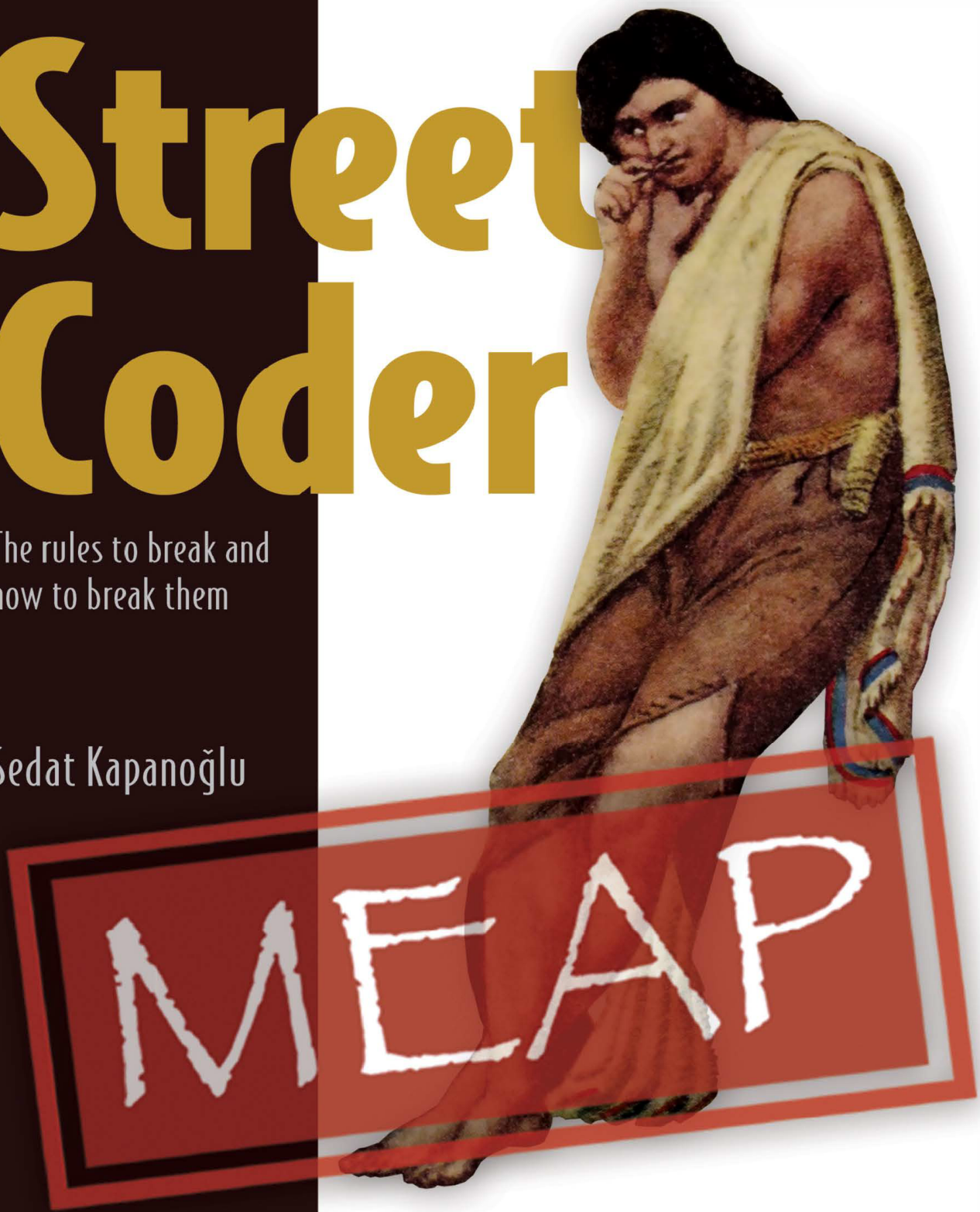
With examples in C#

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 2

Copyright 2020 Manning Publications

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

©Manning Publications Co. To comment go to liveBook

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 well-known 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

©Manning Publications Co. To comment go to liveBook

brief contents

1 To the streets

2 Practical theory

3 Useful anti-patterns

4 Tasty testing

5 Refactor code

6 Error handling

7 Write secure code

8 Write scalable code

9 Write optimized code

10 Final words

©Manning Publications Co. To comment go to liveBook

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.

©Manning Publications Co. To comment go to liveBook

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: the code 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.

©Manning Publications Co. To comment go to liveBook

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.

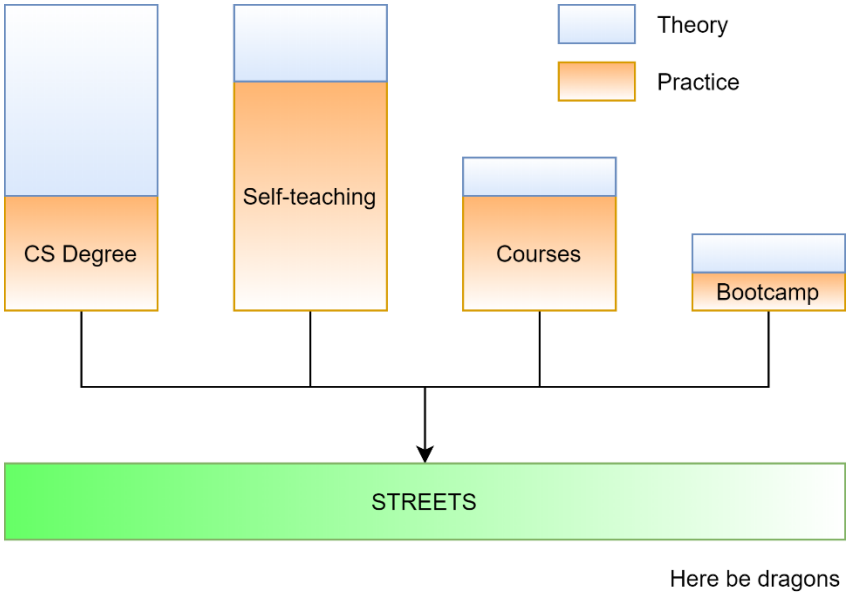


Figure 1.1 Starting a career through different paths

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.

©Manning Publications Co. To comment go to liveBook

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

[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.

©Manning Publications Co. To comment go to liveBook

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 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.

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

[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.

©Manning Publications Co. To comment go to liveBook

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 theoretical 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

©Manning Publications Co. To comment go to liveBook

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

[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.

©Manning Publications Co. To comment go to liveBook

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.

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.

---

[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

[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.

©Manning Publications Co. To comment go to liveBook

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 to algorithms and data structures the same way too. They use a dictionary data-type because it's handy to keep keys and values. They don't know the consequences.

The unconditional trust in package ecosystems and frameworks are 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 instead of 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. 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 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.

[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

©Manning Publications Co. To comment go to liveBook

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 about 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 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

[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.

©Manning Publications Co. To comment go to liveBook

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.

©Manning Publications Co. To comment go to liveBook

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 have an inherent disinclination to care about theory or a contrarian attitude against them, being 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.

©Manning Publications Co. To comment go to liveBook

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 are no exception. You are 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 are 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...".

©Manning Publications Co. To comment go to liveBook

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

©Manning Publications Co. To comment go to liveBook

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:

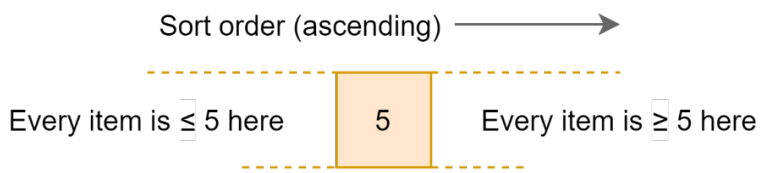


Figure 2.1 We can eliminate one side of the element with one comparison on a sorted list

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 at most 3 comparisons for a sorted array with 8 items to determine if an item exists in it, and more importantly it would only take about 10 lookups at most to find if an item exists in an array with 1000 items. That's the power you get by going over in halves. Your implementation could look like this:

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

©Manning Publications Co. To comment go to liveBook

```
        if (lookFor > value) {
            start = middle + 1;    #B
        } else {
            end = middle - 1;    #C
        }
    }
    return false;
}
```

#A Find the middle of the range
#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 Tackling the Big-O bigotry

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 sort 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

©Manning Publications Co. To comment go to liveBook

> 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.

©Manning Publications Co. To comment go to liveBook

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:

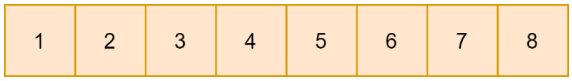


Figure 2.2 A symbolic representation of an array

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:

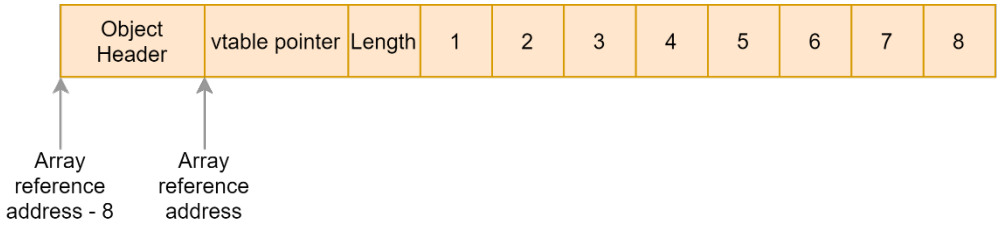


Figure 2.3 Actual layout of an array in memory

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:

©Manning Publications Co. To comment go to liveBook

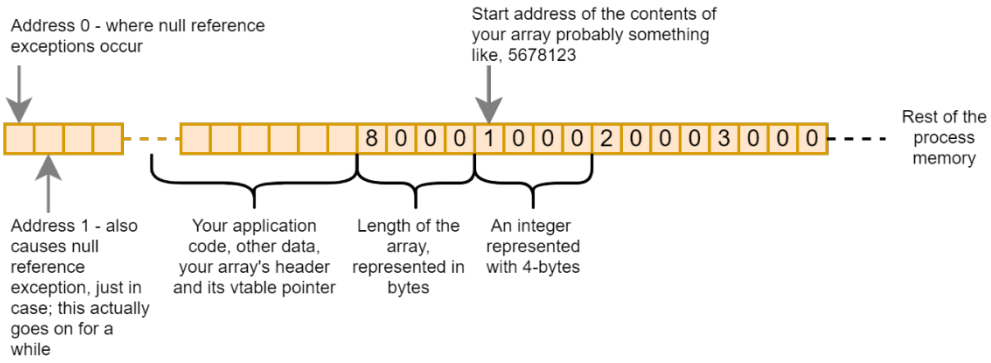


Figure 2.4 Memory space of a process and an array

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.

©Manning Publications Co. To comment go to liveBook

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 are 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

©Manning Publications Co. To comment go to liveBook

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.

©Manning Publications Co. To comment go to liveBook

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.

©Manning Publications Co. To comment go to liveBook

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` as they are both 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. Lists grow exponentially, so the growth requirements get less and less as they get bigger.

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. After it's full, it will double its capacity. 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 regardless if you know how many items it will have or not. That would 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.

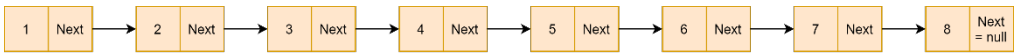


Figure 2.5 Layout of a linked list

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.

©Manning Publications Co. To comment go to liveBook

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 are 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:

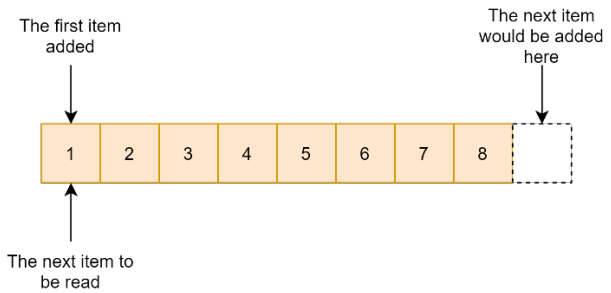


Figure 2.6 A high-level layout of a queue

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.

©Manning Publications Co. To comment go to liveBook

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 number from a data structure. 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 – 1)". 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:

©Manning Publications Co. To comment go to liveBook

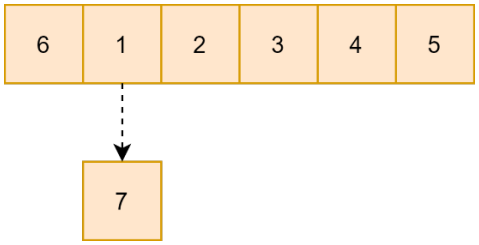


Figure 2.8 Storage of overlapping items in a dictionary

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:

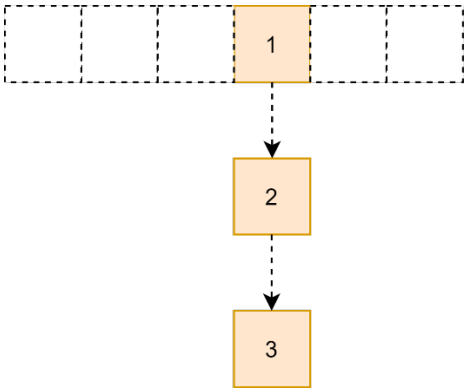


Figure 2.9 A dictionary when you screw up your GetHashCode()

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.

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

©Manning Publications Co. To comment go to liveBook

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:

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

The code might look like a hacking scene from a cheesy hacker movie. It's cryptic, hard to understand even for someone who is familiar with the operations. We are 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.

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, 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.

©Manning Publications Co. To comment go to liveBook

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:

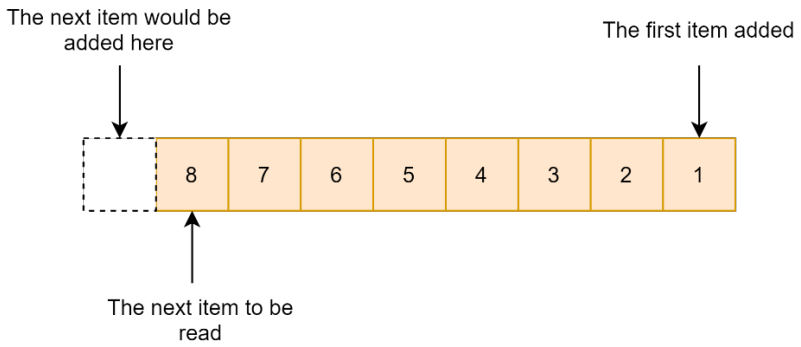


Figure 2.10 A high-level overview of a stack

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*.

©Manning Publications Co. To comment go to liveBook

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 an extra step of memory allocation or garbage collection. If you have a local reference variable though, like any variable with a class type, only the reference is stored in the stack. Actual memory buffer for the class is allocated on the heap, where it will be garbage collected later.

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

©Manning Publications Co. To comment go to liveBook

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 it becomes 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	C#, Java, Rust, Swift, Kotlin,	Python, Ruby, Lisp

©Manning Publications Co. To comment go to liveBook

Different types *cannot* be substituted for each other.	**TypeScript, C++**	
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 especially with 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. If many programming languages can work without explicit and strict types, why do we burden ourselves with 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 are developing a microblogging platform that only allows certain amount of characters in every post, so 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 are inserting the right records in the database.

©Manning Publications Co. To comment go to liveBook

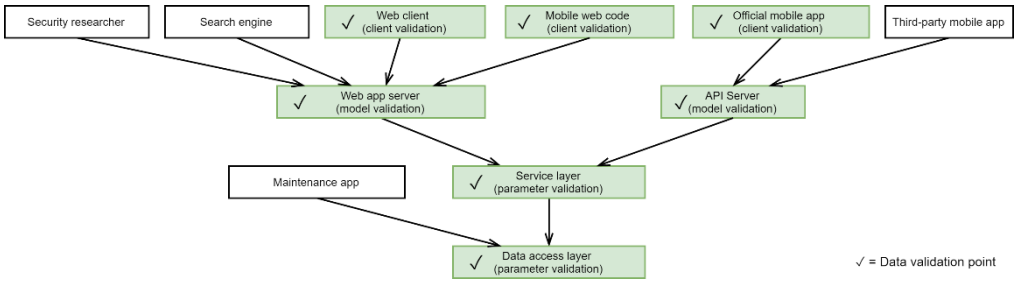


Figure 2.11 Unvalidated data sources and places where you need to validate data repetitively

That might eventually make you validate the input at multiple places around the code and you need to make sure that you are 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,

©Manning Publications Co. To comment go to liveBook

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. Don't be that person. 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 2.1. 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

©Manning Publications Co. To comment go to liveBook

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 2.1 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:

©Manning Publications Co. To comment go to liveBook

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.

©Manning Publications Co. To comment go to liveBook

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 are 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; // not a valid URL
    }
    // take the part after the last slash
    string[] parts = url.Split('/');
    string lastPart = parts[^1];    #B
    return lastPart;
}
```

#A This is an alien language called regular expressions. 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.

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

©Manning Publications Co. To comment go to liveBook

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!

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

©Manning Publications Co. To comment go to liveBook

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.

©Manning Publications Co. To comment go to liveBook

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 were 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 F12 on Visual Studio for PC.

```
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

©Manning Publications Co. To comment go to liveBook

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) {
```

©Manning Publications Co. To comment go to liveBook

```
        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
```

©Manning Publications Co. To comment go to liveBook

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.

©Manning Publications Co. To comment go to liveBook

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 2.2.

Listing 2.2 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";
    }
```

©Manning Publications Co. To comment go to liveBook

```
}
```

#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:

©Manning Publications Co. To comment go to liveBook

```
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.

Nullability checks helps 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.

©Manning Publications Co. To comment go to liveBook

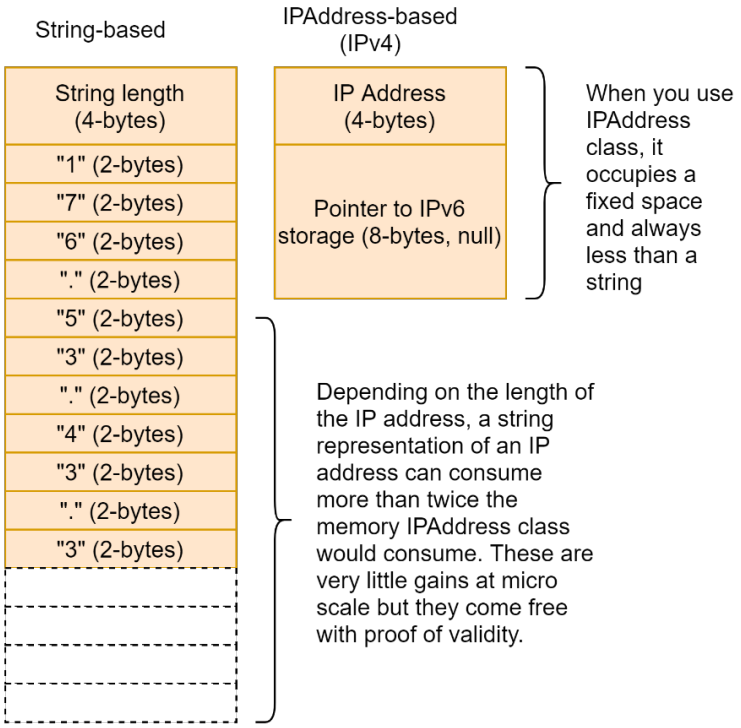


Figure 2.12 Storage differences of data types, excluding common object headers.

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.

Performance and scalability aren't simple concepts as I mentioned here, as we'll go into in later chapters, 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:

©Manning Publications Co. To comment go to liveBook

```
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:

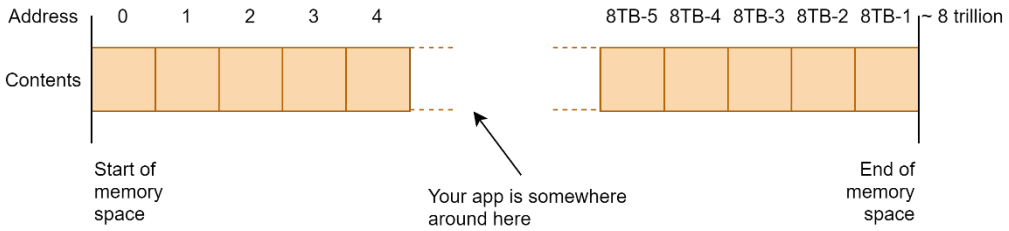


Figure 2.13 Memory layout of a 64-bit process that can address up to 8TB

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.

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

©Manning Publications Co. To comment go to liveBook

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 this example:

Listing 2.3 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,
```

©Manning Publications Co. To comment go to liveBook

```
        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 are 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.

©Manning Publications Co. To comment go to liveBook

- Knowing how call stack works also complements the performance implications of value and reference types.

©Manning Publications Co. To comment go to liveBook

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

©Manning Publications Co. To comment go to liveBook

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.

©Manning Publications Co. To comment go to liveBook

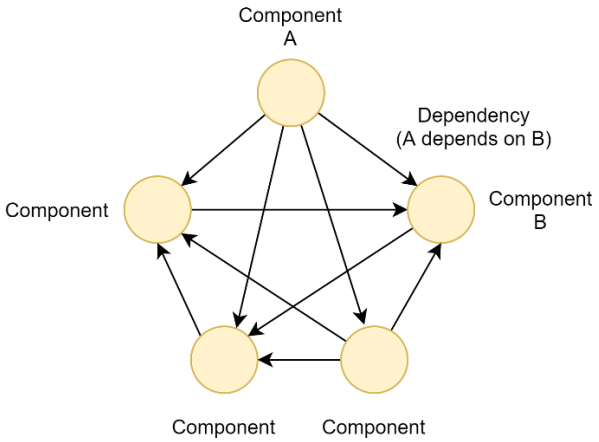


Figure 3.1 The occult symbol for dependency hell

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

©Manning Publications Co. To comment go to liveBook

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.

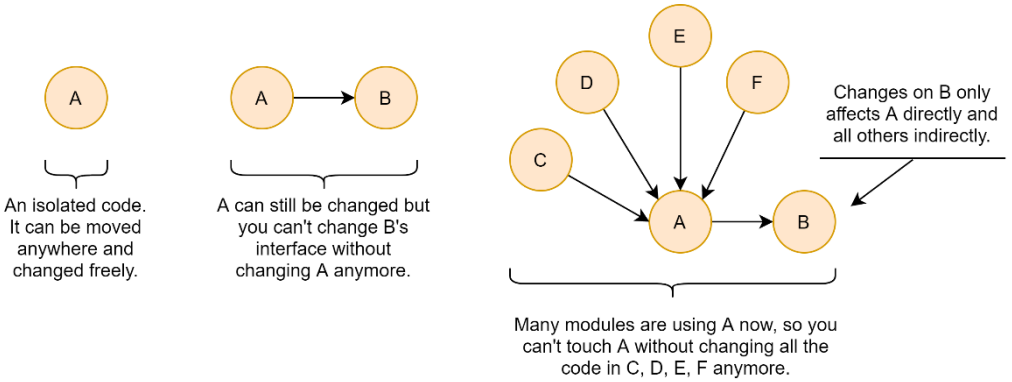


Figure 3.2 Resistance to change is proportional to dependencies.

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.

©Manning Publications Co. To comment go to liveBook

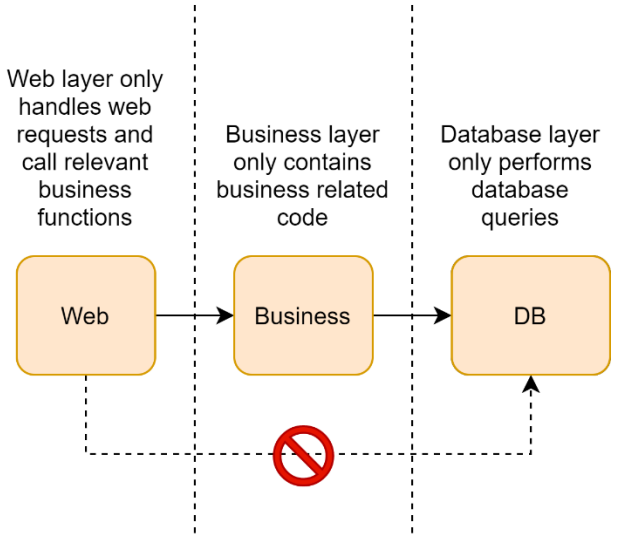


Figure 3.3 Violation of abstraction that you need to avoid.

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, the changes in the DB layer now need to be communicated 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.

©Manning Publications Co. To comment go to liveBook

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.

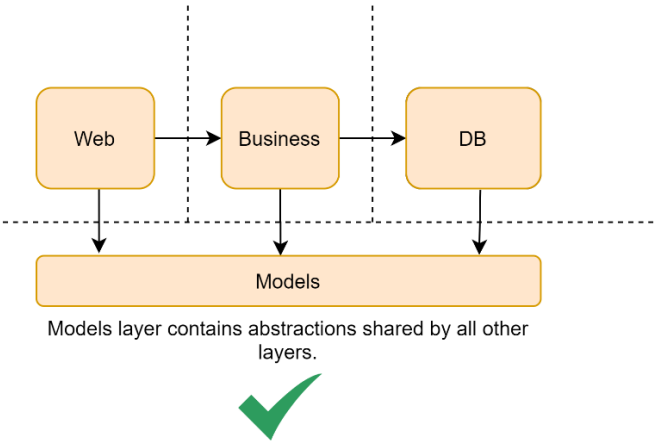


Figure 3.4 Extracting common functionality without violating abstractions

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 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.

[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)

©Manning Publications Co. To comment go to liveBook

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.

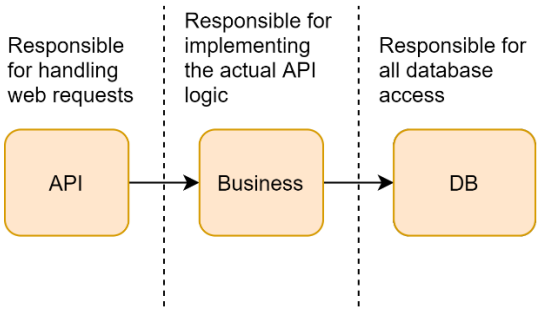


Figure 3.5 Basic architecture of our mobile app API

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

©Manning Publications Co. To comment go to liveBook

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.

©Manning Publications Co. To comment go to liveBook

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

©Manning Publications Co. To comment go to liveBook

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.

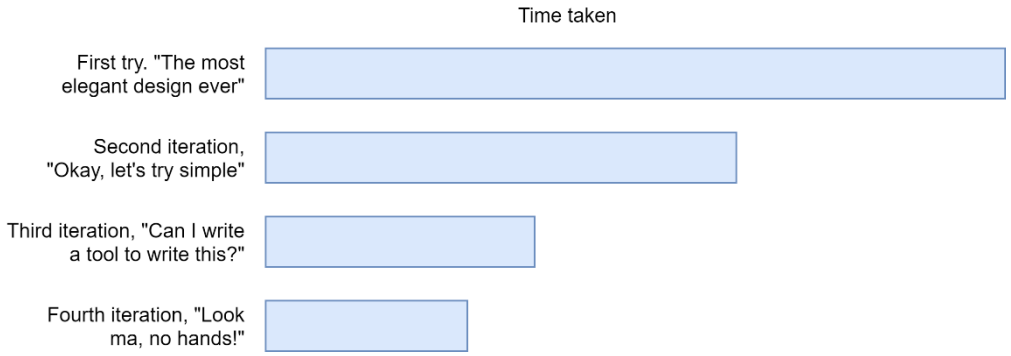


Figure 3.6 The brilliance of doing something over and over and expecting the same results

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.

©Manning Publications Co. To comment go to liveBook

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 3.6.

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.

©Manning Publications Co. To comment go to liveBook

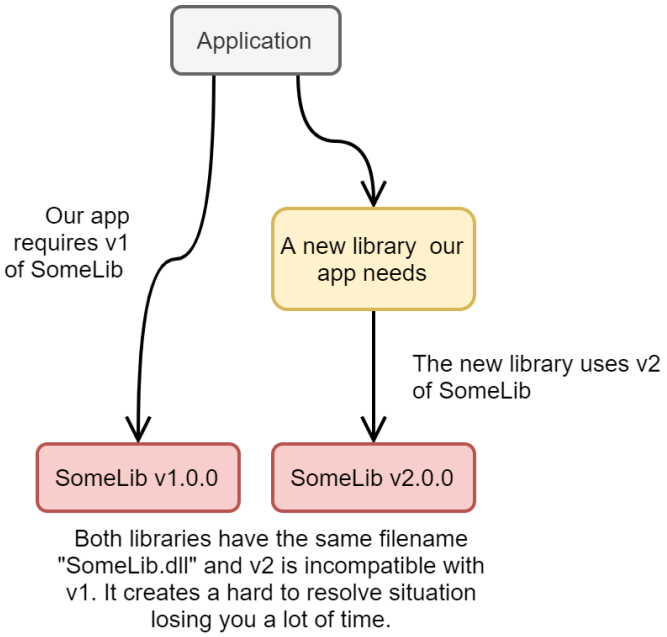


Figure 3.7 Unfixable version conflicts

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.

©Manning Publications Co. To comment go to liveBook

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

©Manning Publications Co. To comment go to liveBook

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.

©Manning Publications Co. To comment go to liveBook

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`

©Manning Publications Co. To comment go to liveBook

`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.

```
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

©Manning Publications Co. To comment go to liveBook

```
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:

©Manning Publications Co. To comment go to liveBook

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 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

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

©Manning Publications Co. To comment go to liveBook

purposes without creating confusion and more importantly, it explains its job better wherever its used.

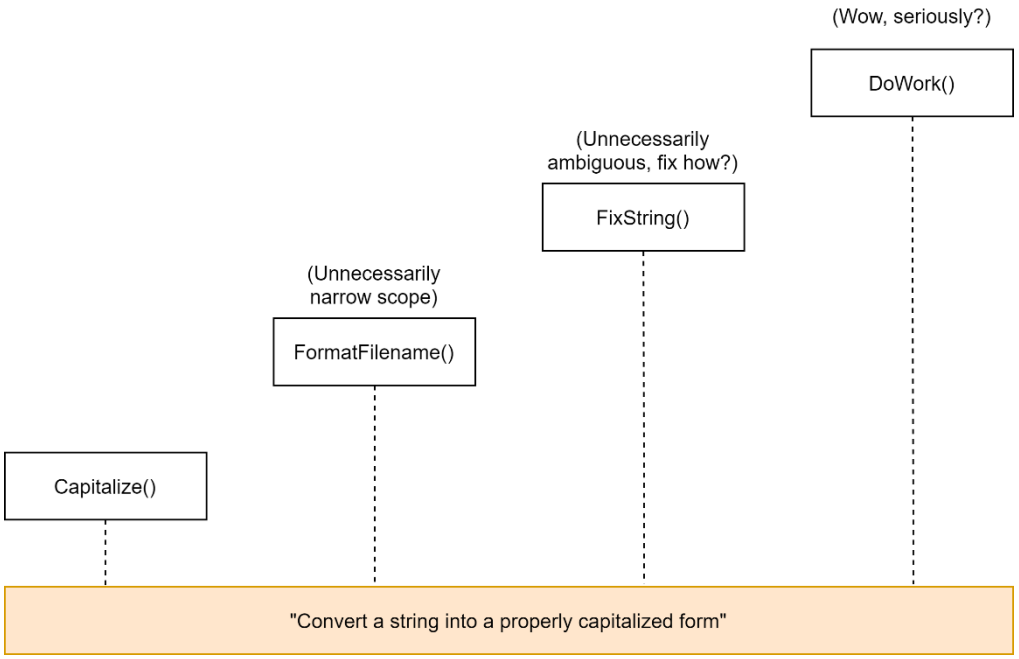


Figure 3.8 Pick a name as close as possible to the actual functionality

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.

©Manning Publications Co. To comment go to liveBook

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.

©Manning Publications Co. To comment go to liveBook

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
  }

  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.

©Manning Publications Co. To comment go to liveBook

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.

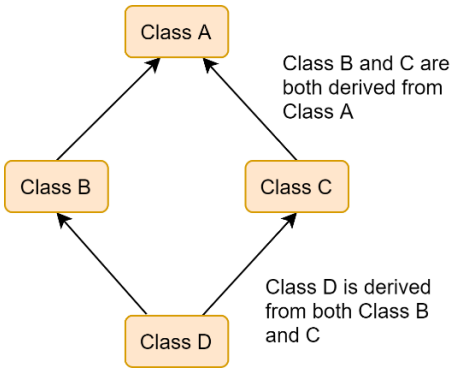


Figure 3.9 Diamond dependency problem, how should class D behave?

A greater problem with inheritance than multiple inheritance is the problem of dependencies. 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 SOLID principles which is a set of core principles for OOP stemming from the ideas of Barbara Liskov, and Michael Feathers, and brought together by Robert C. Martin. The letter D

©Manning Publications Co. To comment go to liveBook

in SOLID stands for "Dependency inversion principle", and states that code should never depend on the concrete implementation, but an abstraction instead.

Other SOLID Principles

S stands for "Single-responsibility principle" that basically says a class should be responsible for one thing only as opposed to having one class do multiple things, aka God classes.

O stands for "Open-closed principle" that basically states that a class should be open for extension but closed for modification. We should add behavioral differences with different flavors of the class instead of trying to jam as much as possible code in the same class. This goes hand in hand with Single-reponsibility principle.

L stands for Liskov Substition Principle which is explained later in this section.

I stands for "Interface segregation principle". It favors smaller and goal-specific interfaces over generalized, broadly scoped interfaces.

You won't see these in the rest of the book because this isn't a book about OOP best practices. There are great books out there for that.

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.

Inheritance also causes implied equivalence. That's not a bad thing usually. It's even dictated by *Liskov Substitution Principle* (LSP). LSP states that the correctness of a program shouldn't change if you replace a class in your program with one of its descendants. The problem is if you want to express discrete types but reuse the code at the same time, your code will fail.

How do you reuse code then? How to inherit 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.

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.

©Manning Publications Co. To comment go to liveBook

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 3.3, we are 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);    #B
  if (response.StatusCode == 200) {    #A
    dynamic obj = JObject.Parse(response.Content);    #C
    var period = obj.properties.periods[0];
    return (double)period.temperature;    #D
  }
  return null;
}
```

#A Check for successful HTTP status code.
#B Send the request to NWS.
#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;
```

©Manning Publications Co. To comment go to liveBook

```
  // ... 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

©Manning Publications Co. To comment go to liveBook

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;
  }
}
```

©Manning Publications Co. To comment go to liveBook

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.

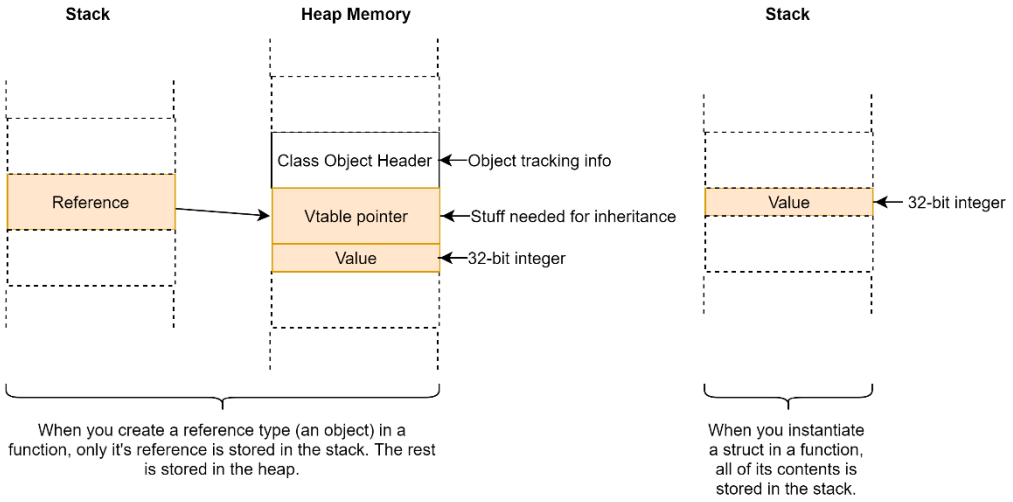


Figure 3.10 The difference between how classes and structs are laid out in memory.

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.

©Manning Publications Co. To comment go to liveBook

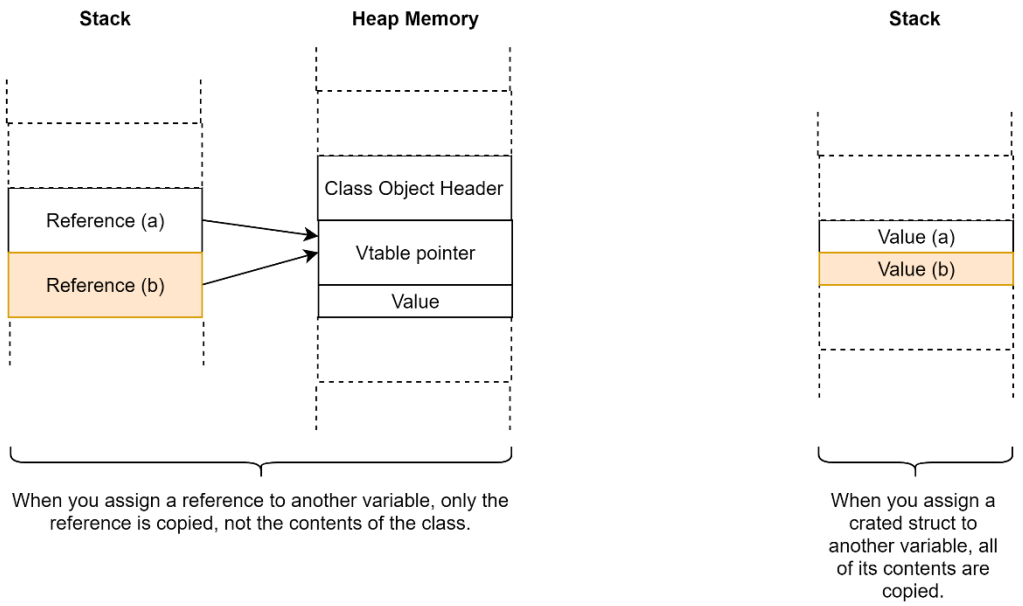


Figure 3.11 Efficiency of small structs in memory storage

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".

©Manning Publications Co. To comment go to liveBook

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.

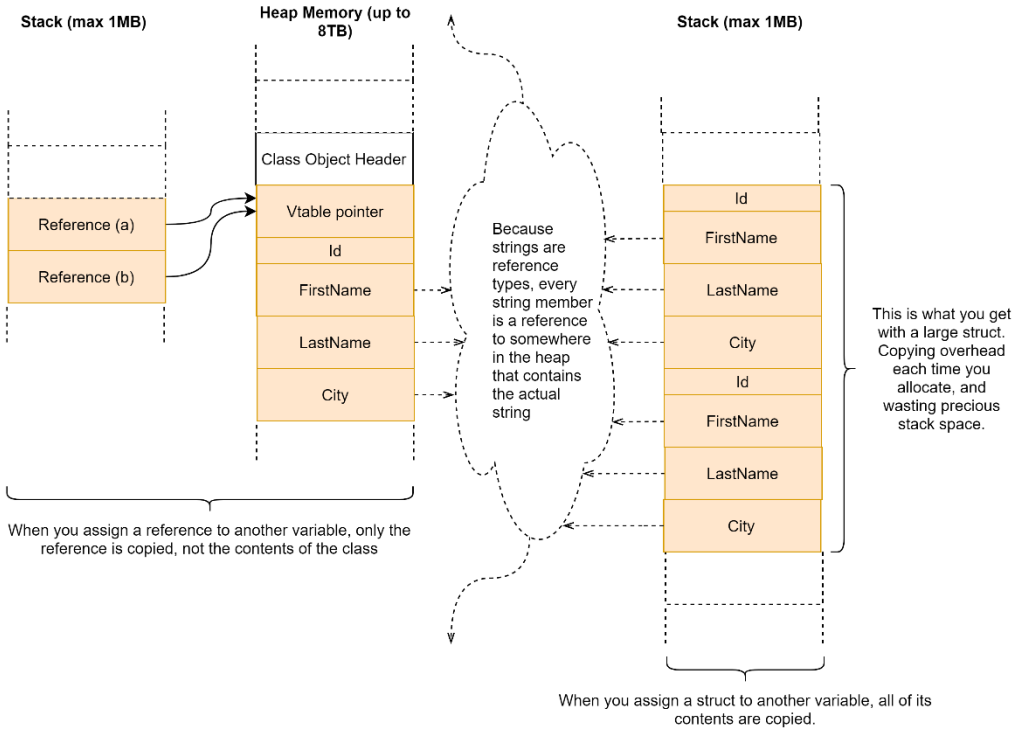


Figure 3.12 Impact difference between value types and reference types in larger types.

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.

©Manning Publications Co. To comment go to liveBook

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.

©Manning Publications Co. To comment go to liveBook

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) {
```

©Manning Publications Co. To comment go to liveBook

```
    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` 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.

©Manning Publications Co. To comment go to liveBook

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.

©Manning Publications Co. To comment go to liveBook

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);
}
```

©Manning Publications Co. To comment go to liveBook

#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.

©Manning Publications Co. To comment go to liveBook

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.

©Manning Publications Co. To comment go to liveBook

#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.

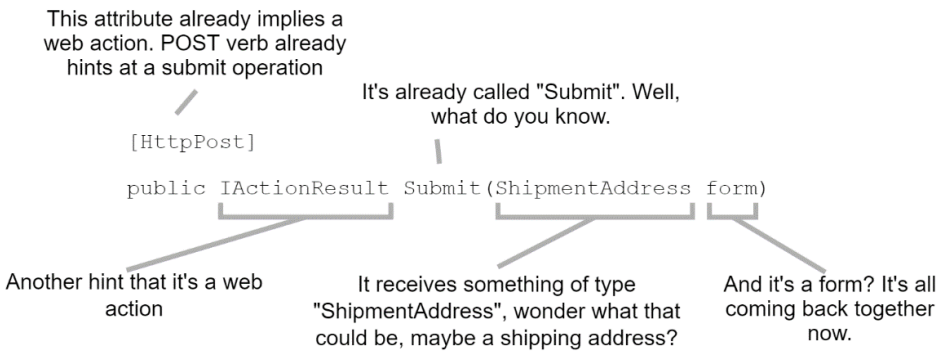


Figure 3.13 Reading hints in a code

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

©Manning Publications Co. To comment go to liveBook

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.

Listing 3.18 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 3.19, we extract parts of the logic in our Submit function by using what we identified as logical parts. We basically have a validation

©Manning Publications Co. To comment go to liveBook

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 3.19 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.

©Manning Publications Co. To comment go to liveBook

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.

©Manning Publications Co. To comment go to liveBook

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

©Manning Publications Co. To comment go to liveBook

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 doe, 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 the 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.

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.

©Manning Publications Co. To comment go to liveBook

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 all of the 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!

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 in order to decide how you want to test it. Those are mainly risk and

[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.

©Manning Publications Co. To comment go to liveBook

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.

©Manning Publications Co. To comment go to liveBook

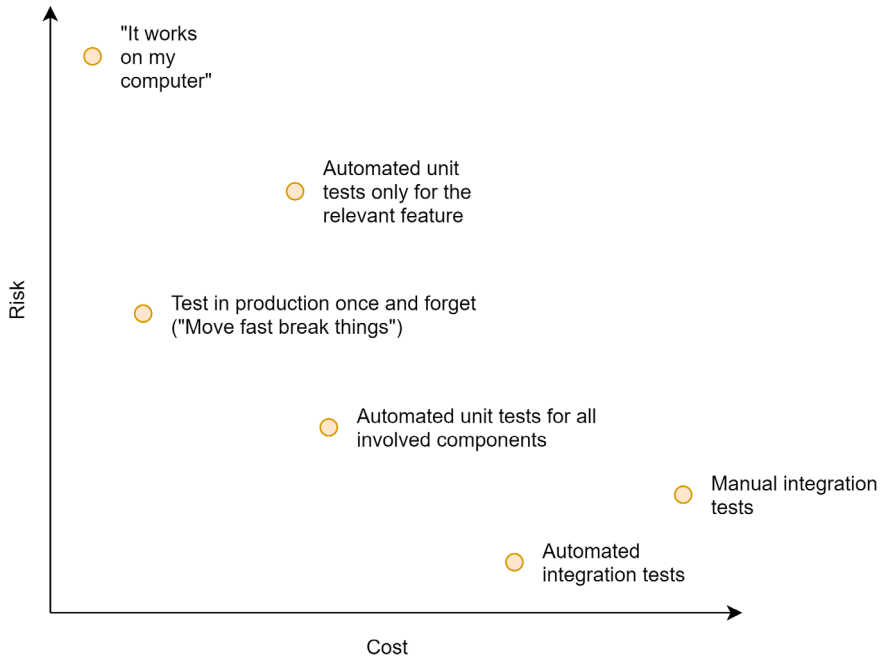


Figure 4.1 An example mental model to place different options of testing in your mind

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.

©Manning Publications Co. To comment go to liveBook

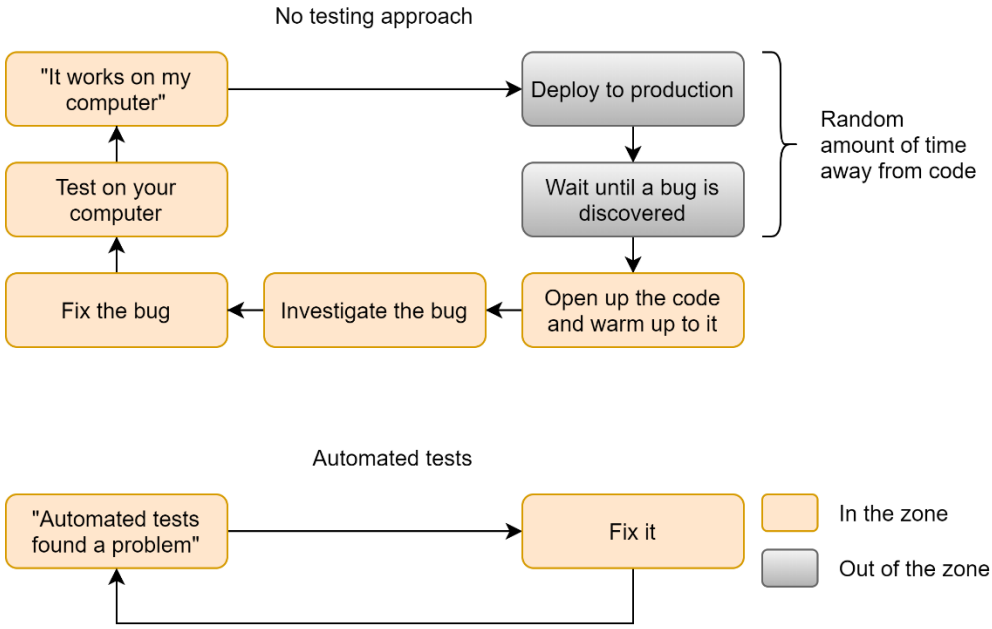


Figure 4.2 The expensive development cycle of "it works on my computer" versus "automated tests"

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.

©Manning Publications Co. To comment go to liveBook

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.

©Manning Publications Co. To comment go to liveBook

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

©Manning Publications Co. To comment go to liveBook

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
```

©Manning Publications Co. To comment go to liveBook

```
      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,

©Manning Publications Co. To comment go to liveBook

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.

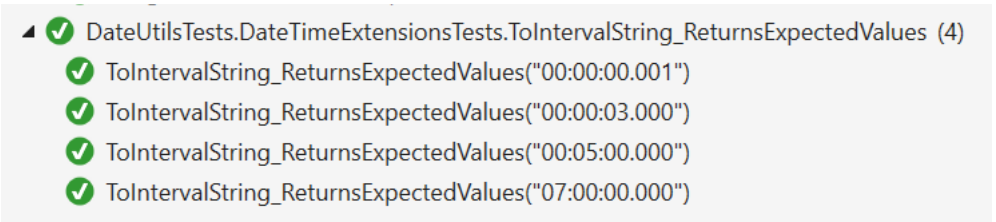


Figure 4.3 Test results that you can't take your eyes off of

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

©Manning Publications Co. To comment go to liveBook

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

©Manning Publications Co. To comment go to liveBook

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 as long as they're not 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.
- 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.

©Manning Publications Co. To comment go to liveBook

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,

©Manning Publications Co. To comment go to liveBook

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

©Manning Publications Co. To comment go to liveBook

```
[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

©Manning Publications Co. To comment go to liveBook

```
//[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.

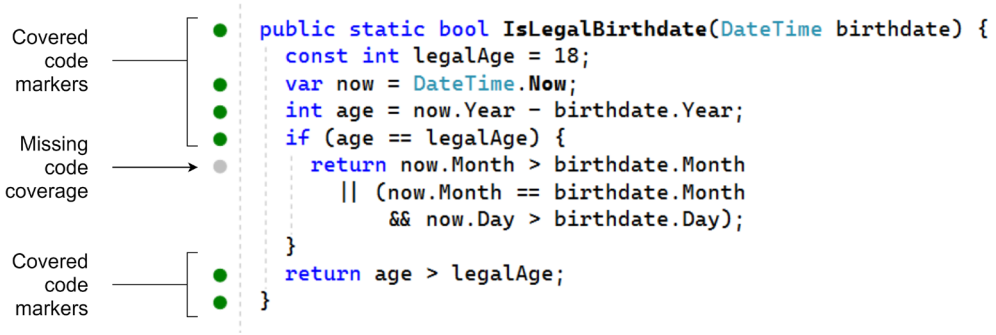


Figure 4.4 Missing code coverage

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

©Manning Publications Co. To comment go to liveBook

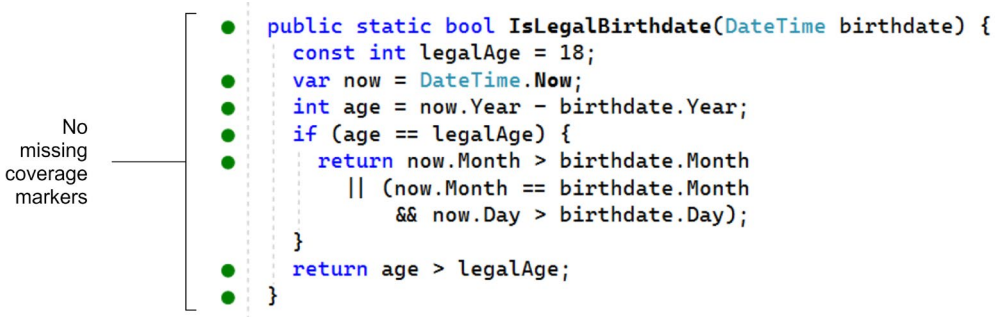


Figure 4.5 Full code coverage

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.

©Manning Publications Co. To comment go to liveBook

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.

©Manning Publications Co. To comment go to liveBook

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.

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"`

©Manning Publications Co. To comment go to liveBook

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")]
```

©Manning Publications Co. To comment go to liveBook

```
  [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

©Manning Publications Co. To comment go to liveBook

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 4.8, 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

©Manning Publications Co. To comment go to liveBook

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 have to be cast and converted into `int` if you happen to interact with standard .NET components with that type. You need to make sure that you're not digging yourself into a hole. 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 very 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, because, 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

©Manning Publications Co. To comment go to liveBook

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 parameters 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 and you get 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

©Manning Publications Co. To comment go to liveBook

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 throw your keyboard at

Kidding about the last one of course. Remember? You already greenlit that code in the code review. You have no right to throw keyboards at that person anymore. If anything, you both should be keyboarded. 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

©Manning Publications Co. To comment go to liveBook

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 4.3.

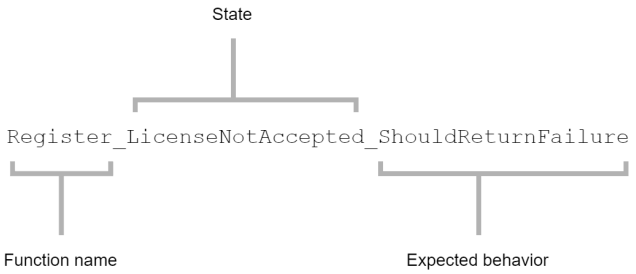


Figure 4.3 Components of a test name

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.

©Manning Publications Co. To comment go to liveBook