial complete" and been very confused about it.

Polynomial-time (P) problems are basically problems that can be solved faster than trying all possible combinations which otherwise have factorial complexity, the second worst complexity of all complexities. NP is the superset of P (polynomial) problems that can only be solved with brute force. Polynomial, compared to NP, is always welcome. NP, nondeterministic polynomial time problems don't have a known polynomial algorithm to solve them, but their solution can be verified in polynomial time. NP-complete in that sense means "we're terrible at solving this, but we can verify a suggested solution quite fast."

Bugs are basic units of metrics for determining software quality. Because software developers consider every bug a stain on the quality of their craftsmanship, they usually either aim for zero bugs, or actively deny their existence by claiming that it works on their computer, or that it's a feature, not a bug.

Software development is immensely complex because of the inherent unpredictability of a program. That's the nature of a *Turing machine*, a theoretical construct that all computers and most of the programming languages are based on, thanks to the works of Alan Turing. A programming language based on a Turing machine is called *Turing complete*. Turing machines allow the infinite levels of creativity we have with software, but it's just impossible to verify their correctness without executing them. Some languages depend on a non-Turing complete machine, such as HTML, XML, or regular expressions which are way less capable than Turing complete languages. Because of the nature of a Turing machine, bugs are inevitable. It's impossible to have a bug-free program. Accepting this fact before setting out to develop software makes your job easier.

9.1 Don't fix bugs

A development team must have a triaging process for deciding which bugs to fix for any sizeable project. The term "triaging" originates from World War I where medics had to decide which patients to attend and which patients to leave unattended to allocate their limited resources for those who still had hope to survive. It's the only way to effectively utilize a scarce resource. Triaging helps in deciding what you need to fix first, or whether you should fix them in the first place.

How do you prioritize a bug? Unless you're just a single person driving all the business decisions, your team needs to have shared criteria to decide on the priority of a given bug. On the Windows team at Microsoft, we had a complicated set of criteria to decide on which bugs to fix assessed by multiple engineering authorities. Because of that, we had daily meetings to prioritize bugs and debated, in a place called War Room, whether a bug was worth fixing. That's understandable for a product with such an immense scale like Windows but may be unnecessary for most software projects. I remember I had to ask for prioritization of a bug because an automated system at an official marriage center in Istanbul was broken after an update and all marriage ceremonies had to stop. I had to make my case by breaking down being unable to marry into tangible metrics like "applicability", "impact", "severity" etc. "How many couples get married in a day in Istanbul?" had suddenly sounded like a meaningful interview question.

A simpler way to assess priority could be with a tangential second dimension called "severity." Although the goal is essentially to have a single priority, having a secondary dimension can make assessment easier when two different issues seemingly have the same priority. I find priority/severity dimensions handy and a good balance between business-oriented and technology-oriented. Priority is the business impact of a bug while severity is the impact on the customer. For example, if a web page on your platform isn't working, it's a high severity issue as it's unusable by the customer. But its priority might be entirely different depending on if it's on the home page, or an obscure page visited by few customers. Similarly, if your business logo on your home page goes missing, it might have no severity at all, yet it can have the topmost business priority. The severity dimension takes

some load off business prioritization as it's impossible to come up with accurate metrics to prioritize bugs.

Couldn't we achieve the same level of granularity with a single priority dimension? For example, instead of having three priority and severity levels, wouldn't just six priority levels do the same job? The problem is, the more levels you have, the more difficult it becomes to differentiate them. Usually, a secondary dimension helps you come up with a more accurate assessment of the importance of an issue.

You should have a threshold for priority and severity where any bugs below would be categorized as "won't fix." For example, any bug that has both low priority and low severity can be considered a won't fix and taken off your radar. Priority and severity levels usually mean as shown in table 9.1.

Table 9.1 Actual meanings of priority and severity

Priority	Severity	Actual meaning
High	High	Fix immediately.
High	Low	Boss wants this fixed.
Low	High	Let the intern fix it.
Low	Low	Won't fix. Never fix these unless there's nothing else to do at the office. In that case, let the intern fix it.

Tracking bugs incur costs too. At Microsoft, it took our team at least an hour a day just to assess the priority of bugs. It's imperative for your team to avoid re-visiting bugs that won't ever likely be fixed. Try to decide on that earlier in the process. It gains you time, and it still ensures you keep a decent product quality.

9.2 The error terror

Not every bug is caused by an error in your code, and not every error implies the existence of a bug in your code either. This relationship between bugs and errors is most evident when you see a popup dialog that says, "Unknown Error." If it's an unknown error, how are you so sure that it's an error in the first place? Maybe, it's an unknown success?

Such situations are rooted in the primitive association between errors and bugs. Developers instinctively treat all errors as bugs and try to eliminate them consistently and insistently. That kind of reasoning usually leads to "unknown error" situation, as something has gone wrong, and the developer just doesn't care to understand if it's an error or not. This understanding makes developers treat all kinds of errors the same way, usually either reporting every error regardless of whether the user needs to see them or hiding them all and burying them inside a log file on a server that no one will bother reading ever.

The solution to that kind of obsession with treating all errors the same way is to consider them as part of your state. Perhaps, it was a mistake to call errors "errors". We should have just called them "uncommon and unexpected state changes", or I don't know "exceptions" if you will. Oh wait, we already have those!

9.2.1 The bare truth of exceptions

Exceptions may be the most misunderstood construct in the history of programming. I can't even count the times when I've seen someone simply put their failing code inside a `try` block followed with an empty `catch` block, and be done with it. It's like closing the door to a room on fire, and assuming the problem will sort itself out eventually. I mean, it's not a wrong assumption, but can be quite costly.

Listing 9.1 The solution to all life's problems

```
try {
  doSomethingMysterious();
}
catch {
  // this is fine
}
```

I don't blame programmers for that either. As Abraham Maslow said in 1966, when the only tool you think you have is a hammer, every problem looks like a nail. I'm sure when the hammer was first invented, it was the next big thing, and everybody tried to adopt it in their solution process. Neolithic people probably published blog posts of hand markings on cave walls about how revolutionary the hammer is the future, and how it can make problems go away, without knowing that better tools will emerge in the future for spreading butter on bread.

I've seen instances where the developer just put a generic exception handler for the whole application that actually ignores all exceptions, preventing all crashes. Then, how come we keep getting bugs? We should have solved the bug problem a long time ago if adding an empty handler was the cure.

Exceptions are a novel solution to the undefined state problem. In the days when error handling was only done with return values, it was possible to omit handling the error, assume success, and continue running. That would put the application in a state the programmer never anticipated. The problem with unknown state is that it's impossible to know the effects of that state, or how serious it can be. That's pretty much the sole reason behind operating system fatal error screens, like the kernel panic on UNIX systems, or the infamous Blue Screen of Death on Windows. They halt the system to prevent any potential further damage. Unknown state means that you cannot make predictions about what will happen next anymore. Yes, the CPU might just freak out and enter an infinite loop, or the hard disk drive might decide to write zeros on every sector, or your Twitter account might decide to publish random political opinions in all caps.

The difference in exceptions from traditional error values is that it's possible to detect if they aren't handled, and act accordingly. The usual recourse for unhandled exceptions is to terminate the application as the given state isn't anticipated. Operating systems do the same too. They terminate the application if it fails to handle an exception. They can't do the same for device drivers or kernel-level components because they don't run in isolated memory spaces unlike user mode processes. That's why they must halt the system completely. That's less of a problem with microkernel-based operating systems as the number of kernel-level

components are minimal and even device drivers run in user space, but that has a slight performance penalty, and we haven't come to terms with it yet.

The greatest nuance we're missing with exceptions is the fact that they're *exceptional*. They're not for generic flow control; you have result values, and flow control constructs for that purpose. Exceptions are for cases where something happens outside of a function's contract, and it can't fulfill its contract anymore. A function like `(a,b) => a/b` guarantees performing a division operation, but it can't do that when `b`'s value is zero. It's an unexpected and undefined case.

Consider that you download software updates for your desktop app, store the downloaded copy on disk, and switch your app with the newly downloaded one when the user starts your app the next time. That's a common technique for self-updating apps outside a package management ecosystem. An update operation would look like figure 9.1. This is a bit naive as it doesn't take half-finished updates into account, but that's the whole point of it.

If any exception is raised at any point during the self-update, you'd get an incomplete `app2` folder that would cause the app files to be replaced with a broken version, causing a catastrophic state that's impossible to recover from.

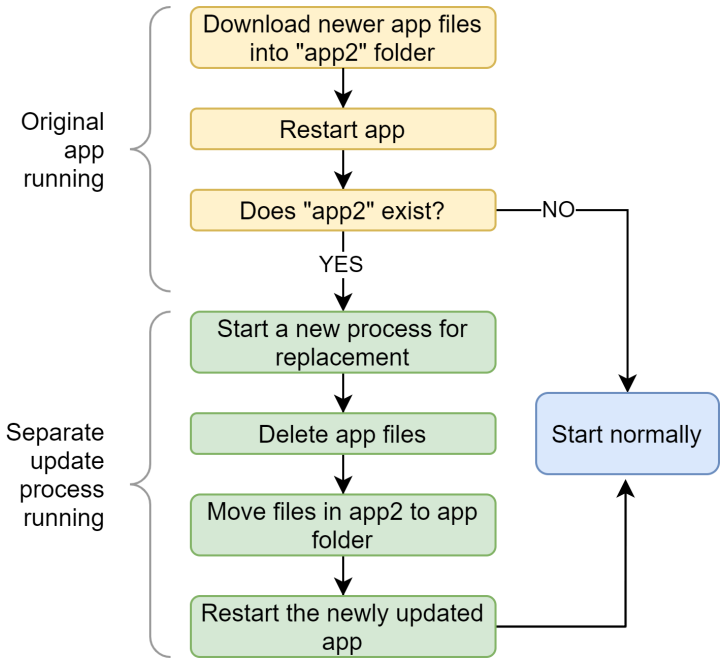


Figure 9.1 Primitive logic for a self-updating app

At every step, you can encounter an exception, and that might cause everything to fall apart if unhandled, or even handled incorrectly. The figure also shows the importance of how your

process design should also be resilient against exceptions. Any failure at any step could leave your app in a corrupt state, never to be recovered again. You shouldn't leave the app in a dirty state even when an exception happens.

9.2.2 Don't catch exceptions

Try/catch blocks are considered quick and easy patches for a crashing code because of an exception. Ignoring an exception makes the crash disappear, but doesn't make the root cause to go away.

Exceptions are supposed to cause crashes because that's the easiest way to identify the problem without causing further problems. Don't be afraid of crashes. Be afraid of bugs that don't cause clean crashes along with a convenient stack trace that helps you pinpoint the exact place where it happened. Be afraid of problems that are hidden by empty catch statements, lurking in the code disguised as a mostly correct looking state, slightly accumulating bad state over a long time, finally causing either a noticeable slow-down or a completely irrelevant looking crash, like an `OutOfMemoryException`. Unnecessary catch blocks can prevent some crashes, but might cause you to spend hours reading logs in the long run. Exceptions are great because they let you catch a problem before it becomes a hard-to-catch issue.

The first rule of exception handling is, you don't catch an exception.

The second rule of exception handling is, `IndexOutOfRangeException at Street Coder chapter 9`.

See what happens when you have only one rule? Don't catch an exception because it causes a crash. If it's caused by an incorrect behavior, fix the bug that causes it. If it's caused by a known possibility, put explicit handling statements for that specific case.

Whenever there is a possibility of getting an exception at some point in the code, ask yourself the question, "do I have a specific recourse planned for this exception, or do I just want to prevent a crash?" If it's the latter, handling that exception may not be necessary, and may be even harmful as blindly handling an exception can hide a more serious, deeper problem with your code.

Consider our self-updating application we mentioned in the previous section. It could have a function that downloads a series of application files into a folder as in listing 9.2. We need to download two files from the remote, assuming they are the latest versions. Obviously, there are many problematic issues with that approach, like not using a central registry to determine the latest version and downloading that specific version. What happens if I start downloading an update while the developers are in the middle of updating remote files? I'd get half of the files from previous version and half of them from the next version, causing a corrupt installation. For the sake of our example, let's assume the developers shut down the web server before an update, update the files, and turn it back on after it's complete, preventing such a screw up.

Listing 9.2 Code for downloading multiple files

```
private const string updateServerUriPrefix =
  "https://streetcoder.org/selfupdate/";
```

```
private static readonly string[] updateFiles =
  new[] { "Exceptions.exe", "Exceptions.app.config" };     #A

private static bool downloadFiles(string directory,
  IEnumerable<string> files) {
  foreach (var filename in updateFiles) {
    string path = Path.Combine(directory, filename);
    var uri = new Uri(updateServerUriPrefix + filename);
    if (!downloadFile(uri, path)) {
      return false;    #B
    }
  }
  return true;
}

private static bool downloadFile(Uri uri, string path) {
  using var client = new WebClient();
  client.DownloadFile(uri, path);    #C
  return true;
}
```

#A List of files to be downloaded.
#B We detect a problem with download and signal cleanup.
#C Download an individual file.

We know that `DownloadFile` can throw exceptions for variety of reasons. Actually, Microsoft has great documentation for these functions including which exceptions it can throw. There are three exceptions that `WebClient`'s `DownloadFile` method can throw:

- `ArgumentNullException` when a given argument is null.
- `WebException` when something unexpected happens during a download, like loss of internet connection.
- `NotSupportedException` when the same WebClient instance called from multiple threads, to signify that the class itself isn't thread-safe.

To prevent an unpleasant crash, a developer might choose to wrap the call to DownloadFile in a try catch, so the downloads would continue. Because many developers wouldn't care about which types exceptions to catch, they just do it with an untyped catch block as in listing 9.3. We introduce a result code, so we can detect whether an error has occurred.

Listing 9.3 Preventing crashes by creating more bugs

```
private static bool downloadFile(Uri uri, string path) {
  using var client = new WebClient();
  try {
    client.DownloadFile(uri, path);
    return true;
  }
  catch {
    return false;
  }
}
```

The problem with that approach is that you catch all three possible exceptions, two of which actually point to a definite programmer error. `ArgumentNullException` only happens when you pass an invalid argument, and the caller is responsible for this, meaning there's either bad data or bad input validation somewhere in the call stack. Similarly, `NotSupportedException` is only raised when you misuse the client. That means you're hiding many potentially easy-to-fix bugs which might lead to even more serious consequences by catching all exceptions. No, unlike being dictated by some magic animal slavery ring, you don't gotta catch 'em all. If we didn't have a return value, a simple argument error would cause files to be skipped and we wouldn't even know if they were there. You should instead catch a specific exception that's probably not a programmer error as in listing 9.4. We only catch `WebException`, that's, in fact, expected because you know, a download can fail any time for any reason, so you want to make it part of your state. Catch an exception only when it's expected. We let other types of exceptions to cause a crash because it means we were stupid and we deserve to live with its consequences before it causes a more serious problem.

Listing 9.4 Precise exception handling

```
private static bool downloadFile(Uri uri, string path) {
  using var client = new WebClient();
  try {
    client.DownloadFile(uri, path);
    return true;
  }
  catch (WebException) {    #A
    return false;
  }
}
```

#A You don't gotta catch'em all

That's why code analyzers suggest that you avoid using untyped catch blocks, because they are too broad, causing irrelevant exceptions to be caught. Catch-all blocks should only be used when you really mean catching all the exceptions in the world, probably for a generic purpose like logging.

9.2.3 Exception resiliency

Your code should work correctly even without handling exceptions, even when crashed. You should design a flow that works fine even when you constantly get exceptions, and shouldn't enter a dirty state. Your design should tolerate exceptions. The main reason for that is, exceptions are inevitable. You can put a catch-all `try`/`catch` in your `Main` method, and your app would still be terminated unexpectedly when new updates cause a restart. You shouldn't let exceptions break your application's state.

When Visual Studio crashes, the file you were changing at that moment doesn't go missing. You get reminded of the missing file when you start the application again, and you're provided an option to recover the missing file. Visual Studio manages that by constantly keeping a copy of unsaved files at a temporary location, and deleting them when the file is actually saved. So, at startup, it checks for existence for those temporary files, and

asks you if you want to recover them. You should design your code anticipating similar problems.

In our self-updating app example, your process should allow exceptions to happen, and regardless, recover from it when restarted. An exception resilient design for our self-updater would look like figure 9.2. As shown in the figure, instead of downloading individual files, we download a single atomic package, preventing us from getting an inconsistent set of files. Similarly, we back up original files before replacing them with the new ones, so we can recover in case something goes wrong.

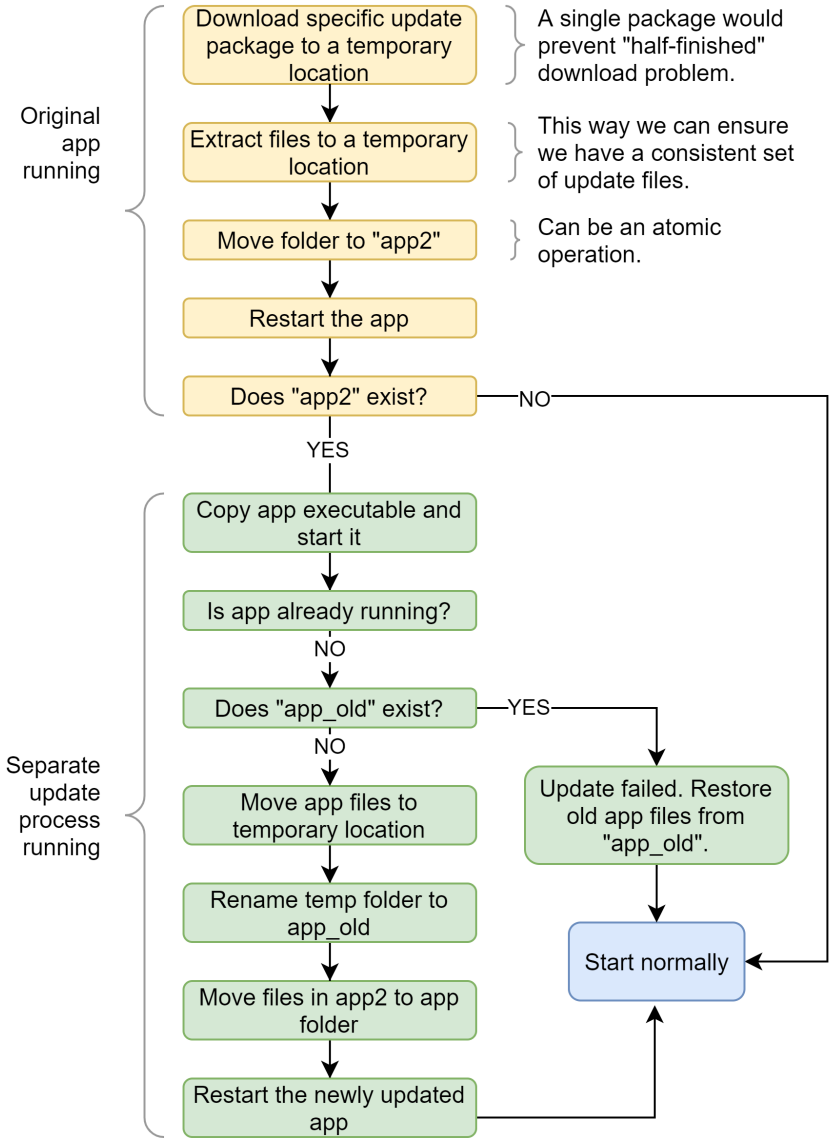


Figure 9.2 More exception resilient version of our self-updating app

How much time it takes to install updates on our devices hint to us that the software update is a complicated subject, and I'm sure I've missed many cases where this design can fail. However, you can apply similar techniques to prevent bad state in your app.

Achieving exception resilient design starts with *idempotency*. A function, or a URL, is idempotent if it returns the same result regardless of how many times it's called. That might

sound trivial for a pure function like `Sum()`, but it gets more complicated with functions that modify external state. An example is the checkout process of online shopping platforms. If you accidentally click the "Submit Order" button twice, does your credit card get charged twice? It shouldn't. I know, some web sites try to fix this by putting up a warning like "don't click the button twice!", but you know, most cats walking on the keyboard are illiterate.

Idempotency is usually thought of in a simplified manner for web requests like "HTTP GET requests should be idempotent, and anything non-idempotent should be a POST request." But, GET requests may not be idempotent, say, for content with dynamically changing parts, or a POST request can be idempotent like an upvote operation: multiple upvotes for the same content shouldn't change the number of times the users have upvoted for the same content.

How does this help us to become exception resilient? When we design our function to have consistent side-effects regardless of how many times it's called, we also gain some of the consistency benefits for when it gets interrupted unexpectedly too. Your code becomes safely callable multiple times, without causing any problems.

How do you achieve idempotency? In our example, you can have a unique order processing number, and you can create a record on the DB as soon as you start processing the order, and check its existence at the start of your processing function as shown in figure 9.3. The code needs to be thread-safe too, because some cats can walk really fast. DB transactions can help you avoid bad state as they're rolled back if they somehow cut-off because of an exception, but they may not be necessary for many scenarios.

In figure 9.3, we define an order status change operation, but how do we ensure that the we do it atomically? What if somebody else changes it before we read the result? The secret to that is to use a conditional update operation for the database which makes sure that the status is the same as the expected one. It might look like this:

```
UPDATE orders SET status=@NewState WHERE id=@OrderID status=@CurrentState
```

`UPDATE` returns the number of rows affected, so, if the state changed during the `UPDATE` operation, the operation itself would fail and it would return zero as the number of rows affected. If the state change is successful, it would return one. You can use this to atomically update the record state changes shown in figure 9.3.

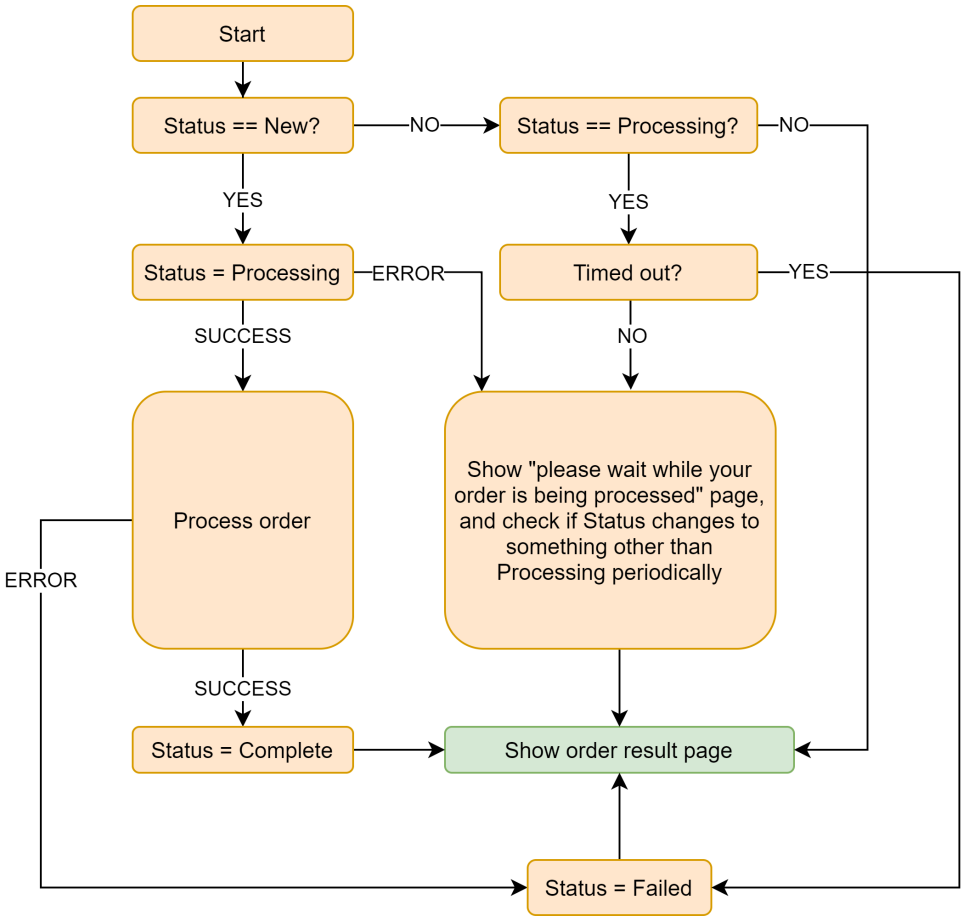


Figure 9.3 An idempotent example of order submission

An implementation example would look as shown in listing 9.5. We define every individual state the order can be in throughout order processing and make our processing able to handle the situation at different levels of processing. If it's already being processed, we just show the "still processing" page, and expire the order if it's timed out.

Listing 9.5 Idempotent order processing

```
public enum OrderStatus {
  New,
  Processing,
  Complete,
  Failed,
}

[HttpPost]
```

```
public IActionResult Submit(Guid orderId) {
  Order order = db.GetOrder(orderId);

  if (!db.TryChangeOrderStatus(order, from: OrderStatus.New,    #A
    to: OrderStatus.Processing)) {    #A
    if (order.Status != OrderStatus.Processing) {
      return redirectToResultPage(order);
    }
    if (DateTimeOffset.Now - order.LastUpdate > orderTimeout) {    #B
      db.ChangeOrderStatus(order, OrderStatus.Failed);
      return redirectToResultPage(order);
    }
    return orderStatusView(order);    #C
  }
  if (!processOrder(order)) {
    db.ChangeOrderStatus(order, OrderStatus.Failed);
  } else {
    _ = db.TryChangeOrderStatus(order, from: OrderStatus.Processing, to:
       OrderStatus.Complete);    #D
  }
  return redirectToResultPage(order);
}
```

#A Try changing status atomically.
#B Check timeout.
#C Show processing page.
#D If it fails, the result page will show the correct outcome.

Despite that being an HTTP POST request, the order submission is perfectly okay to be called multiple times without causing any unwanted side-effects, and therefore it's idempotent. If your web app crashes and you restart your application, it can still recover from many invalid states, such as a processing state. Order processing can be more complicated than this, and it might require external periodic cleanup work as well for certain cases, but you can still have great resiliency against exceptions, even with no catch statements whatsoever.

9.2.4 Resiliency without transactions

Idempotency may not be enough for exception resiliency, but it provides a great foundation because it encourages us to think about how our function would behave at different states. In our example, "Process order" step may cause exceptions, and leave dirty state around for a given order, preventing the same step from being called again. Normally, transactions protect against that because they roll back all the changes without leaving any dirty data behind. But, not every storage has transaction support, file systems for example.

You still have options even when transactions aren't available. Suppose that you created an image sharing app where people can upload albums and share them with their friends. Your content delivery network (CDN), a nifty name for file servers) could have a folder for each album with image files underneath, and you'd have album records in a database. It's quite impractical to wrap the operation of building these in a transaction as it spans multiple technologies.

The traditional approach for creating an album for a programmer is to create the album record first, create the folder, and finally upload the images to the folder based on this information. But if an exception gets raised anywhere in the process, you would get an

album record with some of the pictures missing in them. This problem applies pretty much to all kind of interdependent data.

You have multiple options to avoid this problem. In our album example, you can create the folder for images first at a temporary location, move folder to a UUID created for the album and finally create the album record as the last operation in the process. This way, users would never browse albums that are half-complete.

Another option would be to create the album record first with a status value that specifies that the record is inactive, and add rest of the data. You can finally change the status of the album record to "Active" when the insertion operation is complete. This way, you wouldn't get duplicate album records when exceptions interrupt the upload process.

In both cases, you can have periodic cleanup routines that can sweep records that are abandoned and remove them from the DB. With a traditional approach, it's hard to know whether a resource is valid or a remnant from an interrupted operation.

9.2.5 Exceptions vs errors

It can be argued that exceptions signify errors, and that may be true. But, not all errors are qualified to be exceptions. Don't use exceptions for cases where you expect the caller to handle it most of the time. In that case, that's not an exceptional situation. A very familiar example is `Parse` versus `TryParse` in .NET where the former throws an exception on invalid input while the latter just returns false.

bwwThere was only `Parse` once. Then came `TryParse` in .NET Framework 2.0 because invalid input turned out to be common and expected in most scenarios. Exceptions are an overhead in those cases because they're slow. That's because they need to carry the stack trace with them, which requires walking the stack in the first place to gather stack trace information. That can be very expensive compared to simply returning a boolean value. They're also harder to handle, because you need all the `try`/`catch` ceremony while a simple result value only needs to be checked with an `if` as shown in listing 9.6. You can see that an implementation with `try`/`catch` involves more typing, it's harder to implement correctly because the developer can easily forget to keep its exception handler specific to `FormatException`, and the code is harder to follow too.

Listing 9.6 A tale of two parses

```
public static int ParseDefault(string input,    #A
  int defaultValue) {
  try {
    return int.Parse(input);
  }
  catch (FormatException) {    #B
    return defaultValue;
  }
}

public static int ParseDefault(string input,    #C
  int defaultValue) {
  if (!int.TryParse(input, out int result)) {
    return defaultValue;
  }
```

```
  return result;
}
```

#A Implementation with Parse
#B It's tempting to omit exception type here.
#C Implementation with TryParse

`Parse` still has its place when you expect the input to be always correct. In that case, you do want an exception to be thrown. It's a dare in a way, because then you're sure that an invalid input value is a bug. "Crash if you can!"

So, regular error values are good enough to return responses most of the time. It's even okay not to return anything if you have no use for the return value. For example, if you expect an upvote operation always to be successful, don't have a return value. The function's return already signifies a success.

You can have different types of error results based on how much you expect the caller needs the information. If the caller only cares about success or failure and not the details, returning a `bool` is perfectly fine, with `true` signifying success; `false`, on the other hand, is failure. If you have a third state though, or you're using `bool` already for specifying something else, then you might need a different approach.

For example, Reddit has voting functionality, but only if the content's recent enough. You can't vote on comments or posts older than six months. You also can't vote on deleted posts. That means, voting can fail in multiple ways and that difference might need to be communicated to the user. You can't just say "voting failed: unknown error" because the user might think it's a temporary problem and keep trying. You have to say "this post is too old" or "this post is deleted", so the user learns about that specific platform dynamic and stops trying to vote. I mean, a better user experience could be to hide voting buttons so the user would immediately know that the post is "unvoteable", but Reddit insists on showing them.

In those cases you can simply use an enum to differentiate between different failure modes. A possible enum for a Reddit voting result could look like as in listing 9.7. That may not be comprehensive, but we don't need additional values for other possibilities because we don't have any plans for them. For example, if voting fails because of a DB error, that must be an exception, not a result value. It either points to an infrastructure failure, or a bug. You want your call stack; you want it to be logged somewhere.

Listing 9.7 Voting result for Reddit

```
public enum VotingResult {
  Success,
  ContentTooOld,
  ContentDeleted,
}
```

The great thing about enums is that the compiler can warn you about unhandled cases when you use switch expressions. You get a warning for cases that you didn't handle as they're not exhaustive enough. The C# compiler can't do the same for switch statements though, only switch expressions because they're newly added to the language, and can be designed for these scenarios. A sample exhaustive enum handling for an upvote operation might look like

listing 9.8. You might still get a separate warning for the switch statement not being exhaustive "enough" because in theory, you can assign invalid values to enums due to initial design decisions made for C# language.

Listing 9.8 Exhaustive enum handling

```
[HttpPost]
public IActionResult Upvote(Guid contentId) {
  var result = db.Upvote(contentId);
  return result switch {
    VotingResult.Success => success(),
    VotingResult.ContentTooOld
      => warning("Content is too old. It can't be voted"),
    VotingResult.ContentDeleted
      => warning("Content is deleted. It can't be voted"),
  };
}
```

9.3 Don't debug

Debugging is an ancient term; it even predates programming before Grace Hopper made it popular in the 1940s by finding an actual moth in the relays of a Mark II computer. It was originally used in aeronautics for processes used in finding out aircraft faults, which is now being replaced with Silicon Valley's more advanced practice of firing the CEO whenevr a problem is discovered after the fact.

The modern understanding of debugging mostly implies running the program under a debugger, putting breakpoints, tracing the code step by step, and examining the state of the program. Debuggers are very handy, but they're not always the best tools. It can be very time consuming to find out the root cause of a problem. It may not be even possible to debug a program in all circumstances. You may not even have access to the environment the code is running.

9.3.1 `printf()` debugging

Inserting console output lines inside your program to find a problem is an ancient trade. We developers have since got fancy debuggers with step-by-step debugging features but they aren't always the most efficient tools to identify the root cause of a problem. Sometimes, a more primitive approach can work better to identify an issue. "`printf()` debugging" gets its name from the `printf()` function in the C programming language. Its name stands for "print formatted." It's quite similar to `Console.WriteLine()` albeit with a different formatting syntax.

Checking the state of the application continuously is probably the oldest way to debug programs. It even predates computer monitors. Older computers were equipped with lights on their front panel that actually showed bit states of the registers of the CPU, so programmers could understand why something didn't work. Lucky me, computer monitors were invented before I was born.

`printf()` debugging is a similar way to show the state of the running program periodically, so the programmer can understand where the issue happens. It's usually

frowned upon as a newbie technique, but it can be superior to step-by-step debugging for several reasons. For example, the programmer can pick a better granularity for how frequently the state should be reported. With step-by-step debugging, you can only set breakpoints at certain places, but you can't really skip more than a single line. You either need complicated breakpoint setup, or you just need to press the "Step Over" key tediously. It can get quite time consuming and boring.

More importantly, `printf()`, or `Console.WriteLine()`, writes the state to the console terminal which has history. That's significant since you can build a chain of reasoning between different states by looking at your terminal output, which is something you can't do with a step-by-step debugger.

Not all programs have visible console output, web applications or services for example. .NET has alternatives for those environments, primarily `Debug.WriteLine()` and `Trace.WriteLine()`. `Debug.WriteLine()` writes the output to the debugger output console which is shown in Debugger Output window on Visual Studio instead of the application's own console output. The greatest benefit of `Debug.WriteLine` is that calls to it get stripped completely from optimized ("Release") binaries, so they don't affect the performance of the released code.

That, however, is a problem for debugging production code. Even if the debug output statements had been kept in the code, you'd have no practical way to read them. `Trace.WriteLine()` is a better tool in that sense, because .NET tracing can have runtime configurable listeners apart from the usual output. You can have trace output written to a text file, an event log, an XML file, and anything you can imagine with the right component installed. You can even reconfigure it while the application is running, thanks to .NET's magic.

It's trivial to set up tracing so you can enable it while your code is running. Let's consider an example, a live running web application where we might need to enable tracing while it's running to identify a problem.

9.3.2 Dump diving

Another alternative to step-by-step debugging is to examine *crash dumps*. Crash dumps, while not necessarily created after a crash, are files that contain the contents of the snapshot of the memory space of a program. They're also called *core dumps* on UNIX systems. You can manually create crash dumps with a right-click on a process name on Windows Task Manager and by clicking on "Create dump file" as shown in figure 9.3. That's a non-invasive operation that would only pause the process until the operation is complete, but keep the process running after.

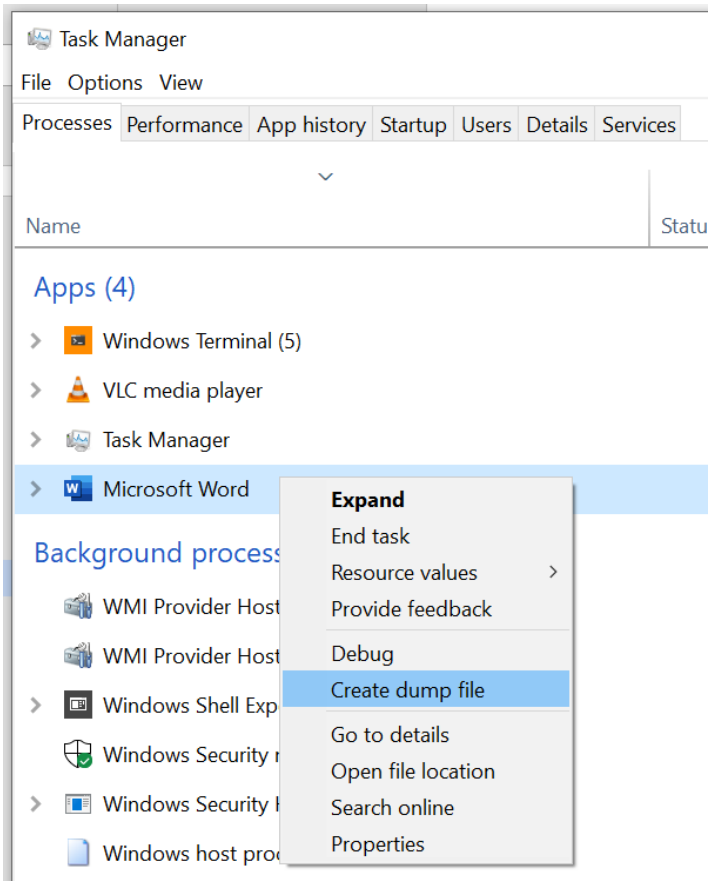


Figure 9.3 Manually generate a crash dump on a running application.

You can perform the same kind of smooth core dumping on UNIX variants without killing the app, but for you it's slightly more involved. It requires you to have dotnet dump tool installed:

```
dotnet tool install --global dotnet-dump
```

The tool's great for analyzing crash dumps, so it's a good idea to have it installed even on Windows. The installation command is the same for Windows.

There is a project on GitHub, under examples for this chapter, called InfiniteLoop that consumes CPU continuously. That could be our web application or our service running on a production server, and it's a good exercise to try to find out a problem on such a process. It's pretty much like honing your lockpicking skills on a lock on the table. You might not think you need lockpicking skills but wait until you hear about locksmith fees. Anyway, the whole code of the application is shown in listing 9.9. We basically run a multiplication operation in a

loop continuously without any benefit to world peace. It probably still wastes way less energy than Bitcoin though. We're using random values determined in the runtime to prevent the compiler from accidentally optimizing away our loop.

Listing 9.9 InfiniteLoop application with unreasonable CPU consumption

```
using System;

namespace InfiniteLoop {
  class Program {
    public static void Main(string[] args) {
      Console.WriteLine("This app runs in an infinite loop");
      Console.WriteLine("It consumes a lot of CPU too!");
      Console.WriteLine("Press Ctrl-C to quit");
      var rnd = new Random();
      infiniteLoopAggressive(rnd.NextDouble());
    }

    private static void infiniteLoopAggressive(double x) {
      while (true) {
        x *= 13;
      }
    }
  }
}
```

Compile the InfiniteLoop application and leave it running in a separate window. Let's assume this is our service in production and we need to find out where it's stuck, or where it consumes so much CPU. Finding out the call stack would help us a lot, and we can do that with crash dumps, without crashing anything.

Every process has a process identifier, a numeric value that is unique among other running processes, shortly called `PID`. Find out the PID of the process after you run the application. You can either use Task Manager on Windows, or just run this command on a PowerShell prompt:

```
Get-Process InfiniteLoop | Select -ExpandProperty Id
```

Or, on a UNIX system, you can just type:

```
pgrep InfiniteLoop
```

The PID of the process would be shown. You can create a dump file using that PID by writing out dotnet dump command:

```
dotnet dump collect -p PID
```

If your PID is, say, 26190, type:

```
dotnet dump collect -p 26190
```

The command would show where the crash dump is saved:

```
Writing full to C:\Users\ssg\Downloads\dump_20210613_223334.dmp
Complete
```

You can later use analyze command of dotnet-dump on that generated dump file:

```
dotnet dump analyze .\dump_20210613_223334.dmp
Loading core dump: .\dump_20210613_223334.dmp ...
Ready to process analysis commands. Type 'help' to list available commands or 'help
       [command]' to get detailed help on a command.
Type 'quit' or 'exit' to exit the session.
> _
```

You'd use forward slashes for UNIX pathnames instead of backslashes of Windows. This distinction has an interesting story that comes down to Microsoft adding directories to MS-DOS in its v2.0 instead of v1.0.

The analyze prompt accepts many commands which can be seen with `help`, but you only need to know a few of them to identify what the process is doing. One is the `threads` command which shows all the threads running under that process.

```
> threads
*0 0x2118 (8472)
 1 0x7348 (29512)
 2 0x5FF4 (24564)
 3 0x40F4 (16628)
 4 0x5DC4 (24004)
```

The current thread is marked with an asterisk, and you can change the current thread with `setthread` command, like this:

```
> setthread 1
> threads
 0 0x2118 (8472)
*1 0x7348 (29512)
 2 0x5FF4 (24564)
 3 0x40F4 (16628)
 4 0x5DC4 (24004)
```

As you can see, the active thread changed. But `dotnet dump` command can only analyze managed threads, not native threads. If you try to see the call stack of a non-managed thread, you'd get an error:

```
> clrstack
OS Thread Id: 0x7348 (1)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
```

You need a native debugger like WinDbg, LLDB, or GDB to do that kind of analysis, and they work similarly in principle for analyzing crash dumps. But, we're not interested in the unmanaged stack currently, and usually, the thread 0 belongs to our app. You can switch back to thread 0, and run command `clrstack` again:

```
> setthread 0
> clrstack
OS Thread Id: 0x2118 (0)
        Child SP               IP Call Site
```

```
000000D850D7E678 00007FFB7E05B2EB InfiniteLoop.Program.infiniteLoopAggressive(Double)
       [C:\Users\ssg\src\book\CH09\InfiniteLoop\Program.cs @ 15]
000000D850D7E680 00007FFB7E055F49 InfiniteLoop.Program.Main(System.String[])
       [C:\Users\ssg\src\book\CH09\InfiniteLoop\Program.cs @ 10]
```

Apart from a couple of uncomfortably long memory addresses, the call stack makes complete sense. It shows what that thread has been doing when we got the dump down to the line number (the number after "@") that it corresponds to, and without even breaking the running process! It gets that information from debugging information files with the extension "`.pdb`" on .NET and matches memory addresses with symbols and line numbers. That's why it's important for you to deploy debugging symbols to the production server too in case you need to pinpoint the errors.

Debugging crash dumps is a deep subject and covers many other scenarios like identifying memory leaks, and race conditions. The logic is pretty much universal among all operating systems, programming languages, and debugging tools. You have a memory snapshot in a file where you can examine its contents, the call stack, and the data. Consider this a starting point and an alternative to traditional step-by-step debugging.

9.3.3 Advanced rubber duck debugging

As we've discussed briefly at the beginning of the book, rubber duck debugging is a way to solve problems by telling it to a rubber duck sitting on your desk. The idea is that while putting your problem in words, you reframe it in a clearer way so that you can magically find a solution to it.

I use StackOverflow drafts for that. Instead of asking a question on StackOverflow and wasting everybody's time with my perhaps silly question, I just write my question on the web site without posting it. Why StackOverflow then? Because being aware of the peer pressure on the platform forces you to iterate over one aspect that's crucial when constructing your question:

"What have you tried?"

Asking yourself that question has multiple benefits, but the most important one is that it helps you realize that you haven't tried all the possible solutions yet. Solely thinking about that question has made me figure out numerous times the other possibilities that I haven't considered .

Similarly, StackOverflow mods ask you to be specific. Too-broad questions are considered off-topic. That also pushes you to narrow your problem into a single specific issue helping you deconstruct your problem in an analytical way.

When you practice this on the web site, you'll make this a habit, and you'll be able to do it mentally later.

9.4 Summary

- Prioritize bugs to avoid wasting your resources on fixing bugs that don't matter.
- Catch exceptions only when you have a planned, intentional action in place for that case. Otherwise, don't catch them.
- Write exception resilient code that can withstand crashes first, instead of trying to avoid crashes as an afterthought.

- Use result codes, or enums, instead of exceptions for cases where errors are common or highly expected.
- Use framework-provided tracing affordances to identify problems faster than clunky step-by-step debugging.
- Use crash dump analysis to identify problems on a running code in production if other methods are unavailable.
- Use your drafts folder as a rubber-duck debugging tool and ask yourself what you've tried.