

THE IMPOSTER'S HANDBOOK, SEASON 1

A CS PRIMER FOR SELF-TAUGHT
PROGRAMMERS, EDITION 2.3.0

ROB CONERY

FOREWORD: SCOTT HANSELMAN

I never did a formal Computer Science course. My background was in Software Engineering. For a long time I didn't understand the difference, but later I realized that the practice of software engineering is vastly different from the science of computers.

Software Engineering is about project management and testing and profiling and iterating and SHIPPING. Computer Science is about the theory of data structures and $O(n)$ notation and mathy things and oh I don't know about Computer Science.

Fast forward 25 years and I often wonder if it's too late for me to go back to school and "learn computer science." I'm a decent programmer and a competent engineer but there are...gaps. Gaps I need to fill. Gaps that tickle my imposter syndrome.

I've written extensively on Imposter Syndrome. I'm even teased about it now, which kind of makes it worse. "How can YOU have imposter syndrome?" Well, because I'm always learning and the more I learn the more I realize I don't know. There's just SO MUCH out there, it's overwhelming.

Even more overwhelming are the missing fundamentals. Like when you're in a meeting and someone throws out "Oh, so like a Markov Chain?" and you're like "Ah, um, ya, it's... ah...totally like that!"

If only there were other people who felt the same way. If only there was a book to help me fill these gaps. Ah! Turns out there is. *You're holding it.*

Scott Hanselman @shanselman August 12, 2016 Portland, Oregon

Scott Hanselman has been computerizing, blogging, and teaching for many years. He works on Open Source .NET at Microsoft and has absolutely no idea what he's doing.

FOREWORD: CHAD FOWLER

I've been honored to be asked to write the foreword for several books over the course of my career. Each time, my first reaction is something like "Oh wow, how exciting! What an honor! HEY LOOK SOMEONE WANTS ME TO WRITE THEIR FOREWORD!!!"

My second reaction is almost always, "Oh no! Why me? Why would they ask me of all people? I'm just a saxophone player who stumbled into computer programming. I have no idea what I'm doing!"

No offense to Rob intended, but this may be the first foreword I feel qualified to write. Finally, a book whose very title defines my qualifications not just to write the foreword but to participate in this great industry of ours. A handbook for impostors. It's about time.

You know that friend, classmate, or family member who seems to waste too many precious hours of his or her life sitting in front of a computer screen or television, mouth gaping, eyes dilated, repetitively playing the same video game? In my late teens and early twenties, that was me. I made my living as a musician and had studied jazz saxo-

phone performance and classical composition in school. I was an extremely dedicated student and serious musician. Then I got seriously addicted to id Software's Doom. I played Doom all the time. I was either at a gig making money or at home playing the game. My fellow musicians thought I was really starting to flake out. I didn't practice during the day or try to write music. Just Doom.

I kind of knew how personal computers worked and was pretty good at debugging problems with them, especially when those problems got in the way of me playing a successful game of Doom death-match. I got so fascinated by the virtual 3D rendered world and gameplay of Doom that my curiosity led me to start learning about programming at around age 20. I remember the day I asked a friend how programs go from text I type into a word processor to something that can actually execute and do something. He gave me an ad hoc five minute explanation of how compilers work, which in different company I'd be ashamed to admit served my understanding of that topic for several years of professional work.

Playing Doom and reading about C programming on the still- small, budding internet taught me all I knew at the beginning about networking, programming, binary file formats (We hacked those executables with hex editors for fun. Don't ask me why!), and generally gave me a mental model for how computer systems hung together. With this hard-won knowledge, I accidentally scored my first job in the industry. The same friend who had explained compilers to me (thanks, Walter!) literally applied for a computer support job on my behalf. At the "interview", the hiring manager said "Walter says you're good. When can you start?"

So, ya, I really stumbled into this industry by accident. From that point on, though, I did a lot of stuff on purpose. I identified the areas of computer technology that were most

interesting to me and systematically learned everything I could about them. I treated the process like a game. Like a World of Warcraft skill tree, I worked my way up various paths until I got bored, intimidated, or distracted by something else. I covered a lot of territory over many hours of reading, asking questions of co-workers, and experimentation.

This is how I moved rather quickly from computer support to network administration to network automation. It was at this time that the DotCom bubble was really starting to inflate. I went from simple scripting to object oriented programming to co-creating a model/view/controller framework in Java for a major corporation and playing the role of “Senior Software Architect” only a few short years after packing the saxophone away and going full time into software development.

Things have gone pretty well since then, but I’ve never gotten over that nagging feeling that I just don’t belong here. You know what I mean? You know what I mean. You’re talking about something you know well like some database-backed Web application and a co-worker whips out Big-O notation and shuts you down. Or you’re talking about which language to use on the next project, and in a heated discussion the word “monad” is invoked.

Oh no. I don’t know what to say about this. How do I respond? How can I stop this conversation in a way that doesn’t expose me for the fraud I am? WHAT THE HELL IS A MONAD?

Haha, ya.

I hope that non-response made sense, you think as you walk toward the restroom, pretending that’s why you had to suddenly leave the discussion.

In daily work, I find myself to be at least as effective as the average programmer. I see problems and I come up with

solutions. I implement them pretty quickly. They tend to work. When performance is bad, I fix it. When code is hard to understand I find a way to make it easier to understand. I'm pretty good at it I guess.

But I didn't go to college for this stuff. I went to college and studied my ass off, but all I have to show for it is an extremely vast array of esoteric music knowledge that would bore the hell out of anyone who isn't a musician. In college you learn about algorithms. That sounds hard. When I write code, I don't use algorithms I think. Or do I? I'm not sure. I don't invoke them by name most of the time. I just write code. These college programmers must be writing stuff that's unimaginably more sophisticated since their code has algorithms!

And how can my code perform well if I didn't use Big-O notation to describe its performance and complexity characteristics? What the hell does "complexity" even mean in this context? I must be wasting so many processor cycles and so much memory. It's a miracle my code performs OK, but it does.

I think most of my success in the field of computer software development comes from my belief that:

1. A computer is a machine. In some cases it's a machine I own. I could break it into tiny pieces if I wanted.
2. These machines aren't made of magic. They're made of parts that are pretty simple. They're made in ways that tens of thousands of people understand. And they're made to conform to standards in many cases.
3. It's possible to learn about what these components are, how they work, and how they fit together.

They're just little bits of metal and plastic with electricity flowing through them.

4. Everything starts with this simple foundation and grows as simple blocks on top.
5. All of the hard sounding stuff that college programmers say is just chunks of knowledge with names I don't know yet.
6. Most of this stuff can be learned by a young adult in four years while they're also trying to figure out who they are, what they want to do with their lives, and how to do as little work as possible while enjoying youth.
7. If someone can learn all this stuff in just a four year degree, it's probably pretty easy to hunt down what they learn and learn it myself one concept at a time.
8. Finally, and most important, somehow I get good work done and it doesn't fall apart. All this stuff I don't know must be just a bonus on top of what I've already learned.

All this is just stuff you can learn! Wow. In fact, the entirety of human knowledge is just a collection of stuff that you can learn if you want to. That's a pretty amazing realization when you fully absorb it. A university doesn't magically anoint you with ability when you graduate. In fact, most people seem to leave university with very little actual ability and a whole lot of knowledge. Ability comes from the combination of knowledge, practice, and aptitude.

So, what separates us impostors from the Real Deal? Knowledge, practice, and aptitude. That's it. Knowledge is attainable, practice is do-able, and we just have to live with aptitude. Oh well.

Here's a big secret I've discovered: I'm not the only

impostor in the industry. The first time I met Rob, he interviewed me for a podcast. We ended up talking about Impostor Syndrome. On hearing my interview, several people told me “Wow, I feel like that too!” The more I dig, the more I think we all feel like that at some points in our careers.

So, welcome to Impostor Club! I’m kinda bummed now that I know it’s not as exclusive as I had thought, but it’s nice to have company I guess.

Anyway, now we have a handbook.

Ironically, reading and using this handbook might cause you to graduate from the club. If that happens, I wish you luck as you enter full scale computer software programmer status. Let me know how it feels. I’ll miss you when you’re gone.

Chad Fowler August 12, 2016 Memphis, Tennessee

Chad Fowler is the CTO of Wunderlist, which is now a part of Microsoft. He is also a Venture Capitalist with BlueYard Capital

PREFACE

The older versions of this book started off with an exhortation to dig deeper and do your best to inform yourself before arriving at that *strong opinion, weakly held*. I've had some time to think more about the idea, and I've decided to change the opening pages of this book entirely.

Strong opinions weakly held is just a euphemism for *willful ignorance*. That, friends, is a strong opinion and no, I don't hold it weakly.

A strong opinion, by definition, is one you believe firmly in and one that you hold tightly to, for whatever reasons you have. It's *your* opinion, after all. If I can talk you out of this easily, perhaps with a mild comment or slight rebuff, what was the point of you having it in the first place? Were you too lazy to consider other points of view? Too isolated in your thinking? Yes, and yes, and probably yes to every other reason.

Strong opinions are **annoying**. I'll go one step further: they're **toxic**. Let's sidestep politics, faith, and worldview and just focus on technology for now. Your reasons for having an opinion on anything should be exactly that: *reasons*. A thing

you reason using facts, as well as you can assemble them. “I think semicolons aren’t optional” is an opinion and for many a strong one and hopefully we can agree that we aren’t all going to agree.

Taking that idea a step further, however, and claiming that someone else is wrong because of your strong opinion is where it all falls apart. Shielding yourself from criticism by telling someone else they can talk you out of it if only they made sense... that’s toxic, friend.

I think knowing what you’re talking about is much more convincing. Facts are sexy! That’s why I wrote this book and why I’m talking about all of this upfront: *knowing what you’re talking about is always a good idea*. Education, what a drug.

I prefer to educate myself as much as I can because I loathe the idea that I’m defending my point of view based on what someone else said once or an article I read that told me what to think. If you’re reading this book I think we might both agree on that point!

Informing yourself *before* you reach an opinion is an indispensable skill if you want to advance in the tech industry. Don’t get me wrong, having a strong opinion is just fine (spaces instead of tabs, for instance) ¹ as long as you have solid reasons for your strong opinion. The keyword being *reason* instead of *feeling* or *belief*. Those are fine too, but tend to make technical discussions difficult indeed.

As we embark on this journey together, I would encourage you to embrace a different mantra: **informed opinions, happy to discuss**. As strong as your opinion is you should always leave the door open to have your mind changed. It’s a great first step to read up on and investigate your ideas as much as you can, but there really is no such thing as absolute truth; especially in the field of Computer Science.

Your experience will grow throughout your career and your knowledge will grow. **Let it.** Forming strong opinions stops this process from happening and your natural curiosity will ebb... and then you'll park yourself. This could be your happy spot, but eventually it will just be you and your opinions to keep you company.

Keep your mind open and ready. **There's so much to learn, and it's never, ever too late!**

— Rob Conery, Hanalei, Hawaii 2021

Code Samples

I have published this book with every different kind of editorial trickery possible when it comes to code samples. I've used embedded styling, rich text, and images. If there are other ways I'm sure I'll use those too! Each has their drawback - for instance embedding styled text will invariably be overridden by whatever reader you're using and images are horrible for accessibility.

I think I have something that will work, however. The editing software I'm using will embed alt text in images for accessibility - so that's what I've done. I want the code to look *precise*, indentations and all, for readability. No word wrapping, no font overrides.

I also want you to be able to play along so each code example (that merits it) will come with a link to either a Gist or a file up on GitHub. If you're wanting to play along, please do - but it will involve you clicking a link in the book.

Cover Photo Credit

The image used for the cover of this book (and the inspiration for the splash image on the website) comes from NASA/JPL:



The image is entitled HD 40307g, which is a “super earth”:

Twice as big in volume as the Earth, HD 40307g straddle the line between “Super Earth” and “mini-Neptune” and scientists aren’t sure if it has a rocky surface or one that’s buried beneath thick layers of gas and ice. One thing is certain, though: at eight times the Earth’s mass, its gravitational pull is much, much stronger.

David Delgado, the creative director for this series of images describes the inspiration for HD 40307g’s groovy image:

As we discussed ideas for a poster about super Earths – bigger planets, more massive, with more gravity – we asked, “Why would that be a cool place to visit? We saw an ad for people jumping off mountains in the Alps wearing squirrel suits, and it hit us that this could be a planet for thrill-seekers.

When I saw this image for the first time I thought this is what it felt like when I learned to program. Just a wild rush of freakishly fun new stuff that was intellectually challenging

while at the same time feeling relevant and meaningful. I get to build stuff? And have fun!?!? Sign me up!

This is also what this last year has been like writing this book. I can't imagine a better image for the cover of this book. Many thanks to NASA/JPL for allowing reuse.

INTRODUCTION

So what is it you do again? Work with computers right?

Yes, that's right. I work with computers. I'm a programmer, a software developer, and a geek, among other things.

But what do you do, exactly?

Well... I ... I am actually a GOD

So much for pointless smalltalk. If you ever want to make someone doubt your sanity, assert that you're some kind of deity. It does the trick well. I hate smalltalk and I keep this little dandy in my back pocket for those occasions where I need to exit quickly. Which is often.

You know what else? *You're a god too!* This is going to take a bit to explain and yes, I have reasons backing this opinion believe it or not! Every developer builds their own little

universe that follows the same laws that our own physical universe. Who's to say you're not creating little universes on a daily basis?

You can't prove a negative, and I doubt anyone can prove or disprove that we're part of a fractal chain of simulations. It is, however, the only thing that kind of makes sense when you're asked...

What Do You Do?

What's your answer to this? "Write code? Work at company X on Product Y?" Each of these is a fine answer, of course, but let me ask you, programmer to programmer: *what do you do?* I don't mean literally (as in "what project are you working on?"), I mean in a more grounded sense. After all a computer is just a tool like a hammer is a tool. A carpenter can point to a chair and say "I make furniture out of wood using these tools". I think that's a fine answer, and straight to the point.

I make applications out of code using a code editor.

I don't know about you, but that sentence falls a bit flat. Sure it's true in some senses, but I don't think it quite captures what you and I do daily. As you can tell I've been pondering this question for a while and I think I've hit on something more compelling: *I tell computers what to do.*

I like that answer because it's true. I don't build things, not in the same way a carpenter assembles wood into something beautiful and useful. I issue instructions to a CPU, store answers in RAM and, occasionally, have them written to a physical medium of some kind so those answers can be used later on.

Computers don't do this on their own, we need to be involved. They're simply machines and need our instruction. A script with direction, scene, and plot with a storyline that

has a hero, a villain, and a resolution in the 5th act. That's the way I see the art of computation. It's more fun that way. You might see this process completely differently, which is fine.

We could argue about this analogy for a very long time but let's cut to the point: *we don't have to*. You're free to create as you see fit and so am I. Every great play, however, is based on agreed-upon structures and conventions. They might bend a few things here and there but overall, you can typically rely on the idea of *beginning, middle and end* as well as the notions of plot, character, tone and pace, among others.

We can translate these ideas to the world of programming. We assemble our instructions to the CPU in files, classes, modules, and libraries. We move them through processes according to events and ultimately provide some type of answer.

But how is this all done? What is this black art? Who decided on what should go where how and when?

These are good questions and the good news for you is that the answers, hopefully, are held in your hand right now. I've done my best to assemble the theory and application of our industry as a whole into something approachable and immediately usable. I've done this in an effort to enrich our dialog, to have more opinions which are informed instead of strong with a "change my mind" poster next to them.

There's a rich history here, from understanding problem complexity to mathematical frameworks used to implement a solution. There are systems of design wherein we compute things the way the ancient Egyptians built pyramids or a school of fish evade a shark. Trade skills that mimic those of blacksmiths from medieval times and chests full of magical scripts waiting to be explained.

These await you... in due time. And I hope you have enough to give because all steps must be taken in the correct

order. It wouldn't be wise to discuss dynamic programming, after all, unless you understood the data structures and algorithms that propelled it. Algorithmic efficiency makes no sense unless you understand Big-O which, itself doesn't make sense unless you understand the essentials of complexity theory.

So let's start at the very beginning - a perfect place to start. Back when computers weren't machines at all... they were *people*...

The First Computers

Algorithms have existed for millennia, but to use them you had to do things manually, which is error prone. People began writing the solutions to complex problems in books, called "Mathematical Tables". These people were called "computers" and in 19th and 20th centuries their jobs were given to machines which were also called "computers". This is the story of how it happened.

This might sound absurd, but stay with me. In April of 2016, scientists and philosophers gathered in New York to discuss that very question:

The idea that the universe is a simulation sounds more like the plot of "The Matrix," but it is also a legitimate scientific hypothesis. Researchers pondered the controversial notion Tuesday at the annual Isaac Asimov Memorial Debate here at the American Museum of Natural History.

Moderator Neil deGrasse Tyson, director of the museum's Hayden Planetarium, put the odds at 50-50 that our entire existence is a program on someone else's hard drive. "I think the likelihood may be very high," he said.

The problem with well-meaning articles like this one is that they tend to go for the low-hanging fruit, dumbing-down the core of what should otherwise be a compelling idea – simply to appeal to a mass readership.

Which is unfortunate, because the question has a solid foundation: the physical universe is indeed computed. The very nature of the physical universe describes a progressive system based on rules:

- **Cause and effect:** what you and I think of as “conditional branching”
- **Consistent, repeated patterns** and structure: the magical numbers π , ϕ , and e (among others) hint at a design we can only perceive a part of. Indeed, Plato suggested that the world we see is just an abstraction of its true form.
- **Loops:** the two ideas above (cause and effect, pattern repetition) interact repeatedly over time. A day is a repeated cycle of light and dark, life is the same way – as is a planet orbiting the sun, the sun orbiting the center of the galaxy.

This has the appearance of what we think of today as a program – either one massive one or many small ones interacting. Let’s take a look at some we find in nature.

Nature’s Strange Programs

At this point you might think I’m probably taking this whole idea a bit too far – but I’d like to share a story with you about Cicadas.

I was watching a fascinating show a few weeks back called *The Code*, which is all about math and some interesting patterns in the universe. At one point it discussed a small

insect that lives in North America, the magicicada, which has the strange trait of living underground for long periods during its young life.

These periods range from 7 years to 17 years although the most common periods are from 13 to 17 years. That's a very long time, but that's not the most surprising thing about them. The strangest aspect of these creatures is that they emerge from the ground on schedule: 7 years, 13 years, or 17 years.

Those are prime numbers. This is not an accident; it's a programmed response to a set of variables. Trip out, man...

The emergence, as it's called, is a short time (4-6 weeks) that the cicadas go above ground to mate. They sprout wings, climb trees and fly around making a buzzing noise (the males) to attract a mate. Why the prime number thing? There are a few explanations – and this is where things get strange.

Predatory Waves

We know that predators eat prey, and each is constantly trying to evolve to maximize their chances of staying alive. Cats have remarkable hearing, eyesight and stealth whereas mice counter that with speed, paranoia, and a rapid breeding rate. You would think this kind of thing balances, but it doesn't.

It comes and goes in waves as described in this article from Scientific American:

As far back as the seventeen-hundreds, fur trappers for the Hudson's Bay Company noted that while in some years they would collect an enormous number of Canadian lynx pelts, in the following years hardly any of the wild snow cats could be found ... Later research revealed that the rise and fall ... of the lynx population correlated with the rise and fall of the lynx's favorite

food: the snowshoe hare. A bountiful year for the hares meant a plentiful year for lynxes, while dismal hare years were often followed by bad lynx years. The hare booms and busts followed, on average, a ten-year cycle...

A recent hypothesis is that the population of hares rises and falls due to a mixture of population pressure and predation: when hares overpopulate their environment, the population becomes stressed

... which can lead to decreased reproduction, resulting in a drop in next year's hare count.

This much makes sense and isn't overwhelmingly strange ... until this theory is applied to the cicada:

Now, imagine an animal that emerges every twelve years, like a cicada. According to the paleontologist Stephen J. Gould, in his essay "Of Bamboo, Cicadas, and the Economy of Adam Smith," these kinds of boom-and-bust population cycles can be devastating to creatures with a long development phase. Since most predators have a two-to-ten-year population cycle, the twelve-year cicadas would be a feast for any predator with a two-, three-, four-, or six-year cycle. By this reasoning, any cicada with a development span that is easily divisible by the smaller numbers of a predator's population cycle is vulnerable.

This is where the prime number thing comes in (from the same article):































Prime numbers, however, can only be divided by themselves and one... Cicadas that emerge at prime-numbered year intervals ...

would find themselves relatively immune to predator population cycles, since it is mathematically unlikely for a short-cycled predator to exist on the same cycle. In Gould’s example, a cicada that emerges every seventeen years and has a predator with a five-year life cycle will only face a peak predator population once every eighty-five (5×17) years, giving it an enormous advantage over less well-adapted cicadas.

Who would have thought a tiny insect could master math in this way?

Overlapping Emergences

Another fascinating theory behind the prime-numbered emergence of these cicadas is the need to avoid overlapping with other cicada species. We’re dealing with prime numbers here, and it just so happens that the multiples of 13 and 17 overlap the least of any numbers below and immediately after them:

CICADA EMERGENCE																			
1	2	3	4	5	6	7	8	9	10	11	12		14	15	16				
18	19	20	21	22	23	24	25		27	28	29	30	31	32	33				
35	36	37	38		40	41	42	43	44	45	46	47	48	49	50				
	53	54	55	56	57	58	59	60	61	62	63	64		66	67				
69	70	71	72	73	74	75	76	77		79	80	81	82	83	84				
86	87	88	89	90		92	93	94	95	96	97	98	99	100	101				
103		105	106	107	108	109	110	111	112	113	114	115	116		118				
120	121	122	123	124	125	126	127	128	129		131	132	133	134	135				
137	138	139	140	141	142		144	145	146	147	148	149	150	151	152				
154	155		157	158	159	160	161	162	163	164	165	166	167	168					
171	172	173	174	175	176	177	178	179	180	181		183	184	185	186				
188	189	190	191	192	193	194		196	197	198	199	200	201	202	203				
205	206	207		209	210	211	212	213	214	215	216	217	218	219	220				

Natural Computation

This is natural computation, there is simply no getting around that. There’s nothing magical or mystical about what

these cicadas do – they’re adhering to the patterns and structure of physical world.

Bernoulli’s Weak Law of Large Numbers states that **the more you observe the results of a set of seemingly random events, the more the results will converge on some type of relationship or truth.** Flip a coin 100 times, you’ll have some random results. The more you flip it, the more your results will converge on a fifty-fifty distribution of heads vs. tails.

We’re seeing the same thing with cicada emergences. After millions and millions of years of evolution, a natural emergence pattern is resolving that is based on math and the need to avoid overlap. We’re seeing the computational machinery behind evolution itself, revealed by the Weak Law of Large Numbers. Prime number distribution in two entirely different population controls (predators and emergence).

Our Instructions

Our physical world is the result of a massive set of experiments. Take you, for instance. The odds of your existence are astronomically low... yet here you are, reading this book.

For you to exist, your parents had to meet at the precisely correct time and conceive you at precisely the correct moment. There are nearly countless reproductive cells in both male and female bodies – and out of all of them, two of them had to meet to make you.

That process was repeated with your parents’ parents, on down the line to cave men... and further to the very first organism to flash into existence out of an electro-chemical reaction that took place at exactly the right moment.

You are dust exploded from the heart of the Big Bang! And who can say where the Big Bang came from? Many

scientists believe it's an ongoing expansion and contraction of matter and energy that has been going on continuously.

As you can see, there really is no end to pursuing the crazy odds of you being where you are right now and the only thing without an end that we can figure is a circle. In programming terms, it's *a loop*. Endless expansion and contraction of the universe. An infinite loop which allows cause and effect (aka “events”) to happen, which cause other events to happen.

In other words: *a running program*. Writing code is only the first step - running it is the next and in order to run it you need a *run loop* which waits for some type of input and a reason to exit. A set of instructions that keeps repeating until an exit is triggered. You type “node index.js” and a binary Big Bang goes off, bringing your small computational universe to life.

The von Neumann Automata

If we're programs in a simulation, then we're programs writing programs, what a concept! And an old one too. In 1946, John von Neumann theorized that we would eventually create computers which replicated themselves ¹:

One problem von Neumann posed and essentially solved was: What kind of logical organization is sufficient for an automaton to control itself in such a manner that it reproduces itself? He first formulated this question in terms of a kinematic automaton system, and later reformulated and solved it in terms of a cellular automaton system.

If you've read any of the Bobiverse series, these ideas will be familiar to you. In fact the idea of self-replicating

machines (and processes) has been explored relentlessly over the last few decades.

Our First Paradox

We've arrived at our first paradox, and right on time! **How can a program contain the instructions for itself?** Do those instructions contain themselves too? This is a variation of "does a set of all sets contain itself?" ².

We might be tempted to think of a "fan-out" process, where additional functions are created to concurrently handle a processing load. But that's a managed thing, usually by an operating system. It's not a program literally replicating itself.

How would you write a self-replicating routine? You can't use a process manager - that's cheating. How would you go about creating a clone routine that clones itself? Do you think this is possible? **Would your code contain itself?** Maddening!

We Are Driven to Compute Things

Human beings have understood that there is some process at work in the natural world, and they've struggled to express it in a way other than mysticism and spirituality. This is a tall order: how do you explain the *mechanics* behind the physical world?

Philosophers, mathematicians, and logicians throughout history have been able to explain highly complex processes with equations and numbers. The Sieve of Eratosthenes, for example, is a simple algorithm for finding all the prime numbers in a bound set ³. It was described at around 250 B.C.

Mathematicians, scientists, and engineers have been trying to take raw computation out of our brains and into a

machine of some kind for centuries and you, friend, are continuing this wonderful process.

Let's have a look at your professional family tree.

Computation in the Steam Punk Age

If I were to ask you what is the square root of 94478389393 – would you know the answer? It's unlikely – but if you did know the answer somehow, it would be due to some type of algorithm in your head that allows you to step through a process of deduction. Or maybe you have eidetic memory?

If you did have a photographic memory, you would be in luck if you lived a few hundred years ago. Before we had calculators, going all the way back to 200BC, people used to write mathematical calculations in books called mathematical tables. These were a gigantic pile of numbers corresponding to various calculations that are relevant in some way. Statistics, navigation, trajectory calculations for your trebuchet – when you needed to run some numbers you typically grabbed one of these books to help you.

The problem was that these books were prone to error. They were created by human computers, people that sat around all day long for months and years on end, and just figured out calculations. Of course, this means that errors can sneak in, and when they did, it took a while to find the problem.

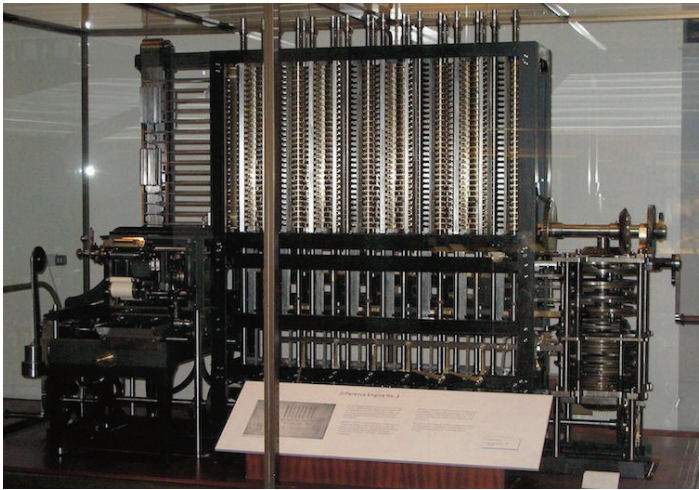
Errors like this are annoying and, in some cases, deadly. Ships were reported to have run aground because of errors in navigation tables – which were traced to errors in the mathematical tables used to generate them.

Charles Babbage, a mathematician, philosopher, and engineer decided that it was time to resolve this problem. The industrial revolution was in full swing and machines were changing humanity at a rapid pace. Babbage believed they

could also change mathematics by removing the need for a human being to be involved in routine calculations.

The Difference Engine

In the early 1820s Babbage designed The Difference Engine, a mechanical computer run by a steam engine. His idea was that, through a series of pulleys and gears, you could compute the values of simple functions.



Difference Engine #2. Babbage conceived a number of machines, none of them were completely built, however. The machine you see here was built by the London Museum of Science in the 1990s based on Babbage's plans. Image credit: Computer History Museum.

Babbage's machine could be "programmed" by setting up the gears in a particular way. So, if you wanted to create a set of squares, you would align the gears and start cranking the handle. The number tables would be printed for you and, when you were done, a little bell would ring.

Babbage had found a way to rid mathematical tables of errors, and the English government was all over it. They gave him some money to produce his machine, but he only got to part of it before the project imploded. After 10 years of

designing, redesigning and arguing with other scientists – the government pulled the plug.

Which was OK with Babbage, he had another idea.

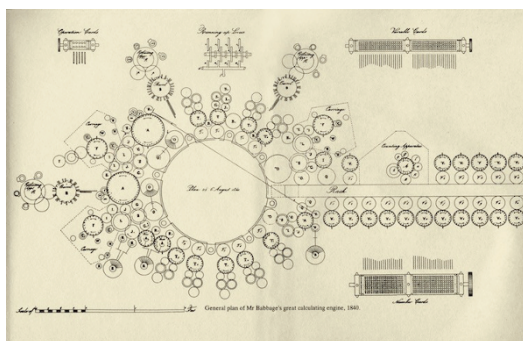
The Analytical Engine

Babbage believed his difference engine could do more. He was inspired by the Jacquard Loom, which was a programmable loom that could create complex patterns in textiles. The loom used a series of pins moving up and down to direct the pattern – and Babbage thought he could use the same idea for his machine using punch cards.

The idea was simple: tell the machine what to do by punching a series of holes in a card, which would, in turn, affect gear rotation. An instruction set and the data to act on – a blueprint for the modern computer.

According to the plans, this machine had memory, could do conditional branching, loops, and work with variables and constants:

The programming language to be employed by users was akin to modern day assembly languages. Loops and conditional branching were possible, and so the language as conceived would have been Turing- complete as later defined by Alan Turing.



Plans for the Analytical Engine. Image credit: Computer History Museum

Babbage knew he was onto something:

As soon as an Analytical Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then arise: By what course of calculation can these results be arrived at by the machine in the shortest time?

The interesting thing is that Babbage was focused on mathematical calculations. Someone else realized his machine could do so much more.

Ada Lovelace

Ada Lovelace is widely regarded as the very first programmer, although some contend that this statement is not only arguable, it also undermines her true importance: understanding the true effect of Babbage's Analytical Engine.

She was a brilliant mathematician and, interestingly, was the daughter of Lord Byron. In 1833 she met Babbage and instantly recognized the utility of what he wanted to build.

They worked together often, and she helped him expand his notions of what the machine could do.

In 1840 Lovelace was asked to help with the translation of a talk Babbage had given at the University of Turin. She did, and in the process added extensive notes and examples to clarify certain points. One of these notes, Note G, stood out above the others:

Ada Lovelace's notes were labeled alphabetically from A to G. In note G, she describes an algorithm for the Analytical Engine to compute Bernoulli numbers. It is considered the first published algorithm ever specifically tailored for implementation on a computer, and Ada Lovelace has often been cited as the first computer programmer for this reason.

Diagram for the computation by the Engine of the Numbers of Bernoulli. See Note G. (page 722 of seq.)

Lovelace's Note G. Image Credit: sophiararebooks.com

Besides Bernoulli's numbers, she added this gem:

The Analytical Engine might act upon other things besides number, were objects found whose mutual fundamental relations could be expressed by those of the abstract science of operations, and which should be also susceptible of adaptations to the action of the operating notation and mechanism of the engine... Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent.

Some historians question how much Lovelace contributed to the first programs written for Babbage's Analytical Engine. We'll never really know the true answer to this question, but we do know that Lovelace thoroughly understood the potential power of this machine.

What Can We Compute?

This book is about you and I telling a computer how to compute something, much like Ada Lovelace thought we could (and should). We practice the art and science of computation as have so many before us, using nothing but their minds, applying their philosophies and science, and eventually using machines they produced.

Which begs the question: *what exactly can a machine compute that we cannot?* I'm not talking about speed - I'm talking about *possibility*. This is simple to reason: a machine can't do anything on its own. We need to tell it what to do, therefore a machine can only compute what we tell it to compute.

The question then becomes simpler: *what can we compute?* Let's say I hand you a pencil and some paper and say "I'm

going to ask you some questions, please compute the answer using this pencil and paper⁴". Before we start, however, I ask you to ponder *what* you're capable of computing.

Your answer might be like mine: *WTF? I guess I can compute anything I know how to given enough time*, which is a perfect answer. This is *basically* correct, but the human mind is only capable of so much. For instance: think of the biggest number you can keep in your head at one time. Yes, you can say "all the grains of sand on the beach" or "stars in the universe", but these are just labels for something you simply cannot fathom.

If I asked you to consider 10 apples, for instance, you could easily picture this concept in your mind. We could do math with those 10 apples (after my kid eats one or two) and you would immediately know we had 8 left. 1000 apples is still doable, though a bit harder to visualize. It's at that point that we need to consider that the human mind is only capable of so much:

Early research found the working memory cut-off to be about seven items, which is perhaps why telephone numbers are seven digits long (although some early telephone dialing started with a two- or three-letter "exchange," often the first letters of a community name, followed by four or five figures, e.g. PENnsylvania 6-5000). Now scientists think the true capacity is lower when people are not allowed to use tricks like repeating items over and over or grouping items together.

"For example, when we present phone numbers, we present them in groups of three and four, which helps us to remember the list," said University of Missouri-Columbia psychologist Nelson Cowan, who co-led the study with colleagues Jeff Rouder and Richard Morey.

"That inflates the estimate. We believe we're approaching the estimate that you get when you cannot group. There is some controversy over what the real limit is, but more and more I've found people are accepting this kind of limit."

MIND'S LIMIT FOUND: 4 THINGS AT ONCE,
LIVESCIENCE

Even with the use of a pencil and paper, our brains have limits. Computers have limits too, but far greater than ours. Computers should be able to compute anything, really, but the problem is *us*. Computers do what we tell them, which means we have to understand what we're telling them first.

The Tools of Computation

Alan Turing and Alonzo Church, whom we'll meet in a few chapters, pondered the idea of machine computation deeply. They both realized, somewhat independently, that a machine can compute "anything computable", which is a roundabout way of saying "if the human mind can figure out a solution, a machine can solve it easily".

That little nugget is called the *Church-Turing Conjecture* and is a cornerstone of Computer Science. We'll read about it more later and yes, I know it sounds pretty obvious but back then, in the 1930s, people had no idea what was possible with a machine vs. a human mind. The only thing they knew for sure was that a machine was faster. A lot faster. All we needed to do was to have a way to tell the machine what to do.

This is where Alonzo Church comes in (and later, Alan Turing). He formulated a symbolic calculus that any

computer could use to compute anything. It's based on functions and is mind bending in elegance and power. You'll learn all about it in the chapter on *Lambda Calculus*.

Alan Turing conceived of a machine that consisted of a head which could read and write to a tape of infinite length. The head could calculate things using the information on the tape, writing those calculations back to the tape for later calculations. The *Turing Machine* is the conceptual model upon which computers are built.

So, we have abstract machines and math that will help us compute things using a machine. Can mathematics also tell us what's possible to compute with a machine?

It turns out, **yes**.

Problems in the Natural Set

Every program written for a machine was, at one time, compiled down to instructions that a CPU could understand. These instructions, as I'm sure you're aware, are literal 1s and 0s – binary.

The fascinating thing is that those 1s and 0s also represent a binary digit. A *number*, which tells us something fascinating: *every program ever created represents a natural number*. Put another way: **programs are natural numbers**.

If you, like me, need a definition for a natural number, here you go ⁵:

The whole numbers from 1 upwards: 1, 2, 3, and so on ... Or from 0 upwards in some fields of mathematics: 0, 1, 2, 3 and so on ... No negative numbers and no fractions.

Another definition holds that **natural numbers are**

whole numbers, without 0, that represent a *bound set*. The amount of toilet paper at the store, for instance, is a bound set. If there is 0 toilet paper at the store, the set doesn't exist. Similarly, there can only be so much toilet paper at the store ⁶! This is a bound set and therefore toilet paper supply is a natural number.

The same can be said of the programs we write. If you have 0 bits in your program I would argue that you don't have a program at all, and I think you'd agree with me. Similarly, your hard drive – indeed the entire hard drive capacity of the planet – is capped at some huge number. It's simply not possible to write a program larger than the hard drive space (and RAM) available to us. This, again, is a bound set and therefore our programs represent natural numbers.

The problems they solve, however, are quite different. A program receives input, does something with it (aka “computes”) and then provides output. That output, at some point, is also a 1 or a 0 which, through the process of encoding, will be handed to you as something you recognize such as true, false, or the number 42.

Let's use a simple example to illustrate this interesting point. Let's say I have a program that tells me if a number is even and I want to check every possible number, to infinity and beyond! I start at 1 and keep going...

What I end up with is a very, very long set of 1s and 0s, without end. If I assemble that output into a long binary string, it will represent a number that keeps growing *ad infinitum* as long as my program is running. The input, likewise, is infinite: it's every number that exists.

This tells us something interesting.

The *solutions* to the problems we solve using a program we write belong in an *unbounded* set of numbers, if we're thinking in terms of binary. If the solutions are unbounded then, correspondingly, so are the problems. Unbounded

numbers are also said to be *real* numbers and the set of real numbers is far, far, far greater than the set of natural numbers ⁷.

The problems belong in an infinite set, as do their solutions. Our programs, however, are stuck in the finite. The extreme shallow end of the complexity pool.

It's a simple, mathematical truth: *we're incapable of writing programs that can solve more than the most basic problems*. By "most basic problems" I mean those that can be represented programmatically by a natural number. We need to get into the realm of real numbers before the good work begins!

But how can we possibly do this? To me, the simple answer is to remove ourselves from the process of solving the problem and let the machine do the work. That's a scary proposition, but seems to be our only way forward.

Do me a favor and watch this wonderful video on *Ant Colony Optimization*. What you're seeing in this video is the power of *machine learning* and evolutionary algorithms. Spawning, learning, replaying. This is an evolutionary approach to solving a problem. Much like life itself. In fact it's *precisely* like life. What if we could use von Neumann's Automata to create programs that replicated themselves as needed, simultaneously solving smaller problems in parallel on their way to an overall solution?

That would be the deep end of the complexity pool and a scary proposition for a lot of us.

The really tough problems are *decision* problems, with variable outcomes. We'll see more of these in the chapter on Complexity Theory but, as mentioned, these decision problems far, far outnumber the paltry problems we're currently solving with computers. Imagine releasing Skynet on a Chess game, letting it spawn as many copies of itself as it needed in order to solve the board in milliseconds. If that happens, P will in fact equal NP⁸!

We'll learn all about P, NP, and problem complexity in a few chapters, which is critical when you sit down to write an algorithm. If you're trying to solve something that's too complex (like, say, solving a chess board), you might get fired like I did. Let's prevent that shall we?

For now, let's bring this introduction back down to Earth.

Creating A Better Solution

If you understand how complex a problem can be you can solve it in a much more straightforward fashion... but you need the right tools and a solid understanding of how to use them. As you'll come to see, we spend a lot of time in theory in this book and, towards the end, we get into practice.

Right now you're stuck in the philosophical build up, but I think that's important too.

Not many people like to spend time digging into theory and I can understand that. I like to learn by doing but, in so many cases, you end wasting a TON of time.

For instance: trying to understand functional programming without becoming familiar with Lambda Calculus is doable and usually sufficient, but you'll miss out on the entire computational model that underpins all programming. You certainly don't need to understand internal combustion to drive a car, but it can really help if your car stops running on the highway in the dead of night.

The same can be said of Big-O notation. It's something people get to know when it's time to interview, but there's so much more to it!

An example we'll get into in a later chapter is that of indexing a database. You probably know that if a query slows down, you might need an index. You could study log files or maybe use a monitoring system that tells you straight up to pop an index on that customers table.

But if you were asked what kind of performance gain can be expected - do you know the answer? If you know the difference between Order n and Order $\log n$, then you can give a clear answer to this question!

A Snowball, Hopefully

It's my sincere hope that you spend some time in the early chapters of this book, learning about computational machines, algorithms and data structures, and lambda calculus. These core concepts will add so much weight to the later chapters and will hopefully create a "snowball effect", with you in accreting knowledge as you tumble down the wonderful hill of Computer Science.

By the time you hit the testing section later on, you'll understand why testing is so important when it comes to properly solving problems and reducing complexity. The Unix section will show you small ways to improve your overall coding experience, putting you back on the problem and away from mechanical nonsense.

Onward!

This book has been in publication for the last 5 years and as of a month ago, it's sold right around 30,000 copies and helped tens of thousands of people dig deeper into their daily tasks, finding inspiration and happiness in what they used to consider tedious and repetitive.

We're building universes, after all, what could be more fun! Let's move on now and see how we can be better world builders.

COMPLEXITY THEORY

WE SOLVE PROBLEMS. SOME ARE MORE
INTERESTING (AND COMPLEX) THAN OTHERS.

Complexity Theory deals with how difficult a given problem is. There are multiple classifications (P, NP, NP-Complete, NP-Hard, Exp, etc) that identify classes of problems based on the idea of time complexity. Over the years complexity theorists have found that many problems are related in some abstract way, and can be reduced, quite interestingly, from one to another.

As a programmer, you solve problems. You should know if the problem is considered easy, hard, or impossible. In addition, you should be able to identify the type or classification of a problem that you're trying to solve.

EVERY SUMMER I get together with friends from college for the weekend. We'll rent a cabin in the mountains, go on hikes, fish and try not to hurt ourselves playing American football. I look forward to it every year; it's the only chance I get to see most of these people.

I'm sure you have a group of friends just like this one.

People you've known for a very long time. Seeing them is always a treat... deciding what you want to do when you see them... well that's when things get more than a bit complex.

Making decisions as individuals is hard enough. Adding more people to the decision process takes something that was hard and makes it feel almost impossible.

Consider this scenario: my friends and I have just returned from a day fishing for trout in a high mountain lake. We're tired, hungry, and more than a little thirsty. We've cleaned ourselves up and someone asks:

So, anyone up for going somewhere for a pint?

There are 6 of us standing in the same room. I'm sure the scene isn't hard to imagine – we look from one to the other, trying to gauge what our answer will be. Someone says “sure”, another says “yeah where?” and a third says “I don't drink but I'm happy to tag along if I can find something to eat”.

Uh oh. Group inertia!

You've had these discussions, I'm sure. They're not very fun because it's difficult to figure out what's going to work out the best for everyone; there are just too many parameters to consider. Joe doesn't like bitter beers, Kim wants a cocktail, Kevin doesn't drink but is craving a pretzel and Olga wants to go to her favorite pub 20 miles away because she knows the bartender and can get us a discount.

These situations are usually resolved by one person (typically the loudest) suggesting that everyone get in the damn car, we'll start out at place X and move on from there. This approach works remarkably well for for a small group of

people. If the group were to grow, however, that's when things get complex.

What do I mean by *complex*, however? In normal conversation we could just assume that this means "it's really hard to figure out". As programmers, however, we should have a more precise way of thinking about complexity. We deal with it every day – it's what we do. When your boss (or client) asks you the Big Question: how hard is it to add feature X?, how do you respond?

If you're like me, you might typically say "I'll have a look and let you know". Maybe you've done feature X before and you might say "it's doable".

Shouldn't we have better words for this conversation? Perhaps a way to more clearly define exactly how complex a given problem might be? Yes. We, as programmers need to be able to do this. The ability to clearly understand the complexity of a problem can save your boss or client tons of time and money. You might even avoid getting fired... like me... which is something I'll explain in more detail in just a few minutes.

Let's head back into the hills and see if we can figure out where my friends and I can go grab a pint and a bite to eat. Our situation is getting complicated, let's see if we can figure out just how complicated it's getting.

Simple Solutions And Polynomial Time (P)

My 6 friends and I are standing in the middle of the room, staring at each other, trying to figure out where we're going to go. People are stating their criteria and asking questions at the same time, and it's turning into a bit of a muddle.

Aaron, whose cabin we're staying at, speaks up and says:

Here's a list of the two pubs in town, and Olga's favorite pub 20 miles away. Put a check next to the one you want to go to. We'll go to the one with the most checks.

This type of solution is simple to implement and, best of all for the 6 of us, happens in a reasonable time frame. What's a reasonable time frame you ask? Good question! The answer is a bit abstract, but important nonetheless: it's less than the amount of time that would prevent us from doing it in the first place.

If deciding on a pub took us 15 minutes, we might get a bit frazzled, but we'd still do it. 30 minutes and we're grumpy for sure – anything more than that would be anarchy.

Thinking about this in terms of programming, we can think of “a reasonable time” as the time we might consider it acceptable and relevant to execute a routine for a given set of inputs. Sorting 100 things should come back in less than a millisecond, for instance. Sorting 1,000,000 things we might be willing to wait for a second or two.

Crunching analytics on billions of records, we might wait a few hours. Indexing a huge corpus of free text might take an entire night, or possibly a few days!

Using Aaron's democratic sorting process for selecting a pub is, best of all, fast. It also has another advantage: it scales really well. If our group of 6 turned into a group of 12, the process would take twice as long. A group of 15 would take 2.5 times as long.

Notice that I'm talking about the complexity of our democratic sorting process using time? This is the key to understanding Complexity Theory: you think about complexity in terms of time as you scale the inputs that go into the algorithm that you're using to solve the problem.

We can describe the way our democratic sorting process will scale using a bit of math:

$$T = 2x$$

Where T is the overall process time in minutes and x represents the number of friends. The equation that describes this scaling is a very simple one, and if you're a mathematician, you would call this type of equation a polynomial equation:

In mathematics, a polynomial is an expression consisting of variables and coefficients which only employs the operations of addition, subtraction, multiplication, and non-negative integer exponents. An example of a polynomial of a single variable x is $x^2 - 4x + 7$. An example in three variables is $x^3 + 2xyz^2 - yz + 1$.

When the complexity of an algorithm scales according to a polynomial equation, mathematicians say that it scales in “P time”, where P stands for *polynomial*. These algorithms are the simplest and often the most boring to work with. List operations, for example, are P time algorithms. Things like sorting, searching, zipping and enumerating all happen in P time.

Complexity Theory, however, isn't exactly concerned with the algorithms that are used to solve a problem. It focuses on the problems themselves. We wouldn't say that our democratic sorting process executes in P time, that only matters to

the programmers. A mathematician is much more concerned with the problem itself, and how difficult it is to solve. If one were sitting next to me, she would say that our pub selection problem is “in P”.

In more concrete terms, P is a complexity class that describes the set of all problems solvable in P time. Things like sorting and searching arrays, arranging your sock drawer and finding your cat who’s been wandering the neighborhood.

In other words: *simple problems*. These tend to be less interesting and not typically what programmers want to spend their time solving. Have you ever had to create a sorting algorithm by hand? I have! I had to do it for this book – and I’m going to make you do it with me in a few chapters. It was an interesting exercise, but I don’t think I’d want to be a programmer if writing sorting algorithms was what I did all day.

99% of the time you and I are working with a class of problems that are not in P. They are significantly more complex (and therefore a bit more interesting) and have a classification all their own.

Hard Problems (Exp)

Aaron’s democratic pub selection process will indeed help us find a pub in short order, but it’s a bit too simplistic. It would be nice if we had a chance to consider everyone else’s opinion before making our choice.

Olga speaks up with a suggestion:

How about each of us takes 30 seconds to explain where we want to go and why. Then we can decide after that.

This is a much nicer way to do things, and also a bit more complicated. Let's think about Olga's idea in terms of math.

Each of the 6 of us will be making a decision. With Aaron's democratic method, there were only 6 decisions + 1: each of us decided where we wanted to go – that's 6 decisions, plus one more in deciding the winner.

With Olga's way of doing things, each of us needs to make 6 decisions apiece:

- I need to decide what I want to do
- I need to listen to the other 5 people in the room and then decide if I'm going to change my mind

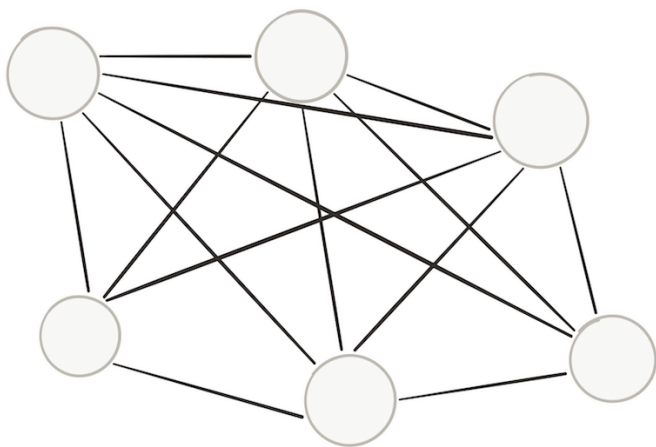
That's 6 people * 6 decisions, which is 36 total decisions. If this is all it took to make a group decision like ours, we would still be in P time because the equation for calculating the time required is still a polynomial equation:

$$T = n^2$$

This is not the extent of what we need to do, unfortunately. We're trying to *optimize* the selection of where we go – which means that I need to make a decision based on Aaron's, and Aaron's is based on Olga's, and Olga's is based on Todd's which is also based on Kim's...

If your brain is starting to hurt while thinking this through, I don't blame you. It's hard for any human mind to grasp the complexity of this type of process! In mathematical terms, this is a *factorial process*, or $n!$ where n is the number of people involved.

You can visualize this with a graph:



Each node on our graph represents one of my friends and each line represents a “weighting” in terms of a decision. If all we cared about was a measurement for each of the nodes, we would have a simple 36-part decision process. We want more than this, however. We want to optimize the combination of decisions, which is a factorial process.

This long-winded explanation is a way of saying that this problem is extremely complex. Rather than use the word “extremely”, however, a mathematician would say it’s “exponentially complex”, or that it is solvable in Exp time.

If we were to add a 7th person to our group – say my friend Kaveh finally shows up – the time to solve this problem would increase exponentially by an order of magnitude.

In pure mathematical terms, exponential time problems are solvable in:

$$T = 2^n$$

Where T is time, n are the inputs.

Exponential time is a very, very, very long time. It encompasses problems that could take millennia to complete... even millions, or billions of years. The sun can blow up 10 times over and our Exp problem could still be executing... chugging along... trying to figure out the optimal solution for us.

With small cases like ours, it would take us a long time to decide where to go, but it's likely that the sun would still be around when we're finished. Just because a problem is in Exp doesn't mean that it will always take a long time to solve. This is just a classification of complexity. Keep this in the back of your mind – I'll be coming back to it in a bit.

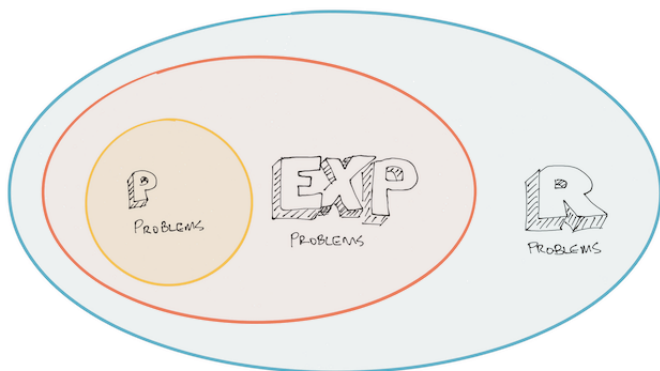
Almost every problem we can think of is in Exp. In fact, all the problems we care about solving are in Exp. I'll get to explain why that's true in just a bit – first, let's see what lies beyond exponential time.

All Solvable Problems (R)

Simply put, R represents all the problems that are solvable in finite time. In other words: *any problem that we can solve is classified in R.*

You might be confused by some of this as these definitions overlap. Problems in P are “solvable”, aren't they? As are problems in Exp? Doesn't this mean they're also in R?

Yes. These classifications are sets of problems within given timescales. A problem that takes 10 seconds to solve is still solvable within finite time. It's also solvable in exponential time. These timescales overlap! It might be easier to see this in an illustration:



Infinitely Complex Problems Beyond R

What problems could possibly take an infinite amount of time to solve? Believe it or not, some deceptively simple ones. There are people who make a living researching these things, if you can believe it.

There's a very famous problem that lies beyond R, and it's known as *The Halting Problem*. You might have heard of it – it was introduced by Alan Turing in the very same paper he used to introduce the Turing Machine. We'll get to Turing, his machine, and the Halting Problem in detail later on – but I'll summarize it very quickly here.

Imagine you've written a program and you want to know if it will end at some point (aka "halt"). Will it return an answer? Will it crash? As it turns out: *it's not possible to write a generalized program that can decide if any other program will halt*. This type of problem is called *undecidable*.

The proof of this is rather elegant and I'll get into it in a later chapter. It has to do with feeding our halt-checking program back to itself, tweaking the internals before we do, which will throw the machine into an infinite loop. The "infi-

nite loop” part is the thing that makes this problem undecidable.

There are quite a few other undecidable problems as well. In fact, if you dive into Complexity Theory a bit more you'll find that most of the problems that exist are undecidable, the proof of which is quite fascinating and a bit beyond my ability to explain. If you want to know more, have a look at MIT Open Courseware. There are hours of material up there.

Let's get back to the Halting Problem, however. I mention that it's solvable beyond R, but the strange thing is that the classification for this problem is something else entirely. It's classified as NP-Hard.

What does that mean? Let's dive into NP.

Determinism, Nondeterminism, and Magical Guesses

We have our three main complexity classes defined:

- P, which is the set of problems solvable in polynomial time. In other words: easy
- Exp, which is the set of problems solvable in exponential time. In other words: hard
- R, which is the set of all solvable problems

What do I mean, however, by the term solvable? As programmers we might think of writing a program to solve a given problem, or maybe constructing an algorithm of some kind. For the sake of moving things along, let's assume that these are the same things, and I'll talk about solutions in the context of a computer program.

When you sit down to solve a problem (let's say it's sorting an array), you lay out a set of instructions in code for the processor to execute. Conditional branches, loops, variables, etc. Hopefully you test your code because you need to

make sure that the computer can determine the correct answer.

This is how we write computer programs: deterministically. We execute a routine and get a result. Based on that result, we'll execute another one, and then another. Each result links together until we arrive at an answer.

There is another way to arrive at this result, which is nondeterministically. In a nondeterministic system we execute a routine and get a result, from that point we might execute one of a series of next steps. We don't know which next step from that series will be executed – it will be decided at that time.

Once that next step is executed (whatever it is), we might execute one of another series of steps – once again having no idea which step in that series it will be until it's chosen. We keep on doing this until we arrive at the answer. When we do have our answer, there's simply no way of determining how we got there, at least from our deterministic program's point of view.

You can think of nondeterminism as using a series of lucky guesses or chance (each of which is always correct) to figure out the answer to a given problem. If we were able to program a computer this way, things would get very interesting indeed.

Let's revisit the pub selection problem above, where my friends and I are trying to decide where to go have a pint and, possibly, a bit to eat. If I could use a nondeterministic algorithm to help us make this decision, we could leave in very short order! I could phrase the problem in a very particular way, and then off we would go:

- “If we go to the Leaking Moose, would everyone be optimally happy?” Answer: no

- “If we go to the Iron Tongue, would everyone be optimally happy?” Answer: no
- “If we go to the Cougar’s Kitchen, would everyone be optimally happy?” Answer: yes

There it is! To arrive at this answer all I needed to do was to iterate over the possible choices and ask a simple yes or no question. My nondeterministic algorithm was able to arrive at the answer directly, because it’s capable of correct lucky guesses every single time.

I’ll explain more in a second (there is an explanation to this, trust me) – but it’s important to understand what just happened. We took an exponentially complex problem and solved it in P time using a lucky nondeterministic process.

This might sound a bit magical, but there are quite a few people studying the notion of nondeterminism, trying to figure out if it’s possible to have a nondeterministic algorithm that can solve exponentially complex problems like our pub selection problem. They like this notion so much that they’ve identified a whole group of exponentially complex problems that can be solved in P time if only we had this amazing nondeterministic algorithm.

These problems are classified as NP: problems solvable in P time given a nondeterministic algorithm. NP can be thought of as a bit of a mythical time frame, because we simply don’t know if a nondeterministic algorithm is possible. If it is possible, then we can say that a given problem (like our pub selection problem) is solvable in nondeterministic polynomial time; which is what NP stands for.

More formally: a problem is classifiable in nondeterministic polynomial time (NP) if it is:

- Solvable in exponential time (Exp)
- Verifiable in polynomial time (P)

- Also solvable in polynomial time by nondeterministic methods

NP is where the action is. From the programs we write on a daily basis to the games we play at night with our family and friends: NP problems are what our brains enjoy the most.

Does $P=NP$?

We know, so far, that problems in P are pretty simple and can be solved in polynomial time using deterministic methods (aka “computers”). Problems in NP are quite a bit more complex, but become very simple if we have nondeterministic methods.

But we don’t have those just yet. The question is: **will we ever have the ability to solve problems nondeterministically?** Some people believe it’s just a matter of time before we have a computer chip or algorithm capable of nondeterministic processing. Others think it’s a bunch of silliness. If a nondeterministic algorithm is ever developed that can solve these problems, then all the problems in NP suddenly become solvable in P time. Put in mathematical terms: $P=NP$. If such an algorithm could never exist, then $P \neq NP$. We can’t make either claim immediately because we simply don’t know the answer.

I explained, above, that you can think of nondeterminism as a “lucky guess”, which is sort of true. You can also think of it as a computer’s ability to make the right choice given a set of possibilities. There are languages that are capable of carrying these ideas out. Prolog, for example, allows you to define two functions with the same definition. The programmer does not tell the runtime which to choose (which would be deterministic), the runtime decides. There

are various ways that this is carried out, including “back-tracking”, where one function is tried, and if it fails, the runtime will back up and try the second function with the same definition. This is brute force, and won’t solve our pub selection in P time... yet. Someday? Maybe?

Who knows...

Reductions and NP

I was able to use nondeterminism to solve the pub selection problem in P time by reducing the problem into a series of yes/no questions. The original problem was something a bit different:

What are the optimal pubs for optimal group happiness?

This type of problem is exceedingly difficult in that we’re trying to optimize a combination of things: people and pubs. For most people (myself included), this isn’t something your brain is equipped to handle – it’s just not possible to keep all the permutations together in your mind! These types of problems are called combinatorial optimizations and, as I mention, are very complex. But how complex are they?

Consider the definition of NP as it relates to this problem:

- Solvable in exponential time (check)
- Verifiable in polynomial time (no)
- Solvable in polynomial time by nondeterministic methods (no)

This means that this problem is not classifiable within

NP and is, instead, in Exp. Sort of what you might expect as this problem is exceptionally hard.

Our reduction to a decision problem, however, is a different matter:

If we go to the Leaking Moose, would everyone be optimally happy?

This problem is solvable in exponential time, so we know it's in Exp. It's verifiable in polynomial time – all I have to do is ask if people are happy once we're at the Leaking Moose. Finally, we already know that our nondeterministic algorithm can solve this problem in P time because we just did it.

That means our decision problem is classifiable in NP. This is where things get weird because, in essence, it's the same problem! I just reduced it from an optimization problem to a decision problem!

This type of thing is very common - where one version of a problem is classified differently than its decision problem reduction. The good news (I think) is that there are two classifications for just such an issue:

- NP-Hard: problems that can be reduced to other problems in NP, but are not within NP themselves
- NP-Complete: decision problems classifiable in NP

I don't blame you if your head is swimming. I know mine is... and I'm the one writing this book! If it helps: decision problems are almost always NP-Complete because of the ability to verify the answer to the problem (make sure it's yes). Combinatorial problems are NP-Hard.

Before we move on to some examples, I want to revisit a

statement I made above, about The Halting Problem being classified as NP-Hard. If you've stayed with me through this whole explanation you might be able to reason through the answer.

As we know: solving The Halting Problem is beyond R; solvable only with an infinite amount of time. It is, however, a simple decision problem: will this program halt? Because of that, other problems in NP can be reduced to it, and that's all you need for a classification of NP-Hard.

NP-Complete and Decisions

One of the main goals of turning a complex problem into a decision problem is the idea of verification. With our initial, combinatorial pub selection problem, we can only verify that we have indeed gone to the optimal pub for our group by going through every iteration of the decision process for each person and each pub. A pub crawl might not sound so bad, but visiting each friend (and each combination of friends) to ask if they like the place would get more than a little tedious.

The decision problem variation, however, is very easy to verify, as we did above.

These types of problems were formalized by Leonid Levin and Stephen Cook in the early 1970s, when they formulated the Cook-Levin Theorem. This theorem states that any problem in NP can be reduced to what's known as The Boolean Satisfiability Problem (or SAT):

In computer science, the Boolean Satisfiability Problem ... is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. In other words, it asks whether the variables of a given Boolean formula can be consistently

replaced by the values TRUE or FALSE in such a way that the formula evaluates to TRUE.

This problem sounds intimidating, but it's really not. The best way to think about it is to consider a very long **if** statement (pseudo code):

```
if ((x && y) && (x && !y)) || ((x || y) && (x || !y)){
    //...
}
```

Given this, SAT wants to know what values for **x** and **y** will return true?

Imagine a flow chart, with lots of branches, end points and ultimately a satisfiable terminus which returns true. This is a variation of SAT.

Now, imagine the last program you wrote (or the one you're writing now). There are business rules, database queries, and probably some type of UI that a user can use. At its core, this program is a bunch of decisions and can be described by a large decision tree. This is also a variation of SAT.

I suppose if you look hard enough, you could find a software project that exists to solve a problem in P. Most of them don't, however, if they expect to deliver any kind of value. Software exists to make complicated things easier. Given this, most of what we do on a daily basis is to deal with problems in NP.

This is an important thing to recognize, mostly because problems can usually be reduced from one to the other, as we've seen. Put another way: it's highly likely that a mathematician somewhere has tried to solve the very problem

that you're working on right now. To understand what I mean, we need to take a small trip back in time.

CLASSIC NP-COMPLETE PROBLEMS

In the early 1970s, mathematician Richard Karp expanded Cook and Levin's work with his paper *Reducibility Among Combinatorial Problems*. In this paper, Karp showed that you could reduce several NP problems to SAT in polynomial time, and he came up with a list called "Karp's 21 NP-Complete Problems":

The problems [\[edit \]](#)

Karp's 21 problems are shown below, many with their original names. The nesting indicates the direction of the reductions used. For example, **Knapsack** was shown to be NP-complete by reducing **Exact cover** to **Knapsack**.

- **Satisfiability**: the boolean satisfiability problem for formulas in **conjunctive normal form** (often referred to as SAT)
 - **0-1 integer programming** (A variation in which only the restrictions must be satisfied, with no optimization)
 - **Clique** (see also **independent set problem**)
 - **Set packing**
 - **Vertex cover**
 - **Set covering**
 - **Feedback node set**
 - **Feedback arc set**
 - **Directed Hamilton circuit** (Karp's name, now usually called **Directed Hamiltonian cycle**)
 - **Undirected Hamilton circuit** (Karp's name, now usually called **Undirected Hamiltonian cycle**)
 - **Satisfiability with at most 3 literals per clause** (equivalent to 3-SAT)
 - **Chromatic number** (also called the **Graph Coloring Problem**)
 - **Clique cover**
 - **Exact cover**
 - **Hitting set**
 - **Steiner tree**
 - **3-dimensional matching**
 - **Knapsack** (Karp's definition of Knapsack is closer to **Subset sum**)
 - **Job sequencing**
 - **Partition**
 - **Max cut**

Karp's 21 NP-Complete problems.

One of the best things you can do for your career is to get to know these and other NP-Complete problems, at least at a high level. They're fascinating to understand! Who knows? You might even save your job someday...

Let's take a look at a few.

Knapsack

This problem has been around for over 100 years, and is a combinatorial optimization problem that centers on packing a bag for the weekend:

Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

Once again, we're trying to optimize combinations of values: price and weight for a limited space.

Clique

This problem was first formulated in 1935, and has to do with graphs and graph theory:

Consider a social network, where the graph's vertices represent people, and the graph's edges represent mutual acquaintance. Then a clique represents a subset of people who all know each other, and algorithms for finding cliques can be used to discover these groups of mutual friends.

You can see how the solution to this problem could apply to any social aspect of an application, or grouping of like "things" that are self-assembling.

Bin Packing

This problem is a variation of Knapsack, and is once again a combinatorial optimization problem that you encounter quite often in the programming world:

... objects of different volumes must be packed into a finite number of bins or containers each of volume V in a way that minimizes the number of bins used.

If you've ever had to pack up your house or apartment and move, you've had to deal with the Bin Packing problem. This is a classic NP-Hard problem because of its combinatorial nature, and the fact that verifying that you've optimally packed things up means that you have to carry out every possible iteration to prove yours is the best.

We can reduce this to a decision problem, however, by iterating over bin configurations and asking if the current configuration is optimal. This reduction turns the combinatorial problem into a decision problem, which would classify it as NP-Complete.

Traveling Salesman

The classic NP-Hard problem of trying to figure out the cheapest way to send a salesman on a trip:

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

Once again, a combinatorial optimization problem, which we can recognize now as NP-Hard. You can reduce this in P time to an NP-Complete decision problem by simply enumerating through every valid path and asking "is there a path that's shorter"?

Approximations And Laziness

If you follow any of the links for these problems, you'll see that there are algorithms that exist which "solve" them in some cases in P time. These are called approximations and are very useful if you can tolerate their inexact nature.

For Traveling Salesman, you could start from Los Angeles and head to the next nearest city, which (for our example) might be San Francisco. When you get there, you see that of all the destinations you could get to, Reno is the next nearest, so you go there. This approach is called "nearest neighbor" and is classified as a "greedy algorithm", which means you do what suits your current position and value on the graph. Nearest neighbor usually returns a path within 25% of the shortest one, on average. Would that work for you? For many companies it just might.

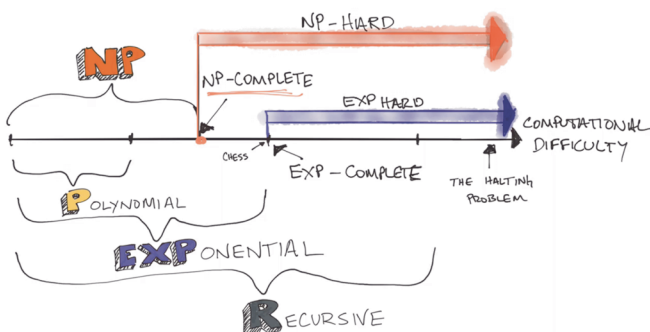
In The Real World

I'd like to share a story with you about getting fired, and how I could have avoided it. Before I do, let's go over what we know so far:

- Simple, rather boring problems can be solved in a short amount of time; what a mathematician would call "P" time.
- More difficult problems, such as a group of people trying to decide on the optimal location for a pint and some food, are more complex and solvable in exponential time, or "Exp".
- Some problems are so complex that they can't be solved in all the time we can possibly have (R, or "finite time"), and are simply undecidable. For instance: Turing's Halting Problem.

- There are special subclasses of problems that are of great interest to us, specifically: exponentially complex problems that we can solve in P time with a nondeterministic lucky guess. These problems are in a classification called “NP”. We can further divide this complexity class into decision problems that we can quickly verify (NP-Complete) and more complicated problems that are in Exp, but that other problems in NP can be reduced to in P time (NP-Hard).

We can visualize this on a timeline:



It might take a few reads to have this all sink in, but you might be wondering something: why in the world should I care. For that, I'd like to share a story.

Where's The Party?

In 2010 I was hired to build a Ruby on Rails site for a client that wanted to enrich the lives of college freshmen. The idea

was simple: let's help these new students make friends and find interesting things to go do.

Simple enough! I had been building web sites for the previous 12 years, this should be straightforward. Indeed it was! I built the core of the site, integrated the designs, and all was well until a status meeting we had one Tuesday afternoon:

So, Rob how's the progress on the algorithm?

Not knowing what algorithm they were talking about, I asked. They responded:

You know, the one that is going to match students with other students and places to go, etc. We were told you were good with this kind of thing...

Oh no. It's true: one of the reasons I got the job was because I had a background in analytics, but I didn't think I'd be doing any of it for this site! They hired me to build a Rails app!

This is where my ego took over. I hate saying "no" to clients, and somewhere in my mind was a cowboy looking for adventure... so I said what turned out to be some fatal words: tell me more...

The "algorithm" turned out to be this:

- Gather a basic interest profile from new signups. Things like favorite music, extrovert/introvert, favorite actor, etc.

- Create a tagging system for submitted events.
Anyone could submit an event to the system - but they had to tag it so we could match on it.
- Optimally match new students to other students and also to events that they would find interesting.

At this point in the chapter you should know what we're dealing with here. That's right: a combinatorial optimization. I could probably reduce this problem to any of Karp's NP-Complete problems, but (here, now, in the future) I don't need to do that to know that this problem is going to require some lengthy discussions.

Back then, I simply said "let me take a look".

You probably know where this is going to end up. **I got fired.** I looked into various ways to solve the problem but quickly realized that this isn't something that:

1. A Ruby on Rails site could handle
2. Could happen in real time
3. I would even know how to start implementing

I pushed back as best I could, the client insisted this is why I was hired, I asked them to show me where I agreed to this in the contract... but it didn't matter. I got fired and the project blew up.

If I knew then what I know now about exponential time algorithms, approximations, decision problems and all the other lovely things discussed in this chapter – I might have been able to walk the client through the problem and why, no matter how many developers they hired then fired, they would never get what they wanted.

Maybe.

Have you worked on a project like this? I'm sure you have. Or, if not, it's likely you will. People try to tackle NP-

Complete and NP-Hard problems all the time – after all we have machine learning and big data don't we?

The good news for you? Hopefully you will now be able to spot these efforts from the start and, hopefully, help your team approach the problem with care.

LAMBDA CALCULUS

ALONZO CHURCH CREATED A WAY TO COMPUTE ANYTHING USING ONLY MATHEMATICS, WHICH IS THE FOUNDATION OF PROGRAMMING.

Before there were computers or programming languages, Alonzo Church came up with a set of rules for working with functions, what he termed lambdas. These rules allow you to compute anything that can be computed. You use Lambda Calculus every day when you write code. Do you know how it works? As a programmer, understanding Lambda Calculus can enhance your skills.

The Code

You can find the code used in parts of this section up at my GitHub repo. In addition, you can buy over 10 hours of video walkthroughs of the algorithms you see in this chapter and others from here. I'll be using screenshots once again for the code samples for formatting reasons – if you want to play along please do... but you'll need the code from GitHub.

IN THE EARLY 20TH CENTURY, mathematicians began to ponder what it means to have a machine solve problems for

you. The question was simple: how do you compute something?

The steps to solving certain problems (aka: algorithms) had been known for millennia; the trick was to be able to give these algorithms to a machine. But how? More than that: is there a limit to what a machine can calculate? Are there solutions that machines simply cannot compute?

This led to some interesting discoveries in the early 20th century, most notably by two men: Alan Turing and Alonzo Church. We'll talk about Alan Turing in a later chapter.

This section is about Alonzo Church's contribution: the **Lambda Calculus**. I should note here, as I've done in so many chapters, that I could spend volumes diving into the details of Lambda Calculus. What you will read here is a simple summary of the very basics. I do, however, think the missing details are critical and if you care, I would urge you to have a look online – there are quite a few resources.

So, consider this a gentle introduction, the results of my recent headlong dive into the subject. Hopefully you will read enough to ignite your curiosity – which it should! What you're about to read is the foundation of computer programming.

Note: *the lambda function syntax you're about to read is included as actual text in the body of the chapter instead of being treated as code. In previous versions of the book I formatted the simple lambda equations as code, but many people found that confusing. The actual code samples that illustrate the lambda functions, however, are left as-is.*

Credit Where Due

I've read quite a few articles and textbooks on Lambda Calculus and I wanted to list them here, as I would never have been able to understand the basics otherwise.

At the top of the list is this detailed explanation of Y Combinator and Ω Combinator from Ayaka Nonaka. It is outstanding. I wanted to add some details about combinators and almost gave up, until I found this post.

Next is Jim Weirich's amazing keynote on the Y Combinator. I remember watching it years ago, having my mind blown. I watched it three times over when writing this chapter and most of it still goes over my head. I link to it again below.

Ben Hall has an outstanding GitHub repository that has all kinds of Church encoding magic. If you find yourself lost in any of this, go study the code in his repo. The Church encoding you see below is based directly on his work.

Finally, I'd like to thank James G (last name omitted... but you know who you are) for his patience helping me understand a small but crucial aspect of reduction and substitution. In the original version of this chapter I managed to get a few things wrong; I wouldn't consider them critical, but they were wrong nonetheless. After a number of lengthy, wonderful emails James convinced me of these errors, which I could cross-check and verify easily. Thanks James!

The Basics

Alonzo Church introduced Lambda Calculus in the 1930s as he was studying the foundations of mathematics. As a programmer, you might recognize the description:

The λ -calculus is, at heart, a simple notation for functions and application. The main ideas are applying a function to an argument and forming functions by abstraction. The syntax of basic λ -calculus is quite sparse, making it an elegant, focused notation for representing functions. Functions and arguments are on a par with one another. The result is an intensional theory of functions as rules of computation, contrasting with an extensional theory of functions as sets of ordered

pairs. Despite its sparse syntax, the expressiveness and flexibility of the λ -calculus make it a cornucopia of logic and mathematics.

A lambda is simply an anonymous function that can be thought of as a value. Most modern programming languages have some notion of an anonymous function, expression, or lambda. Lambda Calculus is where this idea arose. In fact Lambda Calculus is the foundation of what we consider programming today.

Lambda Calculus is an abstract notation that describes formal mathematical logic. There are no numbers or types; only functions. Like modern functional languages, a function in Lambda Calculus can be treated as a value. By arranging these functions carefully, you can build out some very interesting structures.

In this chapter we'll use Lambda Calculus directly, but we'll also jump into ES2016 (JavaScript) to see how some ideas might work with modern code.

Some Rules

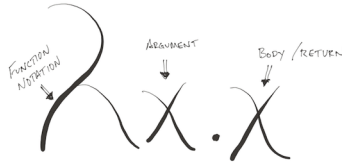
The rules of Lambda Calculus are rather simple:

1. There are only functions, nothing else. No data types (strings, numbers, etc.) of any kind
2. You can substitute functions using variables
3. You can reduce one function to another through application
4. You can combine multiple terms in Lambda Calculus to create a higher-order function called a combinator, which is where the fun begins

There are rules which you can apply to these functions and combinators, and we'll get into a bunch of them using Lambda Calculus and also a bit of JavaScript.

Anatomy

Let's see our first function:



This is a Lambda Calculus function, sometimes called a “term” or “expression”. The first thing to notice is the Greek lambda (λ) on the left, which denotes a lambda function. The next bit is the x , which is the argument to the function. The final part of the expression is the body, which is the second x and is segregated from the argument by a $.$

Using JavaScript, you could think of this function as **function thing(x) {return x}**. The **function** keyword is equivalent to the λ symbol. Our function takes an argument x and returns x . While this is, indeed, an anonymous function in JavaScript, there is a more applicable syntax that can be used with ES6: **(x => x)**

This is a pure lambda expression in the following ways:

- It takes in an argument, x , and since we're using a single line the value x is also returned
- It is a functional closure, which means we can set this expression to a variable and invoke it anywhere without worrying about scoping issues
- lambdas are the same as values

In this way, JavaScript follows Lambda Calculus conventions closely.

Function Application

Let's revisit our first function: $\lambda x.x$. This function takes an argument x and returns it. We can apply a value to this function like so: $\lambda x.x (y)$.

This notation means “substitute all occurrences of x with y ” and is called *function application*. It's just the same as if we did this in JavaScript:

```
(x => x)(y)
//or
function thing(x){return x}(y)
//returns y;
```

This function: $\lambda x.x$ in Lambda Calculus has a special name: the *identity function*. Whatever you pass to it is returned. It's also called the *I Combinator* – which we'll discuss in a bit.

Consider this function and application: $\lambda x.y (z)$. What do you think will happen here? It would be the same as doing this in JavaScript:

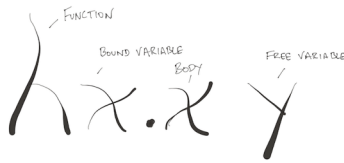
```
let z = 3;
function thing(x){return y}(z);
```

Here I'm using the value 3 just for clarity, though it's worth restating that there are no numbers in Lambda Calculus. There are *representations* of numbers, which we'll get to in a minute, but no strict numbers.

With this function: $\lambda x.y$, it doesn't matter what you pass into it, some value y will be returned. This is called the constant function as it returns a constant value of y no matter what you pass in for x .

Bound and Free Variables

When you apply a value to a function, there are some rules. The first thing you have to know is what *types of variables you're working with*. Note: I'm not referring to data types as there are none in Lambda Calculus; I'm referring to one of two different designations for variables:



In this expression we're dealing with two variables: x and y . The variable x is in the function head as the argument and thus is bound to this expression. The variable y on the other hand is not bound to any lambda function, so its known as "free".

The difference between the two is essential. I can substitute x in this function if (and only if) I make sure to substitute every occurrence of the bound variable x . So if I wanted to change x to z , for instance, I could: $\lambda z.z y$.

This wouldn't change the meaning of the function. If, however, I decided to change y we could run into trouble. To understand why, we need to dive into substitutions and reductions.

Substitution and Reduction

When you apply a value to a function, you substitute that value in the function itself. Substitution is *left-applicative*, which means you start on the left and move to the right as required.

For instance, let's substitute and then reduce this function: $\lambda x. x \ 3$

The first thing to do is to state that all bound occurrences (that's important, they have to be bound) of x will be replaced with 3 in the body, x :

$$\lambda x. x3$$

$$x[x := 3]$$

$$3$$

This notation: $x[x:=3]$ can be read as “replace all bound occurrences of x in body x with 3”. Now, I know I said that there are no numbers in Lambda Calculus - and this is true! I'm using 3 here simply to show how substitution and reduction work.

Let's apply this to JavaScript. Consider this expression:

```
let y = 3;
const fn = (x => y)(1);
```

What do you think the value of **fn** will be? The same principles apply here: we're substituting all instances of x with 1 in our equation, which doesn't matter as we're returning y , which is set to 3. This is, once again, the constant function so it's no surprise that whatever we set x to doesn't matter.

Let's try this again with the identity function:


```
let y = 3;
const fn = (x => x)(1);
```

In this case **fn** would “reduce” to 1.

Applying Multiple Values

You can create lambda functions that handle multiple values as well. It would be tempting to write them like this: $\lambda xy.t$

But this would be incorrect. Functions in Lambda Calculus only have an arity of 1 - which means they can only have one argument or one bound variable. To write this expression properly we would need to reorder things: $\lambda x.\lambda y.t$

This is called currying: breaking a function with multiple arguments into chained functions with an arity of 1. We’ll talk about currying more in a later chapter.

In this expression, the lambda function **x** has a body of $\lambda y.t$. This might look confusing, but if you remember that everything in Lambda Calculus is a function, it makes a bit more sense.

This corresponds to this lambda in JavaScript:

```
(x => y => x + y)
```

Reductions get interesting at this point. Consider this function: $\lambda x.\lambda y.y\ x$ We need to apply the substitution from left to right, so our first substitution will be “replace all bound values of **x** with **x**”:

$$\lambda x. \lambda y. yx$$

$$\lambda y. y[x := x]$$

$$\lambda y. y$$

There is no substitution in the body, $\lambda y.y$, which leaves us with $\lambda x.(\lambda y.y)$, which is the constant function, which means we can reduce further to $\lambda y.y$ and we're set.

We can see this reduction in action with JavaScript as well:

```
const first = (y => y);
const second = (x => first(x));
console.log(second(first));
//[Function: first]
```

In more concrete terms, let's revisit the JavaScript expression above $(x \Rightarrow y \Rightarrow x + y)$. This time we'll reduce it by running a substitution for x and y ... in other words “invoking” the lambda:

```
(x => y => x + y)(2)(3);
```

In this code, x is set to 2 and then passed to y . The resulting function would then be $y \Rightarrow 2 + y$, which we

apply 3 to, resulting in our answer, 5. This is straight up Lambda Calculus.

But what about this?

```
(x => x)(y => y)(2)(3);
```

This won't work and we'll get an error. Can you reason as to why? Let's make it work and then step through it:

```
(x => x)(y => y)(2); //2
```

This substitution and reduction happens in the same way, but the result of one lambda is passed to the next. So, $y \Rightarrow y$ is passed to $x \Rightarrow x$, which returns whatever was passed to it as it's the identity function.

This leaves the value 2, which is then passed to $y \Rightarrow y$, which is once again the identity function that returns 2.

To see this in more detail, set $y \Rightarrow y$ to $y \Rightarrow 5$. In this case $y \Rightarrow 5$ is passed to $x \Rightarrow x$ which, once again, just returns $y \Rightarrow 5$. If we pass 2 to that it's ignored completely, so we get an answer of 5.

Order Of Operations

Just like any mathematical operation, the use of parentheses can affect the order of operations in lambda expressions. Consider this function: $\lambda x.(\lambda y.y\ x)$ How do you think this would reduce? The parentheses dictate the order of operations so we would first reduce the body of the function like so:

$$\lambda x. (\lambda y. yx)$$

$$\lambda x. (y[y := x])$$

$$\lambda x. x$$

If we omit the parentheses, something different happens:

$$\lambda x. \lambda y. yx$$

$$(\lambda y. y)[x := x]$$

$$\lambda y. y$$

An entirely different result.

These are the basics of Lambda Calculus, now let's see what we can do with it!

Church Encoding

Lambda Calculus looks a lot like programming, doesn't it? Unfortunately, this can cause confusion for those trying to grasp it. Programming languages have numbers, strings, control structures and conditional branching built in; Lambda Calculus just has functions.

It's important to muse on this for a bit before we push forward. Put yourself in a classroom seat at Princeton, back in the early 1930s. Alonzo Church is trying to represent

constructs that we take for granted today: conditional branches, loops, and higher-order functions that can be used to compute things.

To do this, he set about creating *representations* for various values and operations. This is called *Church encoding*, and allows us to use booleans, numbers, conditional statements, and loops to construct things called *combinators* which are, unsurprisingly, combinations of functions that do a thing. We'll take a look at those later on; for now let's do the simplest of computations with Church encoding.

Booleans

How would you convey a simple conditional construct, like an if statement, using only symbolic functions? We need to have something like this (pseudocode): **if(true) x else y**.

The first step is to have a notion of what “true” means in Lambda Calculus, and we have that with the following representation: $\lambda x.\lambda y.x$. The first bound variable is returned for a **true** statement in Lambda Calculus. Conversely, a **false** statement returns the second bound variable: $\lambda x.\lambda y.y$.

Formalizing this to JavaScript we might have:

```
let True = (x => y => x);
let False = (x => y => y);
True(true)(false) //true
False(true)(false) //false
```

That's a great start! Now we need to stretch this further to have a conditional statement. We need to evaluate 3 things: returning the first value if **true**, second if **false**. Using Lambda Calculus and leveraging Church encoding, we can use this expression: $\lambda x.\lambda y.\lambda z.x\ y\ z$.

Translating this to JavaScript:

```
let If = (x => y => z => x(y)(z));
```

Our first argument, **x**, will be set to the function under evaluation. The arguments **y** and **z** will then be given to **x** to evaluate. The result of that will be the result of our **If** function.

Now, let's apply things. We can reuse our definitions of **True** and **False** to see if our **If** works:

```
let True = (x => y => x);
let False = (x => y => y);
let If = (x => y => z => x(y)(z));

If(True)("TRUE")("oops..");//TRUE
If(False)("oops")("FALSE");//FALSE
```

Yay! We have booleans and conditional statements, now we just need to work with some values.

Numbers

You can represent numbers in Lambda Calculus by arranging a function to, basically, encapsulate and call itself.

It might be easier to see this rather than to explain:

$$\lambda f. \lambda x. x = 0$$

$$\lambda f. \lambda x. f(x) = 1$$

$$\lambda f. \lambda x. f(f(x)) = 2$$

$$\lambda f. \lambda x. f(f(f(x))) = 3$$

This might look rather arbitrary at first, but there is some logic at work here. Consider this statement: $f(x) = x$. A function of x is equal to x . This means the function essentially has no value at all. This, therefore, represents 0: $\lambda f. \lambda x. x$. There is no reduction to be done here as nothing is applied. Thus f has a representative value of 0.

Now, consider this statement:

$$\lambda f. \lambda x. fx$$

$$(\lambda x. f)x[f := x]$$

$$\lambda x. x$$

The application of this function results in the identity function, which is a bit like multiplying by 1.

Let's try this with JavaScript. The first thing we need to do is to set up a **calculate** function that will figure out how many times a given function is called. This function will accept a Church number and invoke it with a return that is itself a function. Every time that result function is invoked, it will increment itself:

```

let calculate = f => f(x => x + 1)(0);

let zero = f => x => x;
let one = f => x => f(x);
let two = f => x => f(f(x));
let three = f => x => f(f(f(x)));

calculate(zero) // 0
calculate(one)  // 1
calculate(two)  // 2
calculate(three) // 3

```

Very nice! We now have a way of representing numerical values with nothing but a set of functions.

There is a lot more we can do here; things like addition, subtraction, multiplication, simple list operations and so on. In fact, we can encode our way to a Turing complete programming language. Which is really no surprise as the exercise we've just gone through is the foundation of what we think of as *programming itself*.

Combinators

We have booleans, numbers, and conditional branching; let's see what else we can do with Lambda Calculus. Before we get there, I should note that we've already been working with *combinators*, which are simply combinations of smaller functions that do a thing.

More precisely, a combinator is defined as:

... a *higher-order function that uses only function application and*

earlier defined combinators to define a result from its arguments.

In other words, we have lambda expressions and bound variables, nothing else. Let's see what we can do.

Loops and The Omega Combinator

Every good programming language needs to have the ability to loop, so let's see if we can implement that. We've seen that the identity function returns whatever is passed to it, so you might be tempted to do something like this:

$$\lambda x. x(\lambda x. x)$$

$$x[x := \lambda x. x]$$

$$\lambda x. x$$

This works once, but not multiple times. We can see this with JavaScript and Church numbers:

```
const zero = f => x => x;
const one = f => x => f(x);
const two = f => x => f(f(x));
const three = f => x => f(f(f(x)));
const four = f => x => f(f(f(f(x))));
```

A previously defined, a lambda can call itself, but there's no way to call a lambda from within its definition. We've been working with JavaScript a lot and we're able to work with variables and labels so it might seem like we can do this, but with pure Lambda Calculus, there are no such conventions as functions don't have a name.

But what if you could create recursion as a function itself?

We already have a bit of a start with the identity function. In our example above, however, it only returns what's on the right. How can we get it to repeat? What if we added one more application of x ?

$$\lambda x. xx(\lambda x. xx)$$

$$x[x := \lambda x. xx]$$

$$\lambda x. xx\lambda x. xx$$

The Omega Combinator

Well now, look at that. A function that not only returns what its given, but also its replica. This is called the Ω combinator, or “looping” combinator as it allows for recursively looping over a given function.

What does this look like in JavaScript? Let's see:

```
let Omega = x => x(x);
console.log(Omega(Omega));
//RangeError: Maximum call stack size exceeded
```

A recursive loop. This is interesting, but how can we apply some other function to this recursion? In other words, we have a basic looping structure, but it doesn't really do anything.

What would be more fun is to have a recursive function that would execute a given function once per iteration. In

other words, what you and I might consider a for/each construct.

The Y Combinator

The Y combinator is a recursive, fixed-point function. It's sometimes called the "fixed-point combinator" as well. If you're like me you might not know what "fixed-point" means.

It is, essentially, when the result of a given function is the same as the input:

$$yf = f(yf) \text{ for all } f$$

Consider this function:

$$f(x) = x^2 - 3x + 4$$

If I set $x=2$, then the input will equal the output. This is the fixed point of this function. We want to expand on this idea, but instead of passing in a value to the Y combinator, we want to pass in a function, invoke it, and then get that function back so we can invoke it again later.

This is what the Y combinator does: you give it a function as an argument which it invokes. It then returns that function back out to you.

More formally, the Y combinator is defined as:

$$Y = \lambda f. (\lambda x. f(xx))(\lambda x. f(xx))$$

The Y Combinator

It looks just like the Ω combinator doesn't it? The main

difference, however, is that instead of just replicating **xx**, we're replicating the application **f(xx)**, where **f** is the initial function itself. In addition, the Y combinator allows you to pass in a second expression which represents the number of times you want the loop to execute. We'll see this below.

If you're stuck on this (which I was, for a very, very long time), go through the building up of the Ω combinator to see how the recursion/replication works. Ideally you should see the same pattern at work here; all we're doing is adding a function definition (**f**) and then invoking it.

In JavaScript it looks a bit different. You can't reference *f* in the second lambda definition as you'll get an error, so you need to wrap it with an additional lambda:

```
const Y = f => (x => x(x))(x => f(y => x(x)(y)));
```

The Y-Combinator in JavaScript

This is interesting, but what can you do with it? Let's see.

We need a function to iterate over, which you can think of as the code block you might pass to a **foreach** statement. As with all computer science books, Fibonacci must appear at some point, so let's use that:

```
const fib = f => n => n <= 1 ? n : f(n-1) + f(n-2);
```

If you don't recall, Fibonacci defines a set where a given number is the sum of the two previous numbers in the set. The first few numbers are: **1, 1, 2, 3, 5, 8, 13...**

The expression above simply takes in a number, **n**, and checks to see if it's less than or equal to 1. If it is, then it's

returned. If it's not, then the function calls itself to figure out previous values.

There is a better, faster way to write this which I'll go into in the chapter on algorithms. It's important to note, however, that it wouldn't be a pure lambda expression if I did that. We want a self-contained function expression. No variables, no conditionals, etc. Just functions - Lambda Calculus style.

Let's use Y Combinator to implement our function:

```
let Y = f => (x => x(x))(x => f(y => x(x)(y)));
let fib = f => n => n <= 1 ? n : f(n-1) + f(n-2);
let yFib = Y(fib);
yFib(10);
//55
```

Solving Fibonacci using the Y combinator

Yes! It works! We've just used Y combinator to execute our `fib` function 10 times, which yields the result **55**.

Want to play around with some other combinators and see what you can create? Head over to Ben's GitHub repo and have a play!

Summary

My head is starting to hurt a bit; it might be time to move on. Hopefully you have a grasp on the basics of what Lambda Calculus is and why it's important. Simple functions that do a simple thing, which you can arrange carefully to do more complicated things in a symbolic way.

A compelling way to compute things, which allowed Alonzo Church to make this claim:

All total functions are computable.

Put another way: *if it can be computed, Lambda Calculus can compute it*. Turing agreed with this, and in 1937 he proved that his Turing machine provided the same computational abilities as Lambda Calculus, which led to the Church-Turing conjecture. We'll discuss that in a later chapter.

Today, this type of statement is kind shrug-worthy. We have powerful computers that can compute almost anything – so what? Back then, however, people with pencils and papers were the ones doing the calculations and they were limited in their ability.

This led mathematicians to ponder *just what could be computed* in terms of complexity – the human brain is only capable of so much. What was missing was a method of computation that could guarantee a result if the thing to do the computing had unlimited resources. Like a mechanical computer.

Those came along in just a few short years after Church and Turing's work and guess what we use today for programming these things?

Lambda Calculus.

COMPUTING MACHINES

USING ABSTRACT MACHINES TO DERIVE A SOLUTION

Babbage, Lovelace, Church and Alan Turing ¹ laid the foundation for how a machine could be used to compute things, but how did this come about? The abstract notions of machinery and computation led engineers like Jon von Neumann, JP Eckert and John Mauchly to design and build the very first electronic computer. Every computer that exists today is based on these abstractions and designs.

Over the years, the notion of an abstract ability to compute things has taken on many forms. Let's visit that now, starting off with a little history, once again. We'll visit Plato and ponder the true nature of things, drop in on Bernoulli in the 1500s and wind our way to Russia in the early 1900s. We'll visit Bletchley Park and Alan Turing in the early 20th century, eventually ending up back in the United States with John von Neumann, the creator of the modern computer.

Probability and The Theory of Forms

At some time around 400 BC, Plato mused that the world we see and experience is only a partial representation of its true form. In other words: an abstraction. The real world is either hidden from us, or we're unable to perceive it.

He based this notion on his observations of nature: that there is a symmetry to the world, to our very existence, that lies just beyond our ability to fully perceive it. Phi, the Golden Ratio, pi, and e are examples of some cosmic machinery that is just outside our grasp.

To many, the natural world appears to be a collection of random events, colliding and separating with no guiding purpose. To a mathematician, however, these random events will converge on an apparent truth if we simply study them for a long enough time.

The Italian mathematician Gerolamo Cardano suggested that statistical calculations become more accurate the longer you run them. Jacob Bernoulli proved this in 1713 when he announced The Law of Large Numbers in his publication *Ars Conjectandi* which was published after he died.

This law has two variations (weak and strong) – but you can think of it this way: if you flip a coin long enough, the statistical average will come closer and closer to 50% heads, 50% tails. This might seem obvious – after all there are only two sides to a coin and why wouldn't it be a fifty-fifty distribution?

The simple answer is that reality tends to do its thing most of the time, with apparent disregard for our mathematical postulations. The fact that we can completely rely on statistical models over a long enough period is astounding.

This is what keeps casinos in business. All they must do is to make sure the mathematical odds of each of their games is in their favor, and over time the money they make on those

games will reflect those odds increasingly closely. It doesn't matter if you walk in tomorrow and win all their money – statistics says they will get it back if they wait long enough.

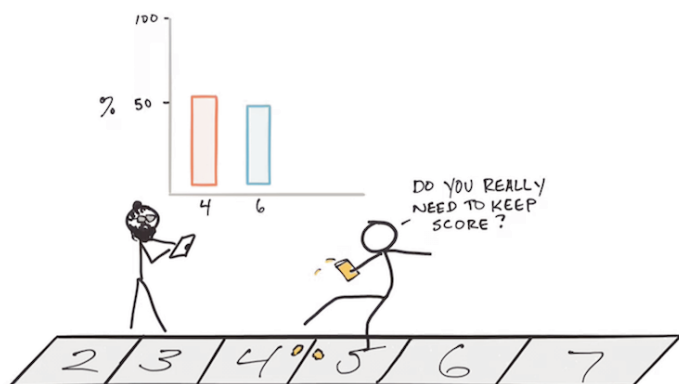
Flipping a coin, however, is just a single event. Playing a game of craps or a hand of blackjack consists of a series of events, one dependent on the next. Does the law of large numbers still hold?

Markov Chains

The Russian mathematician Andrey Markov said yes and set about to prove it in the early 1900s with what has become known as the Markov chain.

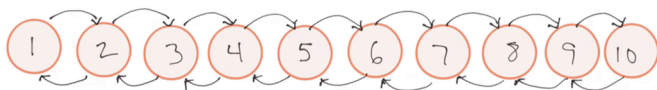
If you've ever used a flow chart, then you'll recognize a Markov chain. It is usually described graphically as a set of states with rules that allow transition between those states:

A famous Markov chain is the so-called “drunkard's walk”, a random walk on the number line where, at each step, the position may change by $+1$ or -1 with equal probability. From any position there are two possible transitions, to the next or previous integer. The transition probabilities depend only on the current position, not on the manner in which the position was reached. For example, the transition probabilities from 5 to 4 and 5 to 6 are both 0.5, and all other transition probabilities from 5 are 0. These probabilities are independent of whether the system was previously in 4 or 6.



A Markov chain looks a bit like a flow chart, and in many ways that's exactly what it is: a probability graph of related events. This fits nicely with the notion of an abstract computing machine.

DRUNK MARKOV CHAIN



In the above diagram, each orange circle is a “state” and the arrows dictate transitions possible between each state. Except for 1 and 10, the only transitions possible at any state along the chain is the next direct state or the one prior. Notice also that none of the states allows for a “loop back”, or a transition back to itself.

This diagram represents a very simple abstract computation, or a “machine”.

Finite State Machine

If we focus only on the notion of state and transitions, we can turn a Markov chain into a computational device called a Finite State Machine (or Finite Automata).

A simple machine does simple work for us. A hammer swung in the hand and striking a nail translates various forces into striking power, which drives a nail into wood. It doesn't get much simpler than that.

The nail, as it is hit, moves through various states, from outside the wood to partially inside the wood to fully inside the wood. The nail transitions from state to state based on actions imparted to it by the hammer.

The action of the hammer transitioning the nail through various states constitute a state machine in the simplest sense.

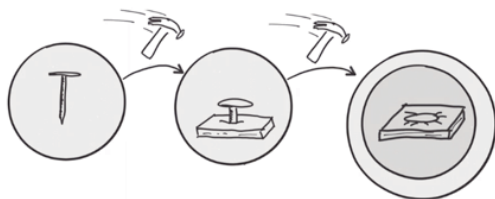
We discussed determinism a few chapters back, and we also discussed Finite State Machines a little. Let's go a bit deeper.

A Finite State Machine (FSM) is an abstract, mathematical model of computation that either accepts or rejects something. We can apply this abstraction to our hammer and nail process above, so we can describe things mathematically. We start by diagramming the state of the nail:

- Outside the wood is the starting state
- Partially driven into the wood is another state
- Fully driven into the wood is a final state, also called acceptance
- A bent nail is also a final state, but not the state we want so it is a rejection state



We can relate these states together through various transitions or actions, which is the striking of the hammer:



This is a deterministic finite state machine, which means that every state has a single transition for every action until we finally reach acceptance or rejection. In other words: when we hit the nail, it will go into the wood until its all the way in, which is our accepted state (denoted by a double circle) – there is no other course of action.

Being a good programmer (which I'm sure you are), you're probably starting to poke holes in this diagram. That's exactly what you should be doing! It's why these things exist!

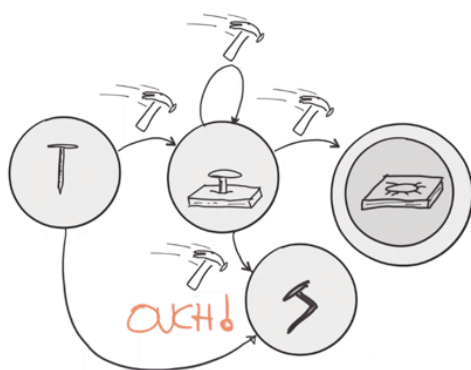
Our FSM only describes moving the nail through three states with only three actions. Furthermore, there is no rejection state for when I partially hit the nail and bend it. How do we account for that?

A non-deterministic finite state machine has one or more transitions for a set of actions, and these transitions are

essentially “random”. A nondeterministic machine is capable of transitioning from one state to the next as a result of some random choice, completely independent of a prior state.

As a programmer, you can think of a nondeterministic machine as a program that will produce different results when run multiple times – even with the same input given.

Let’s update our FSM to be nondeterministic by introducing partial hits which can sometimes bend the nail:



Here, I have two possible transitions for the initial state: I hit the nail, or I miss and bend the nail (yelling loudly when I do). I’ve added an action on the second state as well – the one that loops back to itself. This is the action taken when the nail has not been fully driven into the wood.

The conditional step of hitting the nail multiple times is deterministic, however the random step of bending the nail is not as we don’t know if it will happen before we start hammering. Sort of. There are all kinds of variables and probabilities at play here from muscle twitches to fatigue, hammer integrity and wind direction. It will all come together at some point and the probabilities are so out there that we might as well consider this to be a random event.

Let's move away from hammers and into the world of code and machine processes. You'll often hear people talking about FSMs working over a particular alphabet, string, or language. This is particularly relevant in computer science.

At the lowest level, our alphabet is a bunch of 1s and 0s – bits and bytes on a disk or in memory that we need to understand. This is how computer science people thought about the world decades ago: as holes on a piece of paper fed into a machine.

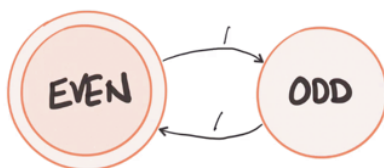
Let's do another FSM, but this time we'll do some real computing. Let's take a string of bits and write a routine that will tell us if there is an even number of 1s in the supplied string.

We'll start out with the two possible states:

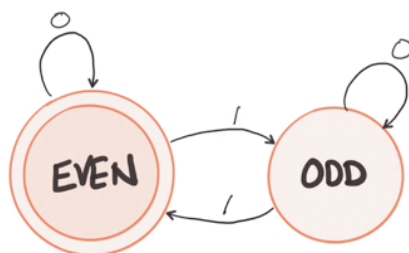


The double-circle here is our accept state, and if that's also our final state we have acceptance. Now we need to account for the action of reading each character in our string, one at a time.

If we read a 1, we'll have an odd number of ones. If we read a 1 while in the odd state, we'll transition to the even state. Another 1 and we're back to the odd state:



What happens if we read in a 0? We do nothing and just keep on reading:



Here is the part where we come to the naming of things. An FSM relies on a simple, known or finite set of conditions. The inputs are 1s or 0s, we have 1Mb of RAM to work with, or maybe 10Mb of hard drive space – these are finite conditions and tend to describe simpler constructs.

Streetlights, alarm clocks, vending machines – these are perfectly good Finite State Machines. An iPhone is not.

To understand this, think about our 1s and 0s example above. What if we wanted to sum all the 1s and then divide that by the sum of all the 0s to get a randomization factor? In short: we can't describe this with an FSM. There is no notion of a summing operation.

The basic problem is this: *you can't store state outside the current state*. In other words: there is no persistence, or RAM.

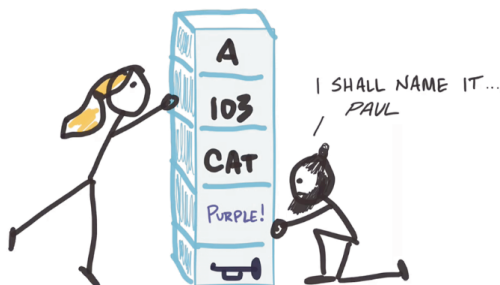
We can't calculate the digits of pi, or perform map/reduce

operations. We can, however, parse Regular Expressions (ugh).

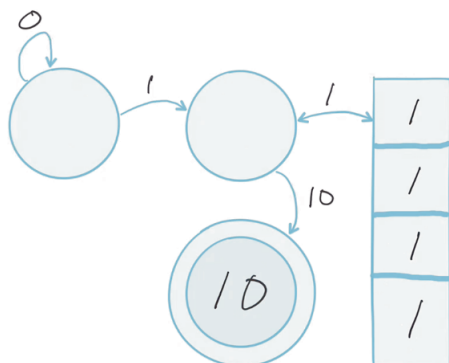
Much of the world runs on FSMs, but for what you and I do to make a living – we need a computational model that allows for a little more complexity.

Pushdown Machine

If we want to sum things with our computation, we need to move away from the concept of an FSM and add computational power to our machine. To sum things, our machine will at the very least need to remember the last state. We can add a facility for this called a stack.



A stack is quite literally that: a stack of data that you can add to (push) or remove from (pop). If we alter our FSM to have a stack, we can now compute a whole new set of problems:



This, however, is no longer a Finite State Machine – it’s called a Pushdown Machine (also called Pushdown Automata or PDA). This machine has a lot more computational power because the state transitions are decided by three things working together:

- The input symbol (a 1 or a 0)
- The current state
- The stack symbol

With this new power we can compute running totals of 1s, creating a summing operation, and a lot more.

The notion of a stack is very powerful, but it’s also a bit limited. You can only compute something that the space on the stack can handle. In other words, if we had 10 possible states and only 5 slots in our stack and we wanted to run a summing operation as before, we could easily run out of memory and our application would crash.

The Pushdown Machine is bounded by its stack and is, therefore, limited in terms of what it can compute. But what if the stack didn’t have a limit?

This problem was solved by Alan Turing.

Turing's Machine

In the mid-1930s, Alan Turing published what has become one of the cornerstones of computational theory as well as computer science in general: his paper, entitled *On Computable Numbers*.

Written when he was just 24 years old, *On Computable Numbers* described a computational process in which you stripped away every “convenience” and introduced a machine with a read/write head and some tape. The tape has a set of cells, each of which holds a simple symbol of some kind, and you can move that tape under the head so the machine could read from it, or write to it:

We may compare a man in the process of computing a real number to a machine which is only capable of a finite number of conditions $q_1: q_2. \dots q_I$; which will be called “ m-configurations “. The machine is supplied with a “tape “ (the analogue of paper) running through it, and divided into sections (called “squares”) each capable of bearing a “symbol”. At any moment there is just one square, say the r -th, bearing the symbol $\epsilon(r)$ which is “in the machine”. We may call this square the “scanned square “. The symbol on the scanned square may be called the “scanned symbol”. The “scanned symbol” is the only one of which the machine is, so to speak, “directly aware”. However, by altering its m-configuration the machine can effectively remember some of the symbols which it has “seen” (scanned) previously. The possible behaviour of the machine at any moment is determined by the m-configuration q_n and the scanned symbol $\epsilon(r)$. This pair $q_n, \epsilon(r)$ will be called the “configuration”: thus the configuration determines

the possible behaviour of the machine. In some of the configurations in which the scanned square is blank (i.e. bears no symbol) the machine writes down a new symbol on the scanned square: in other configurations it erases the scanned symbol. The machine may also change the square which is being scanned, but only by shifting it one place to right or left. In addition to any of these operations the m-configuration may be changed... It is my contention that these operations include all those which are used in the computation of a number.

Each “machine” that Turing describes is designed to take another machine as its input, which allows for massive computational power. Alonzo Church had the same notion with Lambda Calculus: functions that take other functions as arguments. This flexible structure is the foundation of modern computer science.

We can write a Turing Machine (using mathematical notation) to describe the number 4 (M4). We can write another machine to describe the number 6 (M6) – and yet another to perform multiplication on a set of numbers (MX).

We can then write a final Turing Machine that accepts M4 and M6, and uses MX to run the multiplication. This is central to Turing’s idea: small, concise machines orchestrated to derive a final result. 1s and 0s are all we need to describe and compute any problem we can describe to the machine.

This led Turing to claim something rather extraordinary:

If an algorithm is computable, a Turing Machine can compute it

This is another way of stating what Alonzo Church asserted two years prior:

All total functions are computable

This became known as the **Church-Turing Conjecture**, and was an extraordinary claim for the time. Prior to the idea that machines could calculate something for us, mathematicians would declare that a given problem was effectively calculable if someone could sit down and figure it out with a pencil and some paper.

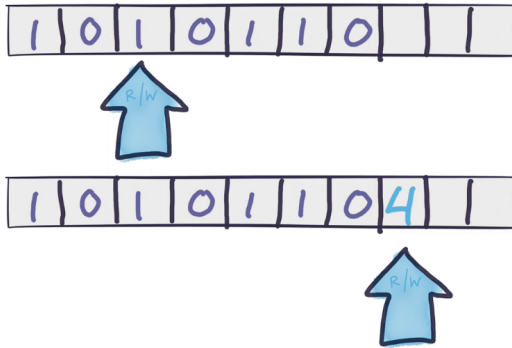
Church and Turing, however, gave us a different way to run calculations and, therefore, a different way to think about what we can compute, in general. Put in a simpler way: **if you can describe it with a Turing machine or with a set of functions, it can be computed.**

A Turing machine is an abstract computational device that can compute anything that you can program. It's important to note that this is not supposed to be a real machine although some have made one.

It's an incredibly simple construct that can simulate any computational algorithm, no matter how complex.

A Turing Machine has four main parts:

- A set of symbols, defined in an alphabet or language the machine can understand (usually just 0s and 1s)
- An infinitely long “tape” which contains a single row of symbols that can be anything
- A read/write head that reads from the tape and writes back to it
- Rules for reading and writing from the tape



If you create an instruction set that is capable of running on a Turing machine, it is said to be Turing Complete. All you need for Turing-completeness is:

- Conditional branching
- Loops
- Variables and memory

Turing was trying to create a model of computation that could, essentially, scale to any problem thrown at it. A Turing machine has more computational power than the machines previously discussed because a Turing machine has an infinite amount of storage (in the form of a “tape”) to read from and to store state to. Now, being good computer scientists, we might get stuck on the notion of an “infinite” bit of tape. Such a thing might be a bit hard to come by in the real world.

Believe it or not, it doesn’t matter; this is a conceptual machine after all. The amount of tape use, the speed of the tape through the machine and/or the symbolic alphabet that you choose to use with it – none of these variables effect the overall computational power of a Turing machine. As we learned last chapter: *if you can describe it with a Turing machine, it’s computable.*

It's a wonderful computational model, and it's derived from the idea of stripping the notion of computation itself to the barest minimum, as Turing further described in his paper:

Computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child's arithmetic book. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper, i.e. on a tape divided into squares.



Given that a Turing Machine can write to, as well as read from, a tape full of symbols, we now can store an “infinite”² amount of information (assuming we have an infinite tape).

State, in a Turing Machine, is stored on the tape that it is given to read from:

The behaviour of the computer at any moment is determined by the symbols which he is observing, and his “state of mind” at that moment. We may suppose that there is a bound B to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations.

The “tape” is being observed by moving under what Turing called a “head”, something that can both read and write. As the tape moves, the machine observes the symbols and the “state of mind” of the machine changes because the information within changes.

By the way: it’s quite fun to see Turing’s treatment of the machine in human terms.

We may now construct a machine to do the work of this computer. To each state of mind of the computer corresponds an “m-configuration” of the machine.

Multiple machines, multiple configurations – all usable by other machines with different configurations: this is the essence of the Turing machine.

One very interesting feature of a Turing Machine is that it can read in and simulate another Turing machine, even a copy of itself. In the 1930s, when *On Computable Numbers* was written, machines were built for a specific purpose. A drill made holes, an elevator lifted things, etc. so it made sense that computing machines would be purpose-built in the same way.

Turing proposed something altogether different. His abstract machine could simulate any other abstract machine – making it a universal computational device.

This idea had quite a profound effect, as you can imagine. The question of what *can we solve* was no longer interesting – with the Church-Turing Conjecture the question evolved to something a bit more profound: what can't we solve?

In a wonderful bit of cheek, Turing provided the answer in the very same paper he used to introduce his machine.

The Von Neumann Machine

We started this book by thinking about complexity and computation in general. We then got a little steampunk and explored the early 1800s and Babbage's Analytical Engine – only to find out that he and Ada Lovelace had designed the first Turing-complete language out of punch cards.

We explored Turing machines in this chapter and got to know both the Church-Turing Conjecture and its counterpart: *The Halting Problem*.

We have a couple of ways of computing anything that is computable – but we're still stuck in the land of theory. This changed in the early 1940s with John von Neumann, JP Eckert and John Mauchly.

Oh, and Turing too, as it turns out.

We know that a Turing machine works on a very simple idea: a read/write head and an infinite supply of tape. The machine has some type of instruction set that influences the current state of the machine. It's simple enough to conceive of a tape and a read/write head – but what about that instruction set? Where does it live?

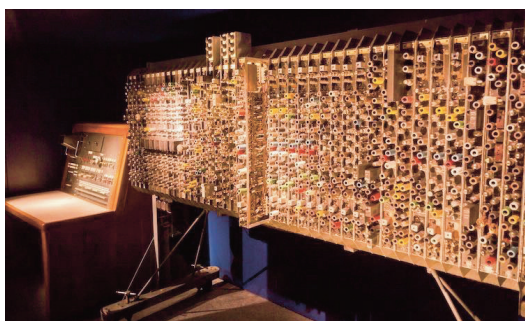
Initially, the idea of producing a working computer was that you would produce it for a specific purpose – maybe a calculator or prime number generator – a dedicated machine

hard-wired to do a dedicated process. This seemed to defeat the notion of a “universal computing device”, but the hardware simply wasn’t there yet.

At almost the same time, Turing and von Neumann were plotting to fix this. They both came up with the same answer: make the instructions themselves part of the tape. This is what we call a program today.

The Automatic Computing Engine

In 1946, just after the war, Turing designed and submitted plans to the National Physics Laboratory (NPL) for an electronic “general purpose calculator” called the Automatic Computing Engine – using the term Engine in homage to Charles Babbage. A number of these machines were created, and they stored their processing instructions as part of the machine’s data.



Turing’s ACE Machine. Image credit: Antoine Tavenaux. You can see this on display at the London Science Museum.

In an interesting historical side note: the NPL wasn’t sure that creating such a machine would work. They were unaware of Turing’s work during the war, where he worked with an electronic computer every day (Colossus) to help

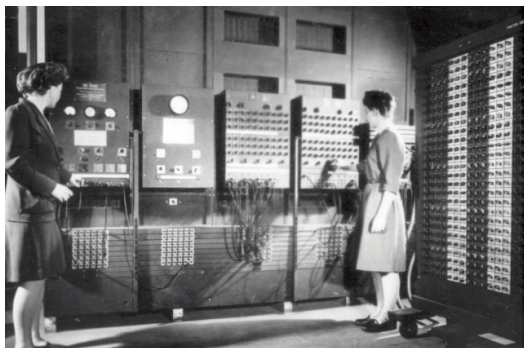
break Nazi ciphers. He couldn't tell them this, however, as he was sworn to secrecy.

ENIAC and EDVAC

In the early 1940s JP Eckert and John Mauchly were designing ENIAC, the world's first electronic computer. Governments were realizing quickly that computers would give them the edge during wartime, and the race was on.

The existence of Colossus, the machine that helped crack German codes during World War II – is known today. Back then, however, this was a big secret that no one outside of Blechley Park knew about. I bring this up because the initial designs for computers, underway with various science teams around the globe, were done in secret and without much sharing.

ENIAC, for instance, was developed to calculate ballistic trajectories and other top-secret calculations. It was Turing-complete and programmable, but to program it you had to physically move wires around.



ENIAC Operators

In 1943 Eckert and Mauchly decided to improve this

design by storing the instruction set as a program that was stored next to the data itself. This design was called the EDVAC.

By this time the Manhattan Project (the US Atomic Bomb project) was rolling, and von Neumann, being a member of the project, needed computing power.

He took interest in ENIAC and later EDVAC which led him to write a report about the project, detailing the idea of a stored-program machine. This is where things get controversial. For one reason or another – maybe because the paper was an initial draft – von Neumann's name was the only name on the report, even though it was mainly Eckert and Mauchly's idea.

As well as Turing's – but he couldn't tell them that because it was still a secret.

This gets further complicated by a colleague of von Neumann's, who decided to circulate the first draft to other scientists. Soft publishing it, if you will. These scientists got very interested in the paper and ... well ... we now have the generally incorrect attribution of stored programs to von Neumann.

One of von Neumann's colleagues, Stan Frankel is quoted as saying:

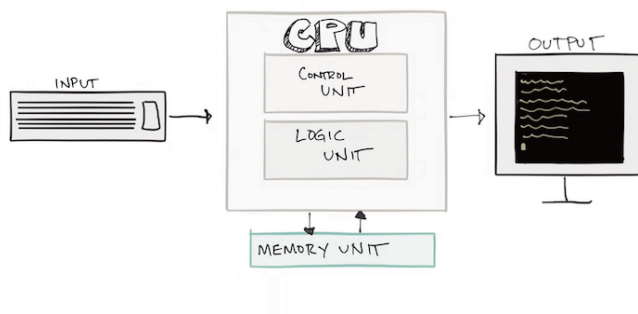
Many people have acclaimed von Neumann as the “father of the computer” (in a modern sense of the term) but I am sure that he would never have made that mistake himself. He might well be called the midwife, perhaps, but he firmly emphasized to me, and to others I am sure, that the fundamental conception is owing to Turing— in so far as not anticipated by Babbage... Both Turing and von Neumann, of course, also made substantial contributions to the “reduction

to practice” of these concepts but I would not regard these as comparable in importance with the introduction and explication of the concept of a computer able to store in its memory its program of activities and of modifying that program in the course of these activities.

As you might imagine, Eckert and Mauchly were not happy about this. Either way, this new idea now has a name.

Von Neumann Architecture

Historical silliness aside, we have now arrived at a modern-day computer and the birth of modern computer science. von Neumann’s machine should look reasonably familiar:



von Neumann Architecture

This machine architecture allows us to create, store, and deliver functions as a set of symbols. Thinking about this in depth can twist your brain a little.

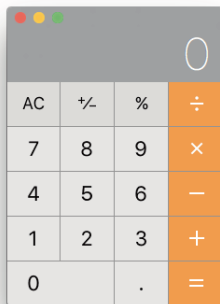
A Turing machine is an abstract construct and the initial attempts to create one ended up with rather large bits of

machinery that needed to be fiddled with by hand to run a given calculation. von Neumann could abstract that fiddling, which meant that Turing Machines could be built with nothing more than code and Turing's original vision of a "Universal Turing Machine" was complete.

To grasp this in a more real sense – think of a pocket calculator.



It does only one thing: math. Now open the calculator on your computer.



My Mac's Calculator.

It does exactly the same thing as the calculator above, but there's no hardware involved. I can also change it to do scientific and programming calculations on the fly, which is kind of magical. **The Mac calculator is as much of a machine as the one you hold in your hand**, but it's made of nothing more than pixels.

You can thank Mauchly, Eckert and von Neumann for this. Machines running within machines. When you consider various code execution environments (like the JVM, the CLR, or Google's V8 Engine) that are running on virtualized machines in the cloud this whole idea tends to start spiraling.

How many abstract machines are involved to execute the code you're writing today? When you run a VM or Docker (or some container functionality of your choice), these are machines within machines executing machines made up of smaller machines within other machines...

It's machines all the way down.

BIG O NOTATION

DESCRIBING THE COMPLEXITY OF AN ALGORITHM
USING MATH

Have you ever written some code that you were rather proud of? It's a pretty good feeling to see a test pass so you can move on to solving another problem or, better yet, writing more tests.

Any coder can solve a problem given enough time, *solving it well*, however – that's what we want to do! But what does that even mean?

Simply put it means that your code does what the spec requires, it can scale, and it's written in a way that other developers will understand *your intentions* in the future. This chapter focuses on the second part of that sentence: the “it can scale” part.

How can you demonstrate that your code can scale using something more than just waving your arms? **You can use Big-O.**

The Code

You can find the code used in parts of this section up at my GitHub repo. In addition, you can buy over 10 hours of video

walkthroughs of the algorithms you see in this chapter and others from here.

BIG-O NOTATION *mathematically describes the complexity of an algorithm in terms of time and space.* It is intimidating and quite a few developers I've encountered (coworkers, at conferences and so on) will instantly withdraw from a conversation at the first mention of "Order N" or the like.

It can come off as elitist if you examine someone else's code and casually drop Big-O into a conversation. It can also mean that you'll pass your next interview when your interviewer asks you about the complexity of some code you've written and whether you can improve it to be **log n** vs. **n²**...

A Word About Data Structures

Before we get going, I think it's worth addressing the examples you're about to see. I'll be using a basic array for each one of them, and if you're not a JavaScript developer you might think to yourself *that's ridiculous! I would never write code like that!* And I would understand.

.NET, for example, has a remarkable amount of list types, both generic and non, that allow you to write rather powerful, exact code for the task at hand. Elixir and other functional libraries allow you to choose freely between an enumerable, a list, a dictionary, or a map.

Your choice of data structure depends squarely on the type of data you're working with and then what you're trying to do with that data. *But do you know why you're making these choices?* Do you know why these structures exist in the first place?

Each data structure in .NET (or Java or Elixir) was created for use in a particular type of algorithm. If you weren't formally trained in data structures (as I wasn't), you would just pattern your choice from what you've read in blogs or been told by a more senior developer. This works fine for many, but you're reading this book because you want to go a bit deeper, to learn the concepts that underly so many decisions you've made in the past.

To ask "why this data structure?" is to also ask "how are we using this data", which then naturally leads to "which algorithms are we going to implement?". The answers to these questions are somewhat interdependent, which presents a problem for me in terms of writing this book in that *I have to start somewhere*.

So I'm going to level the playing field. All we're going to use in this chapter is an array and some integers - much like any coding interview. I'm doing this because I want to be able to focus on *complexity* as a means for making educated choices about algorithms and, correspondingly, the data structures you choose to work with.

We'll get into data structures and algorithms later in the book, for now I ask you to suspend what you know about various list types and to just go with the flow. Yes, the examples are contrived, but only because I didn't have a choice!

Onward...

A Super Simple First Step: $O(1)$

Let's just jump right in and see what kind of mess we can make here, shall we? Let's say I have an array of 5 numbers: **[1,2,3,4,5]**.

Now, let's say I ask you to get the first number from this array. Obviously if you look at it can you say "oh - sure that's

a 1". A program doesn't have eyes, however, so it needs a way to pull that number out.

Being the smart person you are, you decide that it's a simple matter of using an index – the very first index as a matter of fact. We're using JavaScript here so we can get the value thus:

```
const nums = [1,2,3,4,5];  
const firstNumber = nums[0];
```

Now comes the question: *how complex was that operation?* If you were like me just a few years ago I would have said "not complicated at all – I just took the first element of the array". This is a correct answer, but we can be more specific by thinking about complexity in terms of *operation per input*.

We have 5 inputs here because there are 5 elements in the array. How many operations did we need to perform on these inputs to derive a result for our algorithm? *Only one* as it turns out. How many operations would we need to perform if there were 100 elements in the array? Or 1000? Maybe 1,000,000,000? Still: *only one*. We just take the very first element at index 0.

We can capture that inelegantly long paragraph in a more scientific way by saying that our algorithm was "on the order of 1 operation for all possible inputs", or better yet: **O(1)**. This notation is pronounced "order 1" or, more casually, *constant time*. For all inputs to our algorithm there is and will always be **only one operation required**.

As you've probably guessed, **O(1)** algorithms are pretty efficient and also quite desirable!

Iterations and Order(n)

Now that you can figure out how to pull the first item from our array, let's try something more complicated: *let's sum the items in the array.*

Again, let's use some code:

```
const nums = [1,2,3,4,5];
let sum = 0;
for(let num of nums){
  sum += num;
}
```

Now we get to ask ourselves the same question: *how many operations do we have per input to our algorithm?* This time the answer is different: we must add each number to a running sum, so we have to **operate on each one**. This means *one operation per input*.

Using Big-O time notation, we would say this is **O(n)**, or “Order n” where **n** is the number of inputs. This type of algorithm is also referred to as “linear”, or that it has “linear scaling” (think of describing a line on a graph: $y = 2x$ or something of the sort).

This type of scaling is common when you calculate a result by iterating over a collection of values like we're doing above. It's a simple way of doing things, but it has implications in terms of complexity which we can see if we ask ourselves a simple question: *how does this scale?*

As opposed to our **O(1)** algorithm, our CPU is doing a bit more work in our summing operation above – in fact it's doing *n* times the amount of work of our **O(1)** algorithm. If we have an array of 10 items, it won't matter; but what

happens if our array has 1,000,000 elements? Now we need to worry a bit as we have 1,000,000 operations to perform.

It might seem academic, but it's a good question to ponder whenever you write a loop (or worse: a *nested loop* which we'll address in the next section): "is there a way I can make this algorithm a bit more efficient?". For our summing operation - *no*, there isn't. We must consider every element. For other things, however, sometimes a bit of math might do the trick.

Consider our very same array: `[1,2,3,4,5]`. What if I told you to write a summing function for a sorted, contiguous array of integers that starts with the number 1? Ahhhh this time things are different as we know a bit more about our input.

We can use a bit of interesting math here, specifically something that Carl Friedrich Gauss figured out while in grade school. If you want the full explanation, follow the link above; otherwise I'll just get to it.

We can use this equation to figure out the sum of the series $[1...n]$:

$$S = n(n + 1)/2$$

Plugging this into our example array, we would have:

$$5(5 + 1)/2 = 15$$

How do we know what n is? Simple! It's the very last element of the array. Now we can change our algorithm a bit:

```
const sumContiguousArray = function(ary){
  //get the last item
  const lastItem = ary[ary.length - 1];
  //Gauss's trick
  return lastItem * (lastItem + 1) / 2;
}
const nums = [1,2,3,4,5];
const sumOfArray = sumContiguousArray(nums);
```

The answer here will be 15!

Notice that we're not running an iteration? That makes our $O(n)$ algorithm *a lot faster* as we're now in constant time thanks to Mr. Gauss. You might be thinking “but wait, if we have to figure out the length of the array *and* pull off the last element *and* run the calculation – isn't that $O(3)$ or something?”.

This is something we should get straight about Big-O right up front: *yes* that would be the literal complexity, but we're not interested in that. All we care about is that it's *constant time*, meaning that the time complexity will not change based on the number of inputs.

Constant time algorithms are always referred to as $O(1)$. Same with linear time. An algorithm might literally be $O(n + 5)$ but in Big-O that's just $O(n)$

The Not-so-good Approach: Order (n^2)

Let's up the complexity a bit. It was suggested in today's standup that I don't make good integer arrays, and that it's possible that I might have duplicated one of the elements. I insisted that I did not! However, we decided it might be a good idea if you were to create a routine to verify this.

There are some simple solutions to this, and some that are quite a bit more efficient. Let's start with the simple, brute force solution which require a nested **for** loop:

```

const hasDuplicates = function (nums) {
  //loop the list, our O(n) op
  for (let i = 0; i < nums.length; i++) {
    const thisNum = nums[i];
    //loop the list again, the O(n^2) op
    for (let j = 0; j < nums.length; j++) {
      //make sure we're not checking same number
      if (j !== i) {
        const otherNum = nums[j];
        //if there's an equal value, return
        if (otherNum === thisNum) return true;
      }
    }
  }
  //if we're here, no dups
  return false;
};

const nums = [1, 2, 3, 4, 5, 5];
hasDuplicates(nums); //true

```

It works, but it's not ideal. Here we're iterating over our array, which we already know is $O(n)$, and another iteration inside it, which is another $O(n)$. For every item in our list, we must loop our list again to calculate what we need. This type of complexity is $O(n^2)$, or "Order n squared" and as you can imagine, it's not very efficient and is considered quite inelegant.

The big problem is this: if we have 1000 numbers in our list, we'll have $1000 * 1000 = 1,000,000$ operations! That's bad.

If you remember only one thing from this chapter, let it be this: *there is almost always a better way!* In fact, this is one of those things where once you see it, it's hard to *not see it* again. Nested loops working over the same collection - always $O(n^2)$.

These problems often appear in coding interviews, where the simplest answer is to brute force your way through an

“ n^2 ” solution, trying to figure out how it’s possible to do things better.

Which you might be wondering right now... so let’s play Big Job Interview!

Me: *well, this solution works, true, but I wonder if we can do better? Is there a way we might be able to improve on your $O(n^2)$ solution?*

You: *possibly –*

Me: *sure, chicken. If it makes you feel better to be told an answer as opposed to figuring it out yourself... then...*

You: *passive-aggressive bullying isn’t cool Rob*

Me: *sorry... I don’t know the answer either...*

In a few sections we’ll revisit this problem and see if we can improve it.

Refining to Order($\log n$)

We’ve been relying on a brute force approach to work with our array up to this point which works, but as I keep saying it’s 1) inefficient and 2) inelegant. We want to do both of those things, so we don’t just look the part of super stellar programmer!

For the next task, let’s search through our array for a given number. If you don’t have a CS degree (having a few search algorithms at the ready) – how would you go about such a thing. Once again: *yes, we have eyes*, but a program doesn’t, so we need to have a systematic (and deterministic) way of finding a given value in a collection.

The simplest way to go about this process is to think about the *time complexity* of the operation, which is what we’ve been doing. Instead of offhandedly saying “we’ll have to search the array top to bottom” or “just iterate until you find the number”, we can now use a simpler, more direct term: **$O(n)$** .

If we use a brute force approach, we just loop over the entire array *once* to find the number we're looking for. This means that in the worst-case scenario (which is how we think when we use Big-O) we must perform one operation per input.

Can we do better than a linear $O(n)$? It turns out that yes, we indeed can if we make a few assumptions.

Assumption 1: we've noticed that the list is sorted; can we assume that it is sorted for this exercise? For illustration, I'll say *yes, we can*.

Assumption 2: I didn't mention anything about *in-place*; can we use more than just the given array? *Sure, why not*.

Given these two assumptions, we can use a clever algorithm called *binary search*. We'll get into the details later, but this algorithm is performed in a set of simple steps where we split the list in half, discard the half we don't need, then go on splitting until we have the value we're looking for. By splitting the list and discarding the half without our target value each time, **we're able to find the number we're looking for in far fewer operations**.

This type of algorithm is called "divide and conquer" and works on the mathematical principle of *logarithms*. If you're like me (just a year ago) you haven't had to think about logarithms in quite a few years and, perhaps, you've forgotten how they work.

Binary search is one of the simplest ways to improve the performance of our applications. I'll explain why (and how) in just a second but before I get there, we need to do some math.

Quick Logarithm Review

In essence, logarithms help make working with exponents a bit easier. Let's do some quick math and hopefully things will

be a bit clearer. What if I asked you to figure out what x is here:

$$x^3 = 8$$

To answer this you simply need to take the cube root of 8 to get 2. Simple enough! What if we changed things up now:

$$2^x = 512$$

This makes things a bit harder, doesn't it? How would you solve for x here? If you're a math person you already know the answer; but if you're like me and have forgotten most of your math the answer wouldn't exactly pop out at you. What we need here are *logarithms*.

If we used logarithms we could rewrite this equation like this:

$$\log_2(512) = x$$

This equation says “log base 2 of 512 equals some number x ”. The key here is the “base 2” part. It means that we're *thinking in 2s*, and we want to know *how many times to multiply 2 to arrive at 512*. That's all logarithms do! How do we solve this equation? Duh! With a calculator of course!

The important part here is the recognition that logarithms basically deal with splitting a given number into some type of *base*. The default base is 10, which means you should be able to guess at the answer of this expression:

$$\log(100) = x$$

The question you're asking here is *how many times do you multiply 10 by itself to arrive at 100?* The answer is 2!

OK, so let's get back to the problem at hand. We're splitting a list of numbers in half, continually, until we get down to the single value that we want. The key to this is the name of the algorithm itself: *binary* search... splitting things *in half*.

We're dealing with 2s which means our logarithm will be base 2. So this operation can be expressed like this:

$$\log_2(n) = x$$

Which brings us to an interesting thing about Big-O which I'm repeating now and I'll be repeating later: **the exact math doesn't matter**. What matters is that this is logarithmic - nothing else. We can even do away with the base and x - we just care that the operation is $O(\log n)$.

If we were to use an $O(n)$ scan on our list we would have had to perform $n = 5$ operations to find our target number 2. By splitting the list and evaluating the number at the split (thus discarding the half without our number), we could bring the number of operations down to 3.

You might be thinking "hey wait a minute! Splitting *and*

evaluating is 2 operations!” and this is true - but they are 2 $O(1)$ operations so we just consider the entire thing to $O(1)$.

Cutting from 5 operations down to 3 isn't terribly exciting. But what if we had 100 total elements? If you whip out a calculator and run some math, you'll find that:

$$\log_2(100) = 7$$

OK that's not exactly true. The number is something a bit more like 6.643856. But we can reason through this in a simpler way.

We're programmers and have a reasonable facility for base 2 “stuff”, hopefully. We know that powers of 2 are 2, 4, 8, 16, 32, 64 and finally 128. The number we want is somewhere between 64 and 128, so we *round up* to be safe. **This gives us 7.**

Why do we care? Because we can see how our search algorithm scales! We upped our inputs by a factor of 20, but our operations only went up by 4.

That is why we care about doing this entire exercise! We can prove, with math now, that our approach here is better than scanning the entire list for a given number.

Rethinking $O(n^2)$ with $O(n \log n)$

A few sections ago I asked you to find duplicate numbers in our list of numbers and we decided to iterate over every number (which is $O(n)$) and then do it once more to see if our current number was repeated.

This put an $O(n)$ operation inside another $O(n)$, which

we can simplify to $O(n * n)$ which is $O(n^2)$, which is a Bad Thing.

But wait a minute... didn't we just use a nifty algorithm to find a number within our list that was much more efficient than our brute force linear scan? Yes!

So here's a quiz: *what is the Big-O* of using our binary search inside our linear scan? Let's break it down.

Iterating and Searching

We have an $O(n)$ as before with our array iteration – we know that much. We now have a single operation within the iteration that is $O(\log n)$. Putting the two together we have $O(n \log n)$ which we can just think of now as $O(n \log n)$.

So: in terms of time complexity, we have a better algorithm here! It will provably scale better than our original brute force approach as well!

```
const nums = [1,2,3,4,5];
const searchFor = function(items, num){
  //use binary search!
  //if found, return the number. Otherwise...
  //return null. We'll do this in a later chapter.
}
const hasDuplicates = function(nums){

  for(let num of nums){
    //let's go through the list again and have a
    //at all the other numbers so we can compare
    if(searchFor(nums,num) !== null){
      return true;
    }
  }
}
//only arrive here if there are no dups
return false;
}
```

I mentioned earlier that binary search is one of the simplest ways that we can, as programmers, increase the performance of our applications. This might have sounded weird, but consider how your database finds information when you run a search:

```
select * from users where email = 'test@test.com'
```

This query will work fine for a while, but as your users table grows, this query will slow down. The simple reason why is that the database is a program like any other and doesn't have eyes. Therefore, it has to loop over every record in your table until the condition is met, which in our case is matching a user's email. That process has $O(n)$ time complexity.

When queries like this slow down the solution is to apply an index. In our case we would pop one on the users table like so (assuming you're using something Postgres, which you should be!):

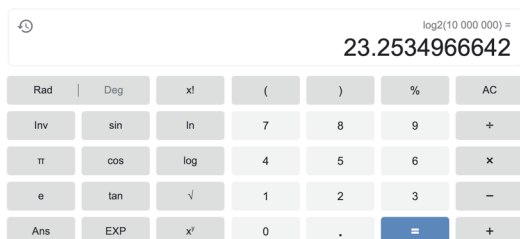
```
create index idx_user_email on users(email);
```

The index is a sorted list and that's it! Every email in your users table is added there and then sorted. Then, when you run the query above, that index is searched using something very like binary search. This process has $O(\log n)$ time complexity.

Why do we care about this? Let's pretend we have 10,000,000 users in our database. To match an email on a user, our database would have to perform up to 10,000,000 operations. Not so good.

Using an index, however, our database has to perform

$\log_2(10,000,000)$ operations which we can ask Google to help us solve:



Rounding up, we can say that we improved performance by quite a few orders of magnitude for that one query! 24 operations vs. 10,000,000 – that’s pretty good!

Our interviewer is digging our style right now and decides to up the game, one more time.

What if I told you that you could assume there was only one duplicate number?

Well now... that's an interesting development. Let's consider everything that we just did and see how we can bring this $O(n \log n)$ algorithm down just a bit more.

The first question we should ask ourselves is *can we do some math* to avoid an iteration? A few sections ago we used Gauss' trick to sum a series of integers, which could help us here. To achieve that, we'll need to perform three operations:

- Find out what the sum should be for an array without duplicates. For this all we need is the highest number in the array passed to us. Since we

know it's sorted, that's the last element and we know this is a constant time thing at $O(1)$.

- Find out the actual sum. This will necessarily be an $O(n)$ scan.
- Subtract the non-duplicate sum from the actual sum and that's the number we're looking for! This is also $O(n)$.

If we're right, this will bring our algorithm down to $O(n)$. Here's our new algorithm:

```
const findDuplicate = function(ary){
  //sum what we're given
  let actualSum = 0;
  //our O(n) scan
  ary.forEach(x => actualSum += x);
  //get the last item
  const lastItem = ary[ary.length - 1];
  //create a new array
  const shouldBe = lastItem * (lastItem + 1) / 2;
  return actualSum - shouldBe;
}
const nums = [1,2,3,4,4,5];
const duplicate = findDuplicate(nums);
```

Nice work! Our interviewer is impressed!

Thinking in Big-O

By now you should be able to equate certain basic operations in code to a given Big-O. If you practice this for a bit, you'll be able to quickly spot patterns and, ideally, improve them if you know the right algorithm to do so.

Specifically:

- **Random access to a given element in a collection is always $O(1)$** , depending on how the list is indexed. Arrays, for instance, allow you to access elements randomly if you know their index. HashSets allow you to access if you know what the value is (the hashed value is the index). Dictionaries allow you random access if you know what they key is, and so on. These types of operations are always $O(1)$ which means if they are combined with other Big-O, they will remain static or constant time.
- **List iterations are always $O(n)$** . If you need to evaluate every item in a list for a given algorithm, this means it will at least be $O(n)$. Sometimes you can get around this with some trickery, which I'll discuss later on.
- **Nested Loops on the same collection are always at least $O(n^2)$** . Loops within loops ... sometimes necessary, but can usually be improved by thinking about data structures (which we'll do later on).
- **Divide and Conquer is always $O(\log n)$** . The very act of dividing a list into smaller sublists is logarithmic. If you have an $O(1)$ operation once the list is split apart, then the Big-O for the entire operation is $O(1 * \log n)$ which is just $O(\log n)$.
- **Iterations that use divide and conquer are always $O(n \log n)$** . Think about looping a list and then executing some algorithm to search for list value or, possibly, to run some kind of sorting.

If you solve a problem by adding another nested loop for every input that you have: that's $O(n!)$ which is bad and you should probably find another job!

Space Complexity vs. Time Complexity

You may have noticed that I've been using the term *time complexity* a lot in this chapter, as that's what we've been focused on: *how long will it take to do a given operation given n inputs*. In data analysis speak this is called a *dimension* and is just a way to think about how complex an algorithm is.

There is another dimension that is also important: *space*. In other words: **what are your algorithms resource requirements?**

The same type of Big-O classifications still apply in that we still refer to things such as $O(n)$ or $O(1)$ "space", but the meaning is somewhat different. Space where? The answer is: *it doesn't matter*. "Computer resources somewhere" is all we care about.

However. There is a bit of reasoning we should be able to apply so we can go a bit deeper.

When a program executes it has two ways to "remember" things: the *heap* and the *stack*. I'll get into the details of each later on, but for right now just consider the stack to be the thing we're worried about. It's the thing that remembers the variables *in the scope of the currently executing routine*. When a variable is declared in a block of code, it's stored on the stack. When a block of code goes out of scope, the variables are removed from the stack. Sometimes the scope is the current block, other times it's the current function or procedure.

Why do we care? The short answer is that **you can easily run out of resources before you run out of time** depending on how you've written your program. The simplest way to think about this is the dreaded "Stack Overflow Exception" which simply means you're executing some kind of loop that has used up every last bit of space on the stack. This can (and often does) happen with a recursive

routine as each value remains on the stack until all functions have executed.

Working with strings is another way to cause yourself space complexity problems. For example, if you use a loop to build a string, your space complexity might be as bad as $O(n * m)$ where n is the number of iterations and m is the length of the string. Not so bad if you're just building out memes, but if you're trying to evaluate string patterns in a book... that could be bad.

DATA STRUCTURES

ARRAYS, HASHES, TREES, GRAPHS AND MORE

Now that we understand how to quantify complexity, let's investigate the various ways that we can work with data to increase efficiency. As I mentioned at the beginning of the previous chapter: *algorithms, complexity and data structures work together*.

ARRAYS ARE simple to work with but make life hard if you're trying to find a particular value. Dictionaries help with that but are slightly more involved. Hash Sets (or Hash Tables) are lovely because the hash of their value is also their key, so accessing data is constant time and is $O(1)$ if you know the value you want. From there things get quite a bit more involved.

Some languages offer a plethora of choices when it comes to working with data, but all these choices are variations of (or small improvements upon) a standard, core set of data structures that every programmer should know. That's what we'll look at in this chapter.

The Code

You can find the code used in parts of this section up at my GitHub repo. In addition, you can buy over 10 hours video walkthroughs of the structures you see in this chapter and others from here.

Array

You know about arrays, I'm sure. Unless you work in JavaScript every day, however, it's unlikely you use them much. They're very lightweight and simple to use, but there is also weirdness.

When you declare a non-dynamic array, you must specify its size upfront. This size cannot change. In most modern languages we don't think about this — but in some older languages this is still the case.

Arrays hold values which are referenced by an index. They allow very fast random access, which means you can access any value from an array using a $O(1)$ routine as long as you know the index.

Arrays are bound to a specific length once they are created. If you need to add an item to an array, the language you're working in will typically copy the original plus whatever value you want to add to approximate dynamically changing the array's size.

This can be a costly operation, as you can imagine. Let's see why.

Many languages allow you to dynamically resize an array (Ruby and Python, e.g.) while other, more strict languages, do not (C# and Java e.g.). Ruby and Python allow for dynamic arrays which are basic arrays, but with some magic behind the scenes that help during resizing. C# and Java have

different structures (like Lists) built specifically for expanding and shrinking.

Arrays are allocated in adjacent blocks of memory when they are created. There is no guarantee that additional memory can be allocated adjacent to an array if you need to add an element, so it becomes necessary to copy and rebuild the array at a new memory location. This can be costly.

Strings, for instance, are simply arrays of characters. If you append a string with another string value in C#, for instance, an array copy/rebuild needs to happen. This is why working with strings in a loop is typically not a good idea.



JavaScript, however, is an interesting case. Arrays in JavaScript are just objects with integer-based keys that act like indexes. They are instantiated from the Array prototype, which has a few “array-like” functions built into it.

Here is a description from Mozilla:

Arrays are list-like objects whose prototype has methods to perform traversal and mutation operations. Neither the length of a JavaScript array nor the types of its elements are fixed. Since an array's length can change at any time, and data can be stored at non-contiguous locations in the array, JavaScript arrays are not guaranteed to be dense; this depends on how the

programmer chooses to use them. In general, these are convenient characteristics; but if these features are not desirable for your particular use, you might consider using typed arrays.

Hey, it's JavaScript.

Arrays are the simplest data structure with the least number of rules. Bits of data are stored *in contiguous memory*, so they also have the smallest footprint of any data structure.

If all you need is to store some data and iterate over it, arrays are a fine choice, especially if you know the indices of the items you're storing. Random, $O(1)$ access to values in an array is also a great reason to choose an array over other, more "ceremonial" data structures.

The final reason to keep arrays fresh in your mind is The Big Job Search. Coding interviews will almost *always* involve working with an array of values at some level (usually integers), so understanding their restrictions (and advantages) is essential.

Stack

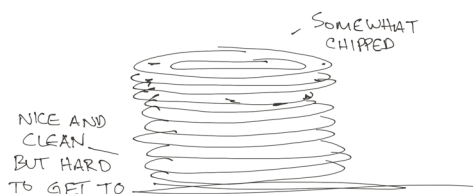
I discussed the idea of a stack in the previous chapter when describing a Pushdown Machine. It's like an array, but it has a few restrictions:

- You can't access items randomly using an index
- You can only add and retrieve items in the stack from the "top"
- It has three explicit methods: Push, Pop, and Peek.

A term that is often used for the nature of a stack is "Last

In, First Out” or LIFO. You *push* an item onto a stack, *peek* to see the value that’s currently on top of the stack and then *pop* that item off the stack.

The simplest way to think about the stack is by visualizing one of my big pet peeves: a stack of plates. The ones on the bottom never get used because we always grab the plates on the top!



Stacks are useful in several surprising ways. They are perfect when you need to know the very last value that was seen in a loop. You might want to know this when you’re:

- Reversing a string
- Traversing a graph or a tree
- Checking an opening/closing structure of some kind (such as balanced parentheses in a sentence)

Honestly, the biggest reason you want to know what a stack is (and does) is for interviews. They come up often! You might not use them regularly in your current job, but you never know when they might fit something you’re trying to do.

Queue

A queue is also like an array, but with some additional rules:

- You can't access values randomly using an index
- You can only add values in one end and retrieve them from the other
- It has two explicit methods: Enqueue and Dequeue

A queue queues things in a queue, and is described as “First In, First Out” or FIFO. You *enqueue* an object into the “end” of the queue and *dequeue* it from the “front”.

It's likely you've used queues often, so I won't labor the point for too long. It's also a word that's quite tough to type, so I'll keep this section short.

An interesting thing about a queue is that you can create one with two stacks. This is one of those interview questions that you'll likely need to know at some point in your career! How would you go about doing this?

The mechanism is obvious: you have an “in” stack and an “out” stack — the tough part is deciding at which point you'll move items from the “in” stack onto the “out” to avoid collisions. If you do it one at a time then things can easily get out of order! Imagine calling *enqueue* 5 times, then *dequeue* once, then *enqueue* again. Your stacks would be a mess!

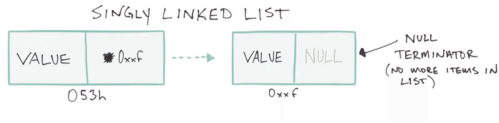
How would you avoid this? I'll leave it to you...

Linked List

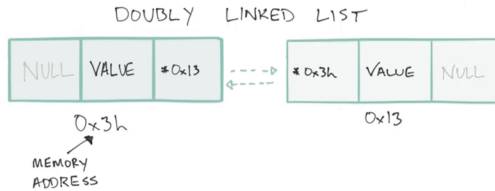
There are two types of linked lists: singly and doubly linked. Both are graphs (which we'll discuss in a minute), which means you can think of them as two-dimensional structures with associations.

A singly linked list consists of a set of “nodes” in memory, that have two elements:

- The value you want to store
- A pointer to the next node in line



A doubly linked list is the same, but contains an additional pointer to the previous node.



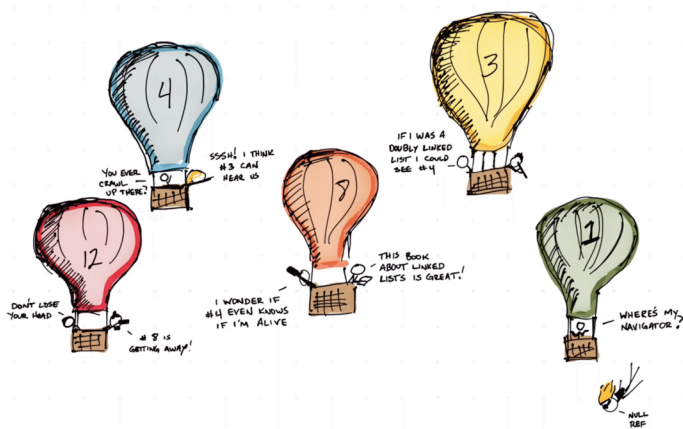
Nodes in a linked list don't need to reside next to each other in memory, and therefore can grow and shrink as needed.

Their loose structure allows you to insert values into the “middle” of the list by simply resetting a few pointers. Same for deleting a node in a linked list.

The downside to linked lists is that finding an item means you must traverse the list to get to the item you want, which is an $O(n)$ operation. Arrays, on the other hand, allow you $O(1)$ access if you know the index.

Linked lists are null terminated, which simply means if a node's pointer is null, then that signifies the end of the list.

The first item in a linked list is the *head*, the rest of the list is referred to as the *tail*. Some languages refer to the tail being the last item in the list — there is no hard definition so if you hear about “tailing the list” just think about the end of it and hope for some context.



The primary reason to choose a linked list over something like an array is *simplicity* and the ability to grow and shrink as needed. Furthermore, because you're in an interview... which might sound horribly snarky but... well, it's true.

Working with them can be a bit weird, but also kind of interesting in that they are self-contained and lightweight:

```
class LinkedListNode {
  constructor(value){
    this.value = value;
    this.next = null; //termination
  }
}

const a = new LinkedListNode(1);
const b = new LinkedListNode(2);
const c = new LinkedListNode(3);
```

Iteration is best thought of as “traversal” because you don’t really know when a linked list will end — so you end up using something like a **while** loop:

```

a.Next = b;
b.Next = c;

let thisNode = a;
while(thisNode != null){
  //do something
  //...
  //traverse
  thisNode = thisNode.Next;
}

```

Hash Table

Arrays are fast for reading data, linked lists are good for writing data and having more flexibility. A hash table is a combination of the two.

A hash table stores its data using a computed index, the result of which is always an integer. The computation of this index is called a *hash function*, and it uses the value you want to store to derive the key:

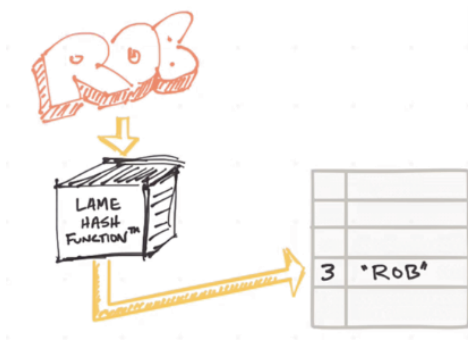
```

const value = "Rob";
const myLameHashFunction = function(val){
  return val.length;
}

```

Most modern programming languages have a hash function you should use; don't create your own. I'll go into why in just a second.

Once you have a key, you can store your value. The key/value pair is referred to as a *bucket* or *slot*.



Hash tables are great when you want quick access to certain values. Because their value is also their index, hash table reads are $O(1)$. This can be complicated, however, if a hashing function is overly complex. You will typically leave this to whatever framework or language you're using, so assuming $O(1)$ is fine.

Similarly, adding values to a hash table does not involve (typically) any traversals — you just hash the value and create the key. This leads to excellent performance, however in the real world this doesn't happen that often.

Even the best hashing algorithms will create duplicate keys (called collisions) if the data size is large enough. When this happens, your reading and writing can be reduced to $O(n/k)$, where k is the size of your hash table, which we can just reduce to $O(n)$.

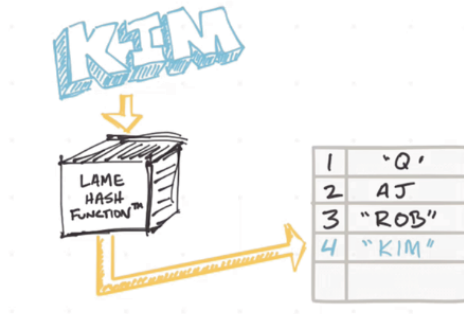
Collisions will likely happen in any hash table implementation, so most implementations have built-in ways of handling these. Let's examine two common ways to deal with collisions: open addressing and separate chaining.

Open addressing resolves a collision by finding the next available slot and dropping the value there.

Let's assume I'm using my Lame Hash Function with a

new name: “Kim”. The key produced will be a 3 since my Lame Hash Function only uses the length of the value instead of something more intelligent. The key produced (3) will collide with my existing entry for “Rob”, which is also a 3. Using open addressing to resolve the collision, “Kim” will be added to the first slot at the end, which is 4.

To find “Kim” in the table, open addressing dictates that we do an $O(n)$ scan from index 3 (where “Kim” should be) and then work our way down until we find the value.



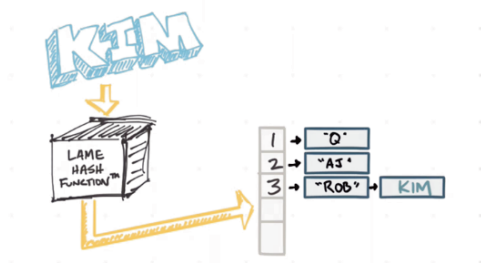
This presents some interesting problems — what if “Doug” and “Dana” want to join our list? They can, but their indexes will be 5 and 6. It spirals from there.

This is called *clustering*, and it can quickly turn a lovely, fast hash table into a simple $O(n)$, not fast hash table.

Separate chaining involves combining two data structures that we’ve already looked at: arrays and linked lists.

When a key is hashed, it’s added to an array that points to a linked list. If we have only one value in our hash table for a given key, then we’ll have a linked list with only a single element.

However, in the case of key 3, the linked list can expand easily and accommodate “Kim” as well:



Hash tables are one of the most common data structures you'll find in modern languages because they are fast. As long as the hashing algorithm is comprehensive and capable, the hash table will be $O(1)$.

One of the main drawbacks, however, is again the hashing algorithm. If it's too complex, then it will take longer to run, and you will still have $O(1)$ read/write, but it could actually be slower than executing an $O(n)$ over an array, for instance.

Dictionary

A dictionary is exactly like a hash table except it has a unique key for accessing a given value. This has an advantage over hash tables in that you *can't have a dictionary with the same key* — so collisions are not something you need to worry about.

Given that a key is guaranteed to be unique, dictionaries provide $O(1)$ access to any element, as long as you know what the key is. The downside is that dictionaries can be a bit larger than hash tables and therefore have increased *space complexity*. Most of the time this isn't something you need to worry about.

Dictionaries are ubiquitous in programming, even more so than arrays and hash tables. If you're a JavaScript developer you might be wondering about this assertion — after all,

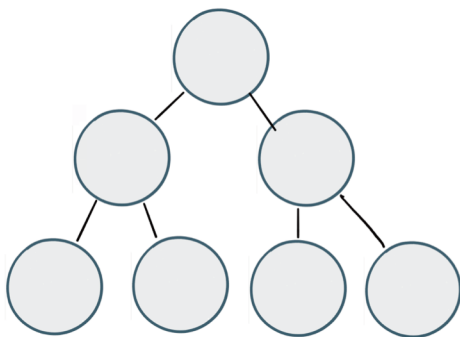
Figure 1 The Author's Jekyll Blog

Using a tree data structure usually involves traversing using recursion, which can be expensive in terms of *space complexity*. Every time you recurse a child node, the stack has to store the values for that scope, and you could run out of space.

There are other ways to traverse a tree by iterating using a Queue and a Stack — I'll get to that in the Algorithms chapter, specifically Breadth-first and Depth-first traversal.

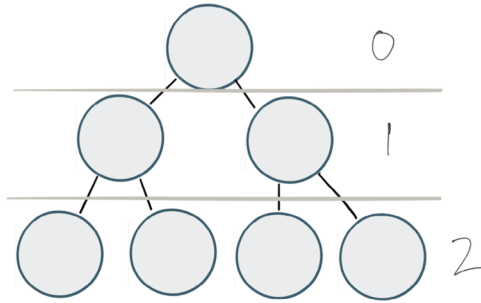
Binary Tree

A binary tree is something you've likely seen. It's a type of graph, but with some rules applied, which are straightforward: each node can have 0, 1, or 2 *child* nodes, but only 1 parent:



Binary trees can represent several interesting things, including *decisions*. Each node represents a given state that is the result of a yes or no decision.

Another interesting aspect of a binary tree is that each level of the tree represents a *logarithmic value*:



At level 0 we have $2^0 = 1$ Node. At level 1 we have $2^1 = 2$ and finally $2^2 = 4$ nodes.

You can also visualize the *divide and conquer* search algorithm we used in the last chapter with a binary tree. Each split operation represents another level of a binary tree.

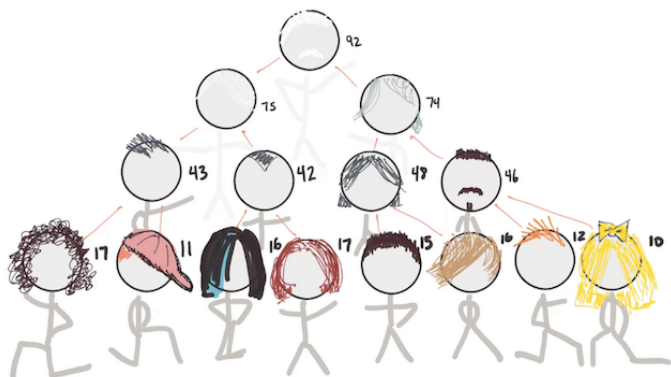
Heap

A heap is an inverted tree that is a set of interconnected nodes that store data in a particular way, which is called the *heap property*.

Every child node belongs to a parent node that has a greater priority (or value, whatever) — this is called a max heap. A min heap is the opposite: every parent has a value less than each of its children.

Put another way: if we're dealing with a max heap then every node on a top level has a greater value than every node on the next-level down.

We can see this in a recent family picture of mine, where we all posed based on our ages:

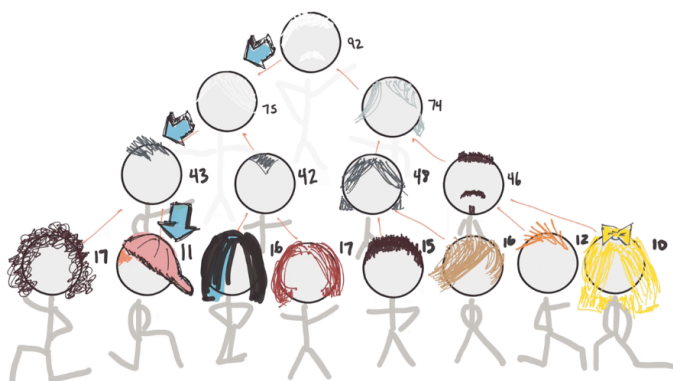


We couldn't afford a camera, so we had to draw it out

Usage

A heap (and its variants, which I'll get to later) can be used in any algorithm where ordering is required. Arrays are random and allow random access to any element within them. Linked lists can change dynamically but finding something within them is $O(n)$ (linear); heaps are a bit different.

You can't do $O(1)$ random access and a single node knows nothing about its children. This means you need to do some type of traversal to find what you're looking for. Given the structure of the tree, however, finding things is considerably easier than with a linked list:



We found Tommy, age 11, easily here. However, we could easily have had to traverse over a few times if he was on the end there next to Jujubee, age 10.

So, what are heaps good for then? It turns out they're wonderful if you're doing comparative operations — something like "I need all people in this heap with an age greater than 23". Doing that in a linked list would be quite slow — the same with an array and a hash as no order is implied.

Heaps are used in data storage, graphing algorithms, priority queues and sorting algorithms. In many languages you'll find data structures that will give you the same kind of ordered structure if you choose your keys wisely.

For instance: in C# you could use a **SortedList** or a **SortedDictionary**, storing objects with parent/child associations.

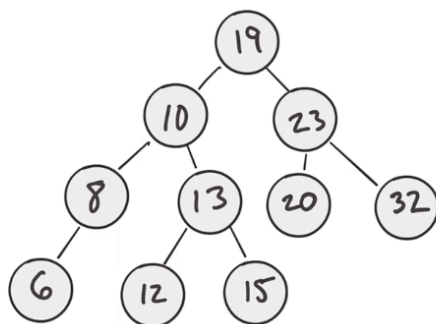
The primary advantage of heaps is performance. They're very fast when it comes to searching as they are pre-sorted and don't involve the extra step of an $O(\log n)$ split — it's just a traversal.

Binary Search Tree

A binary search tree is just like a heap in that it is organized based on the values of the nodes, with one major addition: there is also a left to right value priority.

The rules are:

- All child nodes in the tree to the right of a root node must be greater than the current node
- All child nodes in the tree to the left must be less than the current node
- A node can have only two children



Binary Search Tree

The advantage of a binary search tree is, obviously, searching. It's very easy to find what you're looking for as you'll see down below. The downside is that insertion/deletion can be time-consuming as the size of the tree grows.

For instance, if we remove 13 from the tree above, a decision needs to be made as to which node will ascend and take its place. In this case, I could choose 15 or 12. This operation seems simple on the face of it, but if 15 had children 14 on

the left and 16 on the right, some reordering would need to happen.

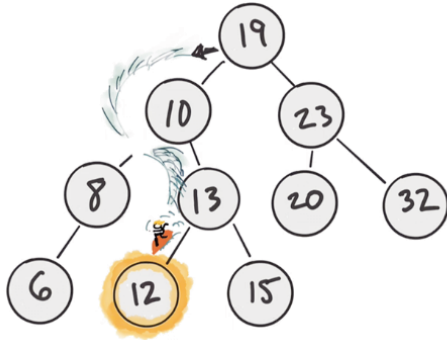
If you know the rules, finding a value in a BST can be done recursively, or by using breadth-first or depth-first search (which we'll get to in a minute):

- If the value is less than the current node you're on, go to the left child node.
- If the value is greater, go to the right child node.

Do this until you find what you're looking for.

For example, here we need to find the number 12. Our root node is 19, so we traverse to the left because $12 < 19$. When we reach 10, we traverse right because $12 > 10$.

Finally, we come to 13, so we go to the left again and arrive at 12.

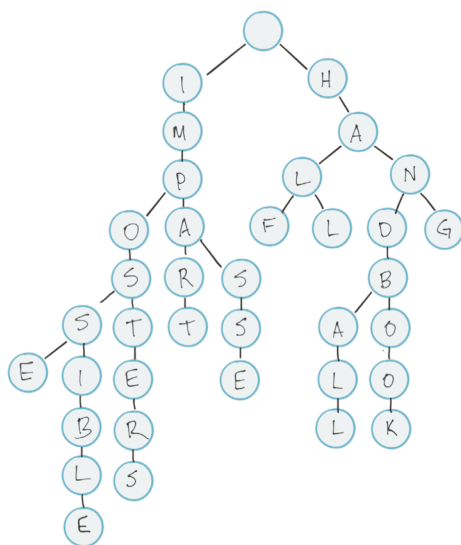


Digital Tree (or Trie)

A digital tree (or “trie”, which most pronounce “try” even though the term comes from **retrieval**) is a specialized tree used in searching, most often with text. In many cases, it can

outperform a binary search tree or hash table, depending on the type of search you're doing.

Tries allow you to know if a word (or part of a word) exists in a body of text. The easiest way to understand a trie is to see one, so here goes:



This trie has an empty root node, and from there letters are added as child nodes. The power of a trie is evident with the prefix “imp” — in this section of the trie there are 6 distinct words represented by 20 total nodes:

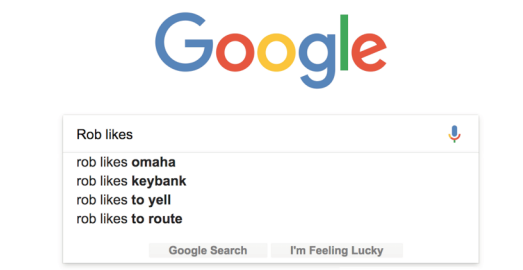
- Imp
- Imposter (and Imposters)
- Impossible
- Impart
- Impasse

The advantages of this structure are speed and space. Finding a word in this structure is $O(m)$, where m is the length of the word you're trying to find.

Tries also have a major advantage when it comes to *space complexity*. Common prefixes are reused, so repetition within the structure is kept at a minimum.

Finally, the major advantage of a trie is that you're able to search the structure for *partial matches* based on a prefix. This kind of thing is great for word completion.

Have you ever wondered how code completion works in your favorite editor, or how Google can do the below so quickly?

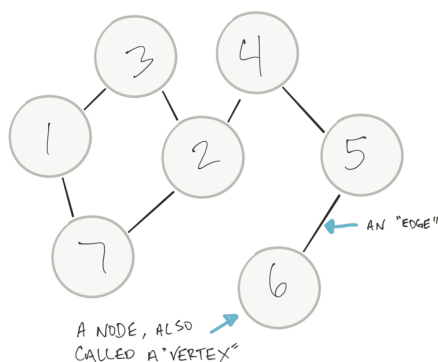


That's right: using tries.

Graphs

I've been mentioning graphs quite a lot in this chapter, and it's time, finally, to dive in and get to know these data structures a bit more.

Graphs are one of the most useful and most *used* data structures in computer science. In short, a graph is a set of values that are related in a pair-wise fashion. Again, the easiest way to understand this (if it's not intuitive) is to see it:

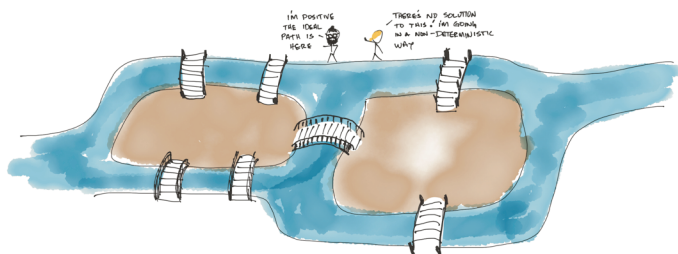


As you're probably figuring — graphs can describe several things. Let's look at a few.

If you took calculus in school, you probably learned about the origins of graph theory with Leonhard Euler's Seven Bridges of Königsberg problem:

The city of Königsberg in Prussia (now Kaliningrad, Russia) was set on both sides of the Pregel River, and included two large islands which were connected to each other and the mainland by seven bridges. The problem was to devise a walk through the city that would cross each bridge once and only once, with the provisos that: the islands could only be reached by the bridges and every bridge once accessed must be crossed to its other end. The starting and ending points of the walk need not be the same.

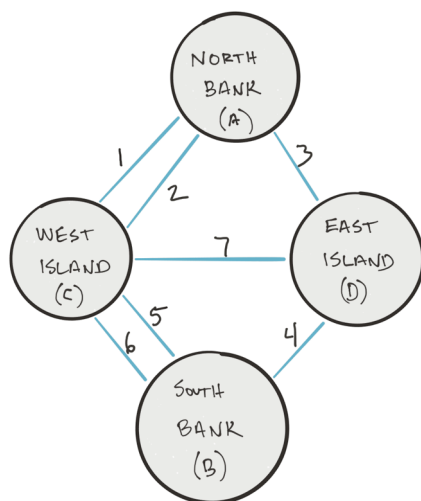
It might be easier to visualize the problem ... which gives me another reason to draw!



The Seven Bridges of Königsberg.

In trying to solve this problem, Euler reasoned that the route on land was not important, only the sequence in which the bridges were crossed. Given this, he could relate the land to the bridges in the abstract, using the idea of nodes that are accessible by an edge.

In other words, *a graph*.



Now that we have our graph, we can restate the problem in terms of a graph:

Can you visit vertices A, B, C and D using edges 1 through 7 only once?

Before reading on, take a second and see if you can solve the problem just tracing your finger across the page. Or draw it out on a paper yourself and see if you can trace a line using a pencil or pen, visiting nodes A through D using edges 1 through 7 only once.

In mathematical terms, a *simple path* accesses every vertex. An *Euler path* will access every edge just once — that's the one we want, an Euler path.

There is an algorithm we can use to solve this problem, which is to determine the number of degrees each vertex has and apply some reasoning. A degree is how many edges a given vertex has, by the way.

Euler reasoned that a graph's degree distribution could determine whether a given edge must be reused to determine the path. His proof, in short, is that a graph must have either zero or two vertices with an odd degree to have an Euler path (a path which visits each edge just once).

Let's tabulate the seven bridges graph:

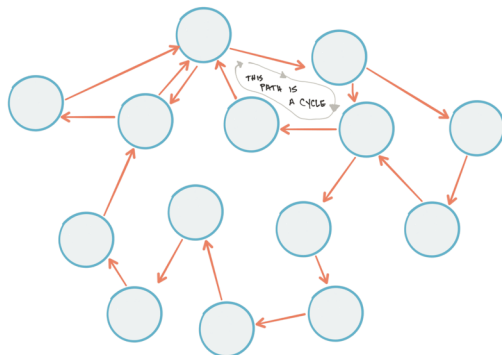
VERTEX	DEGREES
A	3
B	3
C	5
D	3

Here we have four vertices with odd degrees, which tells us that there is no Euler path and, therefore, that the Seven Bridges Problem has no solution.

I wouldn't blame you if you're wondering why I've included the Seven Bridges Problem in this book as well as a discussion of Euler. These seem to be more math-related than anything in computer science, don't they?

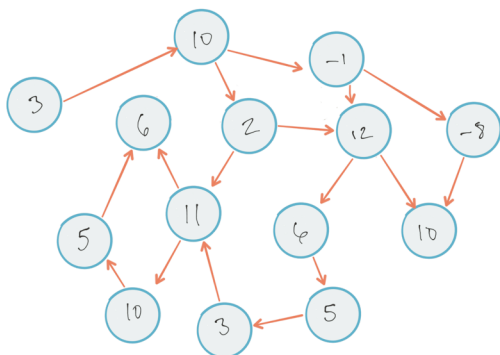
In your next job interview you may very well be asked how you would solve some algorithmic problem that you (hopefully) will recognize as graph based. Fibonacci, Traversal, Balancing, or a Shortest Path problem — if you can spot a graph problem, you'll have a leg up on the question.

There are different types of graphs, as you can imagine. One which you'll want to be familiar with is a directed graph, which has the notion of direction applied to its edges:



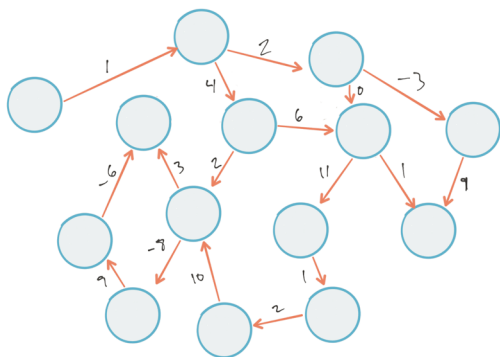
These types of graphs are useful for describing traffic flow of some kind, or any system in which movement is not bi-directional in every case. There is also an undirected graph which you can think of as a series of two-lane highways that connect towns in a countryside. Travel is bidirectional between each town and not directed along a given path.

Values can be applied to various aspects of a graph. Each vertex, for instance, might have a weight applied to it that you'll want to use in a calculation of some kind:



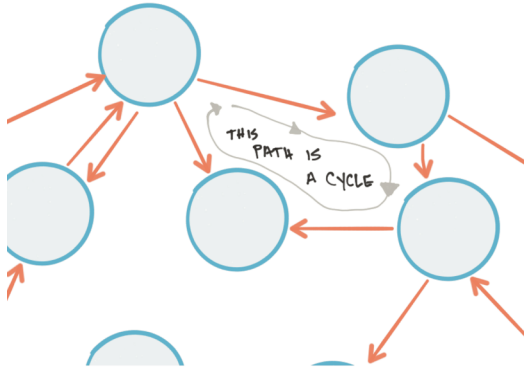
Maybe you're going on a trip, trying to figure out the most efficient way to see the cities you like the most.

There is also an edge-weighted graph, which is useful for calculating optimal paths:



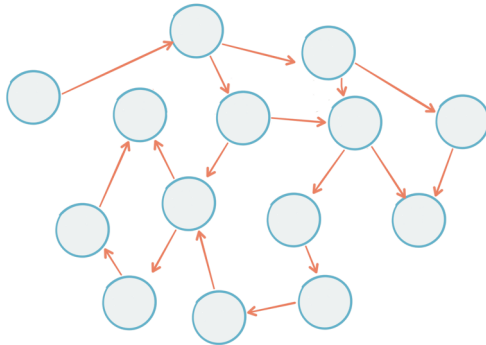
When you have vertices connected circularly, it's called a

cycle. We can see one of these as part of our first graph above:



Cycles are common in directed graphs. If a graph doesn't have a cycle, however, it has a special name.

If we redraw the graph above with edges that don't cause any cycles, we'll have a directed acyclic graph, or DAG:



DAGs are useful for modeling information and processes, like decision trees, algorithmic processing or data flow diagrams. Many of the data structures that we've been

playing with up to now have been DAGs, including all the trees and linked lists.

SIMPLE ALGORITHMS

SORTING, SEARCHING AND GRAPH TRAVERSAL -
THINGS YOU NEED TO KNOW FOR AN INTERVIEW

You don't need to know how to write a sorting or searching algorithm from scratch, frameworks do that for us. You do, however, need to know *how they work* because 1) it's likely you will be asked some details about them during interviews and 2) understanding their complexity could be the difference between keeping and losing your job!

The Code

You can find the code used in parts of this section up at my GitHub repo. In addition, you can buy over 10 hours of video walkthroughs of the algorithms you see in this chapter and others from here. I'll be using screenshots once again for the code samples for formatting reasons – if you want to play along please do... but you'll need the code from GitHub.

Bubble Sort

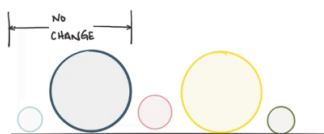
Let's start with the simplest sorting algorithm there is: bubble sort. The name comes from the idea that you're "bubbling up" the largest values using multiple passes through the set.

So, let's start with a set of marbles that we need to sort in ascending order of size:



A reasonable, computational approach to sorting these marbles is to start on the left side and compare the first two marbles we see, moving the larger to the right.

As you can see, the smaller marble is already on the left so there's no change needed:



Then we move on to the next two, which are the dark blue and the pink, switching their positions because pink is smaller than dark blue:



The same goes for yellow and dark blue, although an argument could be made that the author's drawing skills don't make it clear that dark blue is slightly larger.



The last two are simple: the green marble is much smaller than the dark blue, so they switch positions as well.



OK, we're at the end of our first pass, but the marbles aren't sorted yet. The green is out of place still.



We can fix this by doing another sorting pass. This one will go a bit faster because blue and red are in order, red and yellow are in order, but green and yellow are not – so we make that switch:



Not too hard to figure it out from here. The green ball needs to move 1 more time to the left, which means one more pass to sort the marbles – making 3 passes in total.

Eventually, we get there:



Bubble sorts are not efficient, as you can see.

JavaScript Implementation

Implementing bubble sort in code can be done with a loop inside a recursive routine. That sentence right there should raise the hairs on the back of your neck! Indeed, bubble sort is not very efficient (as we'll see in a minute).

Here's one way to implement it:

```

//the list we need to sort
const list = [23,4,42,15,16,8];

const bubbleSort = (list) => {
  //a flag to tell us if we need to sort this list again
  var doItAgain = false;
  const limit = list.length;
  const defaultNextVal = Number.POSITIVE_INFINITY;
  //loop over the entries
  for (var i = 0; i < limit; i++) {
    let thisValue = list[i];
    let nextValue = i + 1 < limit ? list[i+1] : defaultNextVal;
    //compare values
    if(nextValue < thisValue){
      list[i] = nextValue;
      list[i+1] = thisValue;
      //since we made a switch we'll set a flag
      //as we'll need to execute the loop again
      doItAgain = true;
    }
  }
  if(doItAgain) bubbleSort(list);
}
bubbleSort(list);
console.log(list);

```

Executing this code with Node we should see this a sorted list: [4, 8, 15, 16, 23, 42]

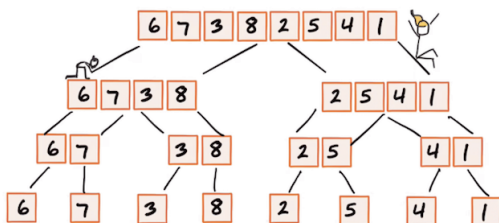
As you can see, we're using recursion as well as a for loop, which should set off some alarms. If you recall from the chapter on Big-O, nested loops operating on the same list almost always means $O(n^2)$ and this algorithm is no exception to that, even if we're using recursion.

The use of recursion also means we're potentially taking up $O(n)$ space as well and opens the possibility of a stack overflow exception given enough items to sort.

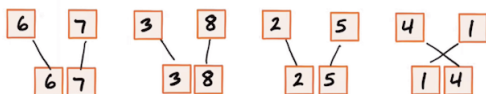
Merge Sort

Merge Sort is one of the most efficient ways you can sort a list of things and, typically, will perform better than most other sorting algorithms. In terms of complexity, we're using a divide and conquer approach, which should tip you off that this is going to be at least $O(\log n)$. Once we divide the array, we need to sort the items which is going to be an $O(n)$ operation since we need to address each item. That means this algorithm's complexity is $O(n \log n)$.

Merge Sort works by splitting all the elements in a list down to smaller, two-element lists which can then be sorted easily in one pass. The final step is to recursively merge these smaller lists back into a larger list, ordering as you go - this is the $O(\log n)$ part:



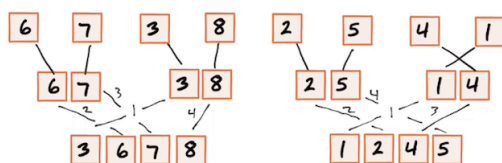
Now we need to merge the lists. The rules for this are simple: compare the first elements of adjacent lists, the lowest one starts the merged list – this is the $O(n)$ part:



This is straightforward with lists of one element being

combined into lists of two elements. But how do we match up lists of two elements?

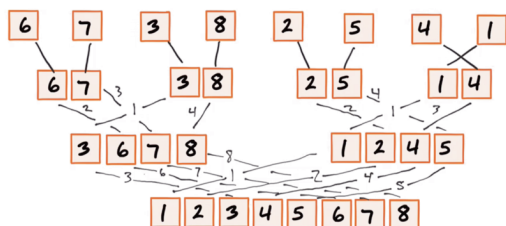
The same way. When combining the [6,7] list with the [3,8] list, we compare the 3 with the 6 – the 3 is the smallest so it goes first. Then we compare the 6 with the 8 and the 6 is smaller, so it goes next. Finally, we compare 7 and 8 and add them accordingly:



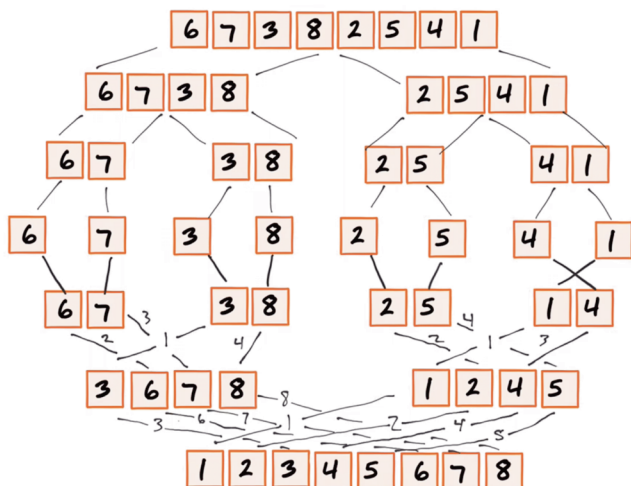
Now, you might be thinking “wait a minute – how do we know that a smaller number isn’t sitting to the right of the 6? Wouldn’t that mess up the sort?” That’s a good question.

It’s not possible to have a lower number to the right of any element in a merged list – when the [6,7] list was created we sorted it. This is the power of Merge Sort: the leftmost numbers are always smaller which gives us a lot of power.

OK, so now we continue in the same way, merging the final lists of 4. We start on the left-hand side of each list, comparing the values, and adding the lowest to the merged list first:



And we're done! Here's the full operation, in case you'd like to see it top to bottom:



Implementing merge sort in code is a bit tricky. You need to have two dedicated routines, one for splitting the list and one for merging.

The first step is to recursively split the list:

```

console.log(mergeSort(list));

const list = [23,4,42,15,16,8,3];
const mergeSort = (list) => {
  //if there's only one item in the list
  //return. This is our recursion check.
  if(list.length <= 1) return list;

  //cut list in half
  const middle = list.length/2;
  const left = list.slice(0,middle);
  const right = list.slice(middle,list.length);

  //recursively run through the splits
  //left and right will be separated down to single elements
  return merge(mergeSort(left), mergeSort(right));
};

```

In this routine we're just splitting whatever list comes in right down the middle. If the list only has one entry, we're returning. This prevents the recursive call on the last line from blowing up.

Next is our **merge** function:

```

const merge = (left, right) => {
  var result = [];
  //if the left and right lists both have elements
  //run a comparison
  while(left.length || right.length){
    //if there are items in both sides...
    if(left.length && right.length){
      //if the first item on left is
      //less than right...
      if(left[0] < right[0]){
        //take the first item on the left
        result.push(left.shift());
      }else{
        //take the first item on the right
        result.push(right.shift());
      }
    }else if(left.length){
      //just take left
      result.push(left.shift());
    }else{
      //just take right
      result.push(right.shift());
    }
  }
  return result;
}

console.log(mergeSort(list));

```

This routine takes two lists and compares their leftmost values. If one of the lists is empty then the left-most value from the other list is appended as the result.

Running this we get should see [3, 4, 8, 15, 16, 23, 42]

Quicksort

Quicksort is a divide and conquer algorithm that uses a pivoting technique to break the main list into smaller lists. These smaller lists use the pivoting technique until they are sorted. The complexity of quicksort, however, is not constant

because the pivot (as you're about to see) is determined at random and the partitioning of the list can put the algorithm at a disadvantage.

In the worst case, quicksort is $O(n^2)$ when the pivot is the smallest or largest element in the list. In the best case it's $O(n \log n)$, like merge sort. We'll discuss this a bit more at the end of this section; for now let's get to know this algorithm.

There are two ways to implement quick sort. Let's go over how the algorithm works and then I'll discuss the different implementations.

We'll start with a set of 8 elements (sorry, no cats or marbles this time).



We need a pivot, so we'll choose the very last element of the list (by convention).



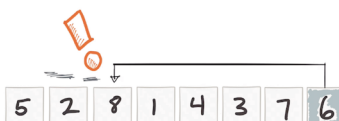
The next step is to partition our list so that all elements in the list that are less than our pivot are in a separate partition to the left, and all the elements greater are in a partition to the right. There are various ways to do this, but the simplest is to start at the beginning of the list – in this case a 5 – and if it's smaller we'll leave it in place.



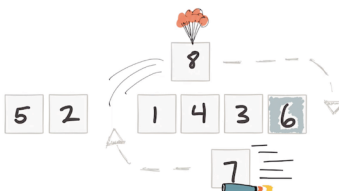
The next element is a 2, so we'll leave that there as well.



Then we come to an 8, which is greater than our pivot, which means we need to move it.



I'll pick it up and move it behind the pivot as it's a larger number, and I'll move the pivot down one position. There's already a number there – a 7 – so I'll move that to where the 8 was.



Now the 7 is the next thing to evaluate. It's greater than 6 so I'll do the same maneuver, switching the 3 and the 7, putting the 7 in the position my pivot was just in.



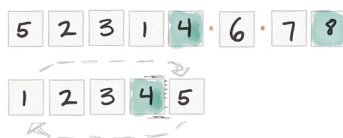
The next elements are 1 and 4 – which are both smaller than 6. This means we're done with our first partitions!



We now have two new lists – the numbers less than 6 on the left and the numbers greater than 6, on the right. Six itself in its final position, so we'll ignore it for now.

Now we pick a pivot for the new lists in exactly the same way – the last element in each list. This means that 8 is the pivot on the right, 4 is the pivot on the left.

The neat thing here is that there are no numbers greater than 8 in our list on the right, so there's nothing we need to do.



The list on the left, however, can be separated in exactly the same way we did before. We'll compare once again from the first position – it's a 5 so we'll stick it behind our 4, and move the number to the left of the 4 (a 1) to 5's old position.

And just like that – we're done!

If our list was presorted for whatever reason (and it turns out that yes, this does happen) then our sorting routine here would take a very, very long time. We'd have to split and order the list for every element, turning the complexity to $O(n^2)$.

To get around this we can select our pivot intelligently.

This requires an initial step where you find the median value of a list and then make that the pivot. From there the sorting operation will usually beat out merge sort because the sorting happens *in-place*, which means we have decreased *space complexity* and fewer overall operations.

Once again, we'll use recursion to help us split the list into partitions. The one tricky thing here is that we need to remove the pivot from the evaluation, which is commented inline:

```
const list = [23,4,42,8,16,15];
const quickSort = (list) => {
  //recursion check. If list is empty or of length 1, return
  if(list.length < 2) return list;
  //these are the partition lists we'll need to use
  var left=[], right=[];
  //default the pivot to the last item in the list
  const pivot = list.length -1;
  //set the pivot value
  const pivotValue = list[pivot];
  //remove the pivot from the list as we don't want to compare it
  list = list.slice(0,pivot).concat(list.slice(pivot + 1));
  //loop the list, comparing the partition values
  for (var item of list) {
    item < pivotValue ? left.push(item) : right.push(item);
  }
  //recursively move through left/right lists
  return quickSort(left).concat([pivotValue], quickSort(right));
};

console.log(quickSort(list));
```

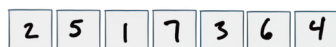
Selection Sort

Selection sort is likely the simplest possible way to sort a list. It's how you and I might think of telling a duck to do it if a duck had hands and needed to get a job as a coder at Amazon.

This algorithm works by scanning a list of items for the smallest element, and then swapping that element for the one in first position. This continues with the remaining items until the list is sorted.

The complexity of selection sort ranges from $O(1)$ when the list is already sorted to $O(n^2)$ when the list is presorted in the reverse order you want. The $O(1)$ complexity makes this an interesting choice if there's a chance of sorting a pre-sorted list (or a list of equal objects) – which happens fairly often.

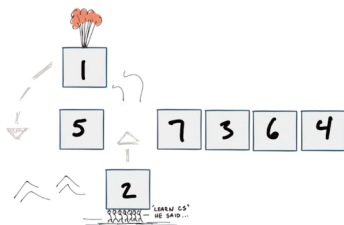
We'll start out with an unsorted list of 7 elements:



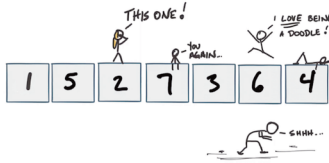
The first task is to find the lowest number, which (in the worst-case scenario) is a linear scan:



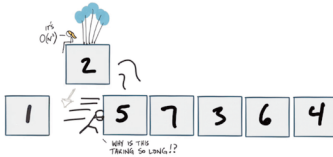
Once we've found the lowest element, we swap it with the first element in the list, as we know this is where the lowest element belongs.



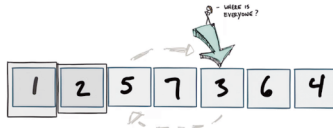
Then we do it all over with the remaining items. In this case the next lowest element is a 2.



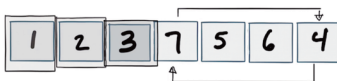
So, we swap the two with the element that was in second position.



Rinse, repeat. We keep scanning the list for the lowest item and swapping it for the items that aren't in their final position (shaded):



Not a very efficient operation. My doodles have rebelled, apparently – but our last operation is swapping the 4 and the 7 and then we're done!



This one is straightforward – no need for recursion. We can use two loops: the first will loop over our list, the second will loop forward for every step:

```
const list = [23,4,42,8,16,15];
const selectionSort = (list) => {
  for (var i = 0; i < list.length; i++) {
    //default the min value to the first item in the list
    //all we need do is track the index for now
    var currentMinIndex = i;
    //loop over the list, skipping the currentMinIndex
    for(var x = currentMinIndex + 1; x < list.length; x++){
      //if the current list item is less than the current min value...
      if(list[x] < list[currentMinIndex]){
        //reset the index
        currentMinIndex = x;
      }
    }
    //has the index changed?
    if(currentMinIndex !== i){
      //if yes, switch the values in the list
      var oldMinValue = list[i];
      list[i] = list[currentMinIndex];
      list[currentMinIndex] = oldMinValue;
    }
  }
  return list;
};
console.log(selectionSort(list));
```

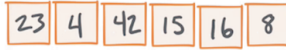
Heap Sort

Heap sort is a bit like selection sort in that it moves unsorted data to a sorted “partition” selectively. The difference, however, is that it uses a heap to do so. If you recall, a heap is a tree structure where parent nodes in one level are either greater than (max heap) or less than (min heap) child nodes in descendent levels.

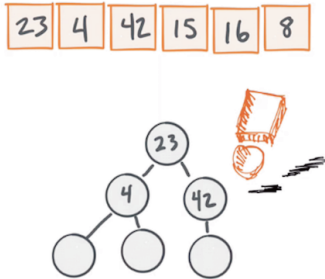
Heap sort is $O(n \log n)$, however it has an advantage over

quicksort of being $O(n \log n)$ in the worst-case scenario, whereas quicksort in the worst case is $O(n^2)$.

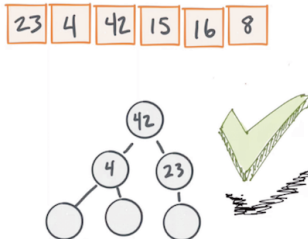
We have an unsorted list of numbers, as always.



The first step is to move these numbers into a heap:

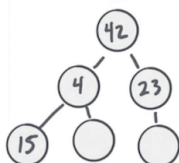


As we go along, however, we realize that we would violate the heap rules if 42 was to be placed before 23, so we swap them.



Now we have a valid heap.

23	4	42	15	16	8
----	---	----	----	----	---



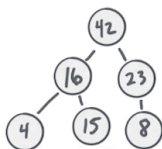
In the same way, 15 and 4 would be in violation – so we swap them.

23	4	42	15	16	8
----	---	----	----	----	---

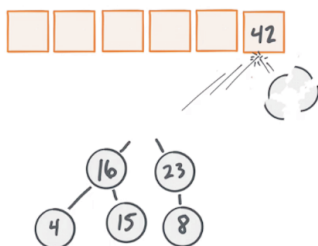


We end by adding 16 and 8, swapping positions for the 15 and 16 so we avoid violations. We now have a completed heap from our original list.

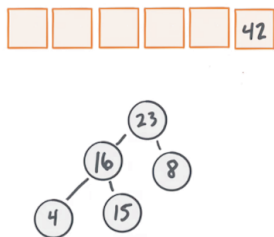
--	--	--	--	--	--



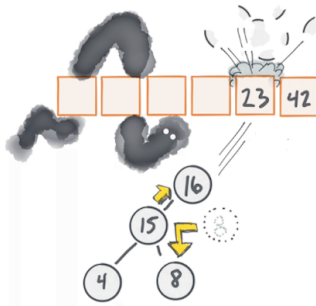
At this point we move on to the final stage, which is systematically removing the root of the heap, putting it back into the list.



As we add items back to our list, the heap is adjusted to make sure it remains valid. Here, we need to move 23 to the root as it's larger than its sibling, 16. 8 tags along for the ride as it's still in a valid position.



Next, we add 23 back to our list, and then elevate 16 to the root, as it's larger than 8. However, this means that 8 doesn't have a parent, so we move it over and place it under the 15.



We keep going in this way until all the nodes in our heap are gone.



Binary Search

Binary search is a fascinating thing. At first glance it seems rather ridiculous – the list you’re searching over has to be sorted first! This isn’t so nuts if you store your data in a binary tree (a BTREE) which we’ve discussed already.

Again, this algorithm is *divide and conquer* which should give a clue – I hope it does because we’ve already had a look at it in the Big-O chapter!

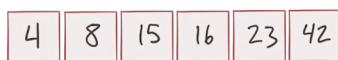
The deal is this: you split a list of sorted items and decide, from there, whether the item you’re looking for is in the left or right list. You can decide this accurately because the list is sorted.

You then split that list and check to see if values on the right or left of the split are greater, lesser, or equal to the value you're searching for.

Keep going until you find what you want.

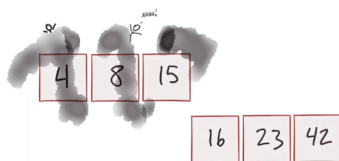
Let's take a look.

We have an ordered set to play with. How it became ordered is a mystery – as are the numbers – which we will figure out later in the book.

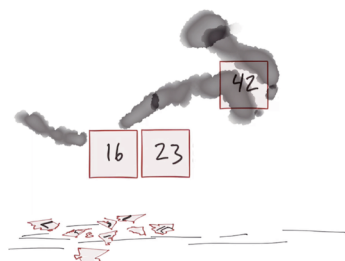


What we need to do is to find the number 23. Now, you and I both have eyes, but a computer doesn't – so we need to give it an intelligent way to find the number 23.

To do this, we'll split the list in two, right down the middle, and remove the entire half that is less than, or greater than, our number. In this case it's the side with 15 and below.



Since our list is sorted, we know the number we're looking for is in the remaining set. So, let's do the same thing. We'll split that list down the middle (or close to it).



The 42 is on the right – which means it and everything after it can be removed.



Now we're left with only two elements, which again we split down the middle. Evaluating each side, we see that the number we're looking for is on the right! We can remove the 16 and we're done!



Looking through a list by splitting it in half continually is straightforward. In this routine we'll search for a value and return its index:

```

const list = [4,8,15,16,23,42];

const binarySearch = (list, lookFor) => {
  //define the range
  var min=0, max=list.length;
  var middle;
  //while there is something to search for...
  while(min <= max){
    //define the middle of the range
    middle = Math.floor((min + max) / 2);
    //if we've landed on it...
    if(list[middle] === lookFor){
      return middle;
    }else{
      //if we haven't landed on it, where is it?
      //if the middle is less than the value we're
      //looking for, reset the min
      //otherwise reset the max
      list[middle] < lookFor ? min=middle : max=middle;
    }
  }
  return -1;
};

console.log(binarySearch(list,3));

```

Graph Traversal

We've learned how to split lists apart using binary search, but how do we search over something a bit more complex, such as a graph? As you've noticed, many of the data structures we've been working with are based on graphs, so traversing them properly is quite important.

Before we go on, it's important to understand that you can carry out these operations in one of two ways: using *recursion* or using *iteration*. One is clever and one is costly – thankfully I'm referring to the same method: *recursion*.

Every time a recursive function is executed, the variables that the function uses remain in scope and are stored on the stack. I know I've brought this up before, but it's worth repeating: *recursion is interesting and clever, but can also be costly.*

I point this out because *you will be asked something like this in an interview someday.* Recursion and *space complexity* are not friends – so come prepared to understand why. Let's go a bit deeper.

We've discussed binary trees in a previous chapter:

```
class BinaryTreeNode{
  constructor(val){
    this.value = val;
    this.right = null;
    this.left = null;
  }
  isLeaf(){
    return this.left === null && this.right === null;
  }
  traverse(node){
    if(node.left) return this.traverse(node.left);
    if(node.right) return this.traverse(node.right);
    //otherwise we're a leaf
    return null;
  }
}
```

A node instance has a value as well as two possible descendants, **left** and **right**. The **traverse** method allows you to start from any node in the tree and then traverse through each node, doing something exciting and amazing.

There are several ways to run calculations with this class.

We can hard-code what we need done in the **traverse** method (searching for a value, for instance) or we can pass along some kind of callback. For the sake of keeping things at a pace for this book, I'm going to sidestep implementation details here and simply wave my arms and say "don't worry about this for now".

The reason is that this code, while clever, is also a bit inefficient. I already discussed the stack overflow problems above, but what if you wanted to control what *type* of search you wanted to perform? In other words, what if you wanted to *go deep* into the tree as you knew the thing you're looking for was likely at the lower levels of the tree?

Conversely: what if you wanted to go *wide* instead? Recursion can be tweaked to address these things, but then you're altering your class to accommodate what should really be the responsibility of an entirely different bit of code altogether.

Instead of filling up our stack, what if we used a simple loop? Let's strip out the **traverse** method and go with a more cohesive **BinaryNode** class:


```

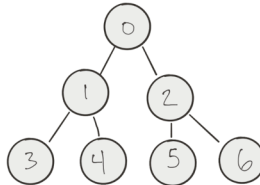
class BinaryNode {
    constructor(val){
        this.value = val;
        this.right = null;
        this.left = null;
    }
    isLeaf(){
        return this.left === null && this.right === null;
    }
}

//Now let's fill out our nodes:
const rootNode = new BinaryNode(0);
rootNode.left = new BinaryNode(1);
rootNode.right = new BinaryNode(2);
rootNode.left.left = new BinaryNode(3);
rootNode.left.right = new BinaryNode(4);
rootNode.right.left = new BinaryNode(5);
rootNode.right.right = new BinaryNode(6);

```

This will give us a two-level tree, which will be good enough to traverse. But how? It turns out there are two ways: *Breadth-first* and *Depth-first*. These types of traversals are typically referred to as “breadth-first search” and “depth-first search”.

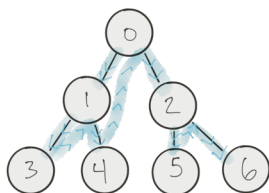
Let's start by visualizing our graph:



With a depth-first search, we want to go as deep as possible into the graph from either the left or the right. Traditionally it's done from the left.

The idea is to go as deep as you can and then backtrack your way back up and then over to the right until the traversal of the tree is done.

With my crude drawing skills, it might look something like this:



We start at the root, proceed to 1 and then 3. Then backtrack to 4, back up and over to 2, then 5 then 6.

To implement this we need a data structure that will keep track of the very next descendant as well as the current sibling. We will always follow the next descendant, and if it doesn't exist, we'll go with the last saved sibling. If both of those don't exist we're done with our tree.

For this, we can use a stack. Here's some basic code to see how DFS works. Note that I'm wrapping this in a class using a static method for **traverse**:

```

class DFS {
  static traverse(root){
    var stack = []; //arrays in JS act like stacks
    let thisNode = null;
    //push the root onto the stack
    stack.push(root);
    while(stack.length > 0){
      thisNode = stack.pop();
      console.log(thisNode.value);
      //push right then left
      if(thisNode.right !== null){
        stack.push(thisNode.right);
      }
      if(thisNode.left !== null){
        stack.push(thisNode.left);
      }
    }
  }
}

```

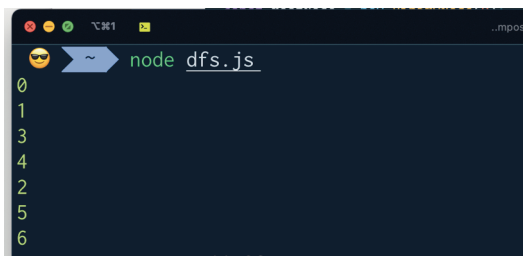
I'm cheating a small bit here by using an array in JavaScript as my stack, but it will work just fine because JavaScript arrays implement the **push/pop** methods in the same way a traditional stack does.

All we need to do is to keep track of how many nodes there are left to traverse. We know that by using a stack and pushing values onto it if they exist down below.

Let's load it up and run it!

```
const rootNode = new BinaryNode(0);
rootNode.left = new BinaryNode(1);
rootNode.right = new BinaryNode(2);
rootNode.left.left = new BinaryNode(3);
rootNode.left.right = new BinaryNode(4);
rootNode.right.left = new BinaryNode(5);
rootNode.right.right = new BinaryNode(6);
DFS.traverse(rootNode);
```

The result, when we execute this, is right on the money:



```
node dfs.js
0
1
3
4
2
5
6
```

“Going deep” on a graph is useful if you’re interested in parent/child relationships.

You could also look through a binary tree and see if it is in fact a *binary search* tree, with all the node values set as required. We know from reading the last chapter that a binary search tree requires that every child node to the right of a given node must have a greater value; every child node to the left must have a lesser one.

Is depth-first the best choice for this? You might check a lot more nodes than you need to! Let’s have a look at an alternative: *breadth-first*.

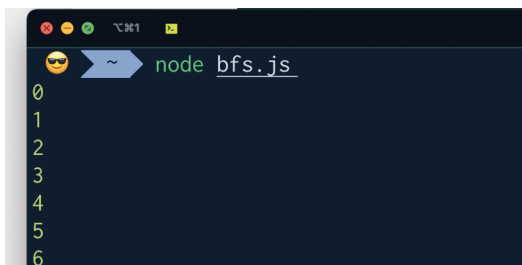
With breadth-first search we need to track the nodes of a tree in each level before traversing to the next level. This means we need to track every node and its children *in order*.

For this we need to use a queue, which in JavaScript once

again means using an Array. Unfortunately, there are no **enqueue** and **dequeue** methods, but we do have their equivalent: **push** and **shift**.

```
class BFS {
  static traverse(root){
    const queue = []; //arrays in JS also act like queues!
    let thisNode = null;
    //push the root onto the queue
    queue.push(root); //equivalent to "enqueue"
    while(queue.length > 0){
      //remove the next item from the queue
      thisNode = queue.shift(); //equivalent to dequeue
      //do whatever calc
      console.log(thisNode.value);
      //add the children into the queue
      if(thisNode.left != null){
        queue.push(thisNode.left);
      }
      if(thisNode.right != null){
        queue.push(thisNode.right);
      }
    }
  }
}
```

If we load it up just like we did in the last section and run it, we will have a completely different result:



We're traversing the graph one level at a time, which

makes great sense if we're trying to evaluate all the parent nodes *first*, which we would be doing if we're validating a binary search tree.

ADVANCED ALGORITHMS

DYNAMIC PROGRAMMING, BELLMAN-FORD AND
DIJKSTRA

I wasn't sure about using the terms "advanced" and "simple" for these chapters on algorithms. I don't want you to think these are harder to implement or to figure out – they're not (necessarily).

The reason I chose these terms is that it's more likely you'll find yourself getting paid to implement a shortest path algorithm or some type of sieve at some point in your career. It's *unlikely* that you'll be hired to implement bubble sort.

I also wanted to provide a framework for you to create your algorithms for solving the more interesting problems you work on every day. This, to me, is the most important part of this chapter, so we'll start there.

The Code

You can find the code used in parts of this section up at my GitHub repo. In addition, you can buy over 10 hours of video walkthroughs of the algorithms you see in this chapter and others from here. I'll be using screenshots once again for the

code samples for formatting reasons – if you want to play along please do... but you'll need the code from GitHub.

Dynamic Programming

No, this section is not about Ruby, Python, JavaScript, etc. Dynamic programming is a way to solve a problem using an algorithm in a prescribed way. It sounds complicated, but it's anything but.

Dynamic programming gives us a way to elegantly create algorithms for various problems and can greatly improve the way you solve problems in your daily work. It can also help you ace an interview.

Let's start with a quick definition so we know what dynamic programming is and how it works. At its core, dynamic programming is simply solving an optimization problem by guessing systematically. It's almost laughable to think about dynamic programming in terms of this definition, but as you'll see it turns out to be rather powerful.

To use dynamic programming, the problem you're solving must be:

- An **optimization problem**. We saw one of these in Chapter 1 (the Bin Packing Problem) when I tried to optimize storage for my daughter's things.
- **Dividable into subproblems**. With dynamic programming you can recurse over and solve to solve the larger, objective problem.
- Have an **optimal substructure**. That's a mouthful, but what it means is that the subproblems you solve must be complete unto themselves. In other

words, if you solve subproblems x , y and z to solve objective problem A , then the solutions to x , y and z should be sufficient on their own to solve A . You don't need to use x plus some other algorithm.

- **Reducible to P time** through memoization. Some problems you can solve with dynamic programming are solvable in exponential time (like Fibonacci), however this can be reduced to P time through memoization. Another fun word, but you can think of this basically as “caching” the answers to the subproblems and then applying.

I wouldn't blame you if you're underwhelmed at this point. The name “dynamic programming” seems bland, and the underlying techniques more than a little vague. You'll understand it well in a few sections as we solve some problems with it, I promise.

Before we get there, it's important to understand where the name came from and why dynamic programming even exists.

This is a funny story (emphasis mine):

An interesting question is, ‘Where did the name, dynamic programming, come from?’ The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was

employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, ‘programming.’ I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let’s kill two birds with one stone. Let’s take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it’s impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It’s impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.

If you want to read Richard Bellman’s original paper on dynamic programming, *you can do so here*.

There you have it: **the name means nothing**. The dynamic programming design process, however, is behind some of the most powerful algorithms we know of. We’ll see those in the next section.

The best way to see its power, however, is to just do it. So let’s! We’ll use dynamic programming to help us get through a job interview.

Again with Fibonacci! Ah well it is a programming book

you know. I'm using it here because it's the simplest way to convey the dynamic programming process. Also: you will be asked how to solve Fibonacci at some point in your career, and you're about to get three different approaches!

Which leads right to a great opening point: *our jobs are about solving problems*. When you go to these interviews, they mostly want to see how you would go about solving something complex. As it turns out, the *Interviewing For Dummies* book says that Fibonacci is a great question for just that case.

Let's start with a definition, just in case you don't know or remember what a Fibonacci Sequence is:

A series of numbers in which each number (Fibonacci number) is the sum of the two preceding numbers. The simplest is the series 1, 1, 2, 3, 5, 8, etc.

Super. Why do we care about these numbers? These numbers (and the algorithm we're about to discuss) underpin nature's symmetry:

The Fibonacci numbers are Nature's numbering system. They appear everywhere in Nature, from the leaf arrangement in plants, to the pattern of the florets of a flower, the bracts of a pinecone, or the scales of a pineapple. The Fibonacci numbers are therefore applicable to the growth of every living thing, including a single cell, a grain of wheat, a hive of bees, and even all of mankind.

If you divide each successive number by itself (so: $5/3$, $8/5$...) you converge on a fascinating number called phi:

What makes a single number so interesting that ancient Greeks, Renaissance artists, a 17th century astronomer and a 21st century novelist all would write about it? It's a number that goes by many names. This "golden" number, 1.61803399, represented by the Greek letter Phi, is known as the Golden Ratio, Golden Number, Golden Proportion, Golden Mean, Golden Section, Divine Proportion and Divine Section. It was written about by Euclid in "Elements" around 300 B.C., by Luca Pacioli, a contemporary of Leonardo da Vinci, in "De Divina Proportione" in 1509, by Johannes Kepler around 1600 and by Dan Brown in 2003 in his best-selling novel, "The Da Vinci Code."

Absolutely fascinating stuff. Our interviewer, however, is waiting patiently for us to come up with an algorithm for calculating a Fibonacci Sequence to the n th position – so let's get to it!

The interviewer has asked us a standard question:

How would you derive a Fibonacci sequence up to a given position?

In other words, if we're given a value of 10, the interviewer will want to see the first 10 Fibonacci numbers. We can solve this (and more!) using dynamic programming.

The first step is to break the problem down into smaller problems (called subproblems) that we can solve. If we're trying to derive a Fibonacci sequence to the 10th position, we can do it with pen and paper like this:

- The first number in the Fibonacci sequence is 0
- The second number is 1
- The third number is $0+1=1$
- The fourth number is $1+1=2$

And so on. This would answer the interviewer's question (about the sequence) but it wouldn't show them what they're after: our ability to solve a problem programmatically. We can do this with the next step in dynamic programming: recursively solve the subproblems until the objective problem is solved.

It's easiest if we see some code at this point. Here's my Fibonacci solver implemented in JavaScript:

```
//the slow way  
let fibCount=1;  
const calculateFibAt = (n) => {  
  fibCount = fibCount+1;  
  if(n < 2){  
    return n;  
  }else{  
    return calculateFibAt(n-2) + calculateFibAt(n-1);  
  }  
}
```

Running this (using Node):

```

..os/algorithms (-zsh)
~/Demos/algorithms master ± node fib.js
0
1
1
2
3
5
8
13
21
34
55

```

Great! By the way I tried four times to write this from memory and *completely failed*. You would think this little routine would be embedded in my mind but ... oh well. If it took you a few times to come up with it don't feel bad! Recursive programming takes some getting used to.

The code in this routine works, is straightforward, and is standard interview fare. We're feeling happy about ourselves at this point, when the interviewer says:

Talk to me about the complexity of this routine in terms of time and also space...

Uh-oh... time for some Big-O! Good news for us is the that we started this part of the book off with a discussion of Big-O and our brains are burning now, thinking "right... wasn't there something bad about using recursion?"

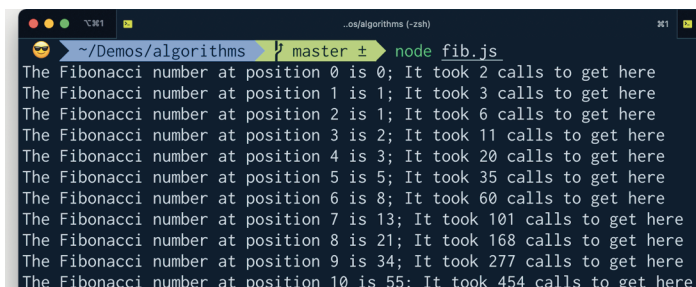
Let's start off by thinking in terms of *time complexity*. How long will it take to run our recursive algorithm? You can see why you might get fired by changing the loop value to 1000. In short: this routine scales horribly. To see this, let's add a counter to the function:

```

for(var i = 0; i<10; i++){
  let fibbed = calculateFibAt(i);
  console.log(`The Fibonacci number at position ${i} is ${fibbed}; It took ${fibCount} calls to get here`);
}

```

When we run this code we should see how many recursive calls have been made. Let's see:



```

~/Demos/algorithms master ± node fib.js
The Fibonacci number at position 0 is 0; It took 2 calls to get here
The Fibonacci number at position 1 is 1; It took 3 calls to get here
The Fibonacci number at position 2 is 1; It took 6 calls to get here
The Fibonacci number at position 3 is 2; It took 11 calls to get here
The Fibonacci number at position 4 is 3; It took 20 calls to get here
The Fibonacci number at position 5 is 5; It took 35 calls to get here
The Fibonacci number at position 6 is 8; It took 60 calls to get here
The Fibonacci number at position 7 is 13; It took 101 calls to get here
The Fibonacci number at position 8 is 21; It took 168 calls to get here
The Fibonacci number at position 9 is 34; It took 277 calls to get here
The Fibonacci number at position 10 is 55; It took 454 calls to get here

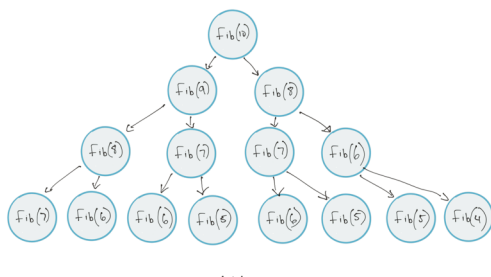
```

453 calls! Good grief! As you can see, the number of calls to our routine goes up more than exponentially with each additional input:

- Input 0 resulted in 1 call
- Input 1 resulted in 2
- Input 2 resulted in 5 calls
- Input 6 resulted in 59
- Input 10 resulted in 453

If you really want to have some fun, let it run for 15 minutes and see how many calls it takes to calculate `calculateFibAt(32)`... it's 18,454,894!!!

Another way to think about this is from the top down. Here we can visualize the complexity for calculating the Fibonacci number in 10th position using a graph:



This is horribly inefficient. Look how many times **fib(6)** and **fib(7)** are called! The interviewer seems happy with our answer and then asks us:

Tell me about the space complexity...

Right. We learned from the last section that a recursive routine will push values onto the stack repeatedly, once for every single call of the current function. If you do this enough, you'll run into a stack overflow exception, which is bad.

As mentioned before: *recursion and space complexity aren't friends*.

Sounds good. So how would you improve this routine?

This is where we get to the next step of dynamic programming. We can reduce the time and space complexity of our algorithm by using memoization. We can do this because the solution to each subproblem is optimal, meaning that it can stand alone, and we don't need anything else to use its value.

In Big-O terms, we can use the memoized solution in linear time, $O(n)$ where n is the position we're interested in, which will speed things up tremendously. What about *space complexity*? Can you figure out a way to do this in constant space? I'll talk about that in the next section.

Memoization is simply caching. In more formal terms it's remembering the solution to a subproblem, so you don't have to calculate it again recursively. This only works if the subproblems are in an optimized substructure. You can think of that as a large graph (like the one above), where you can simply replace `fib(6)` with the number 8. That's darn optimal if you ask me.

To accomplish this, I'll store the results of our loop in some kind of data structure; the question is *which one*? We've learned about a whole mess of them in a previous chapter... which would be the best?

All we need to do is to remember some values in memory and then to iterate over them. If this is all you need, *don't overthink it!*

Since we know that Fibonacci numbers start with 0 and 1, I can use those seeds to calculate the remaining numbers:

```
//the fast way
const calculateFibFaster = (n) =>{
  var memoTable = [0,1];
  for(let i=2; i <= n; i++){
    memoTable.push(memoTable[i-2] + memoTable[i-1]);
  }
  return memoTable;
};

//run it the fast way
console.log(calculateFibFaster(10));
```

The result, when run:

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

Perfect. But what about *time complexity*? This is simple to reason through, but before we do let's run this faster Fibonacci routine 1000 times. You should see that it returns almost instantly.

We want to find the Fibonacci numbers up to a given number n , which means we'll need to perform some operation for each n . If you recall from a few chapters back, iterating over a collection is always $O(n)$. Accessing a value from an array using its index is always $O(1)$, so our total operation here is $O(n * 1)$ which is $O(n)$.

But what about the *space complexity*? We're not using recursion so that means we won't be potentially overloading the stack, which is a good thing. We have a few variables and an array for every n number that we're evaluating, so our space complexity is also $O(n)$.

You might be thinking "hey wait a minute you have a loop variable in there too!" and yes, that's true, but with Big-O you're more concerned about the *nature* of the algorithm. In this case it's simply $O(n)$.

Can we do better here? *Yes*. Well... sort of. Now, we're returning an array of Fibonacci numbers up to the *nth* number. We could ask our interviewer at this point if they're interested in the whole sequence or just the *nth* number?

Just the nth number will do.

Perfect. This means we can now use a *greedy algorithm*, which is a term you should remember for interviews. Let's take a small diversion (again).

Greedy Algorithms

A greedy algorithm does what's best at that moment. Put in math terms:

A greedy algorithm is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

Say what?

Let's do this with code, then I'll see how well I can explain the idea. We'll start by redoing our `fibFaster` function:

```
const fibConstantSpace = function(n){
  let twoFibsAgo = 0, oneFibAgo=1, currentFib=0;
  //make sure to output the first two
  for(let i=2; i<=n; i++){
    currentFib = twoFibsAgo+oneFibAgo;
    twoFibsAgo = oneFibAgo;
    oneFibAgo = currentFib;
  }
  return currentFib;
}

console.log(fibConstantSpace(10));
```

Running this code we end up with the last Fibonacci number in our sequence, which is 55.

The best part, however: *no more arrays required!* What we're doing here is simply “remembering” *only what we need to remember*, essentially “locally optimizing” our decision making by setting these values to variables and forgetting the rest.

That's a **greedy algorithm** in practice. Our interviewer

likes this, but has a question:

Can you think of a way to make this a bit more flexible for calling code? Right now you're just returning the *nth* fibonacci number; what if I wanted to do something else in my calling code?

If this happens to you in an interview, take it as a good sign. You've nailed the question so far! They're happy with your response and are likely just wanting to see how much better you are than the question allows.

We're using JavaScript, so ideally the answer is jumping out at you. Most languages support the idea of *callbacks*, something that will yield control of the current iteration/operation. We can use that here to *yield* the **currentFib** value back to the calling code:

```

var fibConstantSpace = function(n, fn){
  let twoFibsAgo = 0, oneFibAgo=1, currentFib=0;
  //make sure to output the first two
  if(fn){
    fn(0);
    fn(1);
  }
  for(var i=2; i<=n; i++){
    currentFib = twoFibsAgo+oneFibAgo;
    twoFibsAgo = oneFibAgo;
    oneFibAgo = currentFib;
    if(fn) fn(currentFib);
  }
}

fibConstantSpace(10, console.log)

```

There we go! The best of both worlds: we can report back each number so the calling code can do whatever it wants, or we can just return the final value. Running this code you should see the same list of Fibonacci numbers that we've been working with.

Greedy algorithms can be useful when solving some very complex problems. A few chapters ago we examined a few very tough, NP-Hard optimization problems, one of which was The Traveling Salesman Problem.

One way to approximately solve this problem is using a heuristic (that's a fancy word for "rule of thumb") called Nearest Neighbor:

The nearest neighbour algorithm was one of the first algorithms used to determine a solution to the traveling

salesman problem. In it, the salesman starts at a random city and repeatedly visits the nearest city until all have been visited. It quickly yields a short tour, but usually not the optimal one.

In other words: *there's no master plan here*. Nearest Neighbor just looks at the next cheapest city and goes there. This is a classic *greedy algorithm*.

Another greedy solution is finding your way out of a maze. You just put your right hand on the nearest wall and keep walking until you're out. Not the *optimal* solution, but it *will* solve the problem.

For a final example: consider Agile Development. Teams gather quickly in the morning so everyone's aware of what's going on with everyone else. Adjustments are welcomed and deployment rapid - it's all about quick adaptation to the changes in the development process.

Now, if you were to step back and look at a software project as a series of decisions which you could represent on a graph (which you can), then Agile is, itself, a greedy algorithm! It might not be the *optimal* solution to success for a project, but it is a solution! You're simply doing the next, closest thing that makes the most sense to the team and client without a master (waterfall) plan in place.

Bellman-Ford

Right then, done with Fibonacci thank goodness. On to graph traversal! One of the most fascinating uses of graphs is in the optimization of path traversal, which can be used in a vast number of calculations.

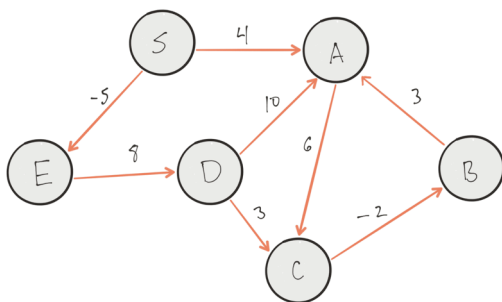
As mentioned in the previous chapter, graphs can be used to represent all kinds of information:

- A network of any kind. Social (friends) or digital (computers or the internet), for example
- A decision tree
- Contributions from members of any kind to a cause of any kind
- Atomic interactions in physics, chemistry or biology
- Navigation between various endpoints
- If you apply weighting to the edges or vertices, you can run useful calculations for just about anything. One of the most common is finding the shortest path between two vertices.

There are numerous algorithms to touch on at this point, but I must round this chapter out by discussing the two you should be aware of: Bellman-Ford and Dijkstra. In this section we'll discuss Bellman-Ford; Dijkstra comes next.

This algorithm is named after Richard Bellman (the same person who wrote about dynamic programming) and Lester Ford Jr. It shares a lot with Dijkstra's algorithm (which we'll see next), but has one major advantage: it can accommodate negative edges.

Let's see how it works. Consider this graph:



This is an edge-weighted, directed graph. We want to calculate the shortest paths between our source vertex S and the rest of the vertices, A through E. We can do this using dynamic programming.

Now, if you were to stare at this and I told you what your task was (to calculate the smallest cost between S and the rest of the vertices), you would probably get overwhelmed! I know I did.

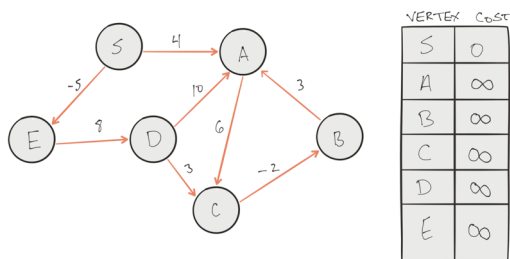
The good news is that we have the Bellman-Ford algorithm, which makes this calculation rather fun. Rather than spend many words on it, let's do it visually, then implement it with some code.

We will be using memoization to keep our calculations in P time, so let's setup a memoization table that we can update as we run our calculations. We know that there are only 6 vertices, so we only need to track, at most, 6 total costs. The only cost we know is the base cost: $\delta(S, S) = 0$

Note: if you're not familiar with that squiggle, δ , it's a "delta", which indicates a difference. This equation states that the difference (or "cost") from S to S is 0.

For the rest of the vertices, we don't know so we'll set them all to infinity. Why infinity? Simply because the calculation we do later will be based on finding the smaller value, and infinity guarantees that the initial state will not remain.

Here's our setup:



Great. The plan is to calculate the cost of each outgoing edge, for each vertex in our graph. This will be 6 total calculations which we'll repeat in iterations. The number of iterations i you need to perform when using Bellman-Ford is: $i = |V| - 1$.

Why is this? You'll see in a second, but the crux of it is that we're calculating the distances between all nodes and remembering the smallest ones. This is called relaxation: we start with the largest values we can think of (infinity) and then slowly relax the costs through an iterative calculation.

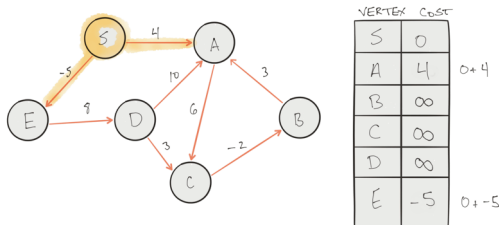
OK, enough words, let's do this.

We'll start with our source, S , and move clockwise around the graph. From S we have two outgoing edges with the values 4 and -5. These costs (C_{sa} and C_{se}) associate S with A and E so we'll add them to our table using this calculation:

$$C_{sa} = \delta(S, S) + \delta(S, A)$$

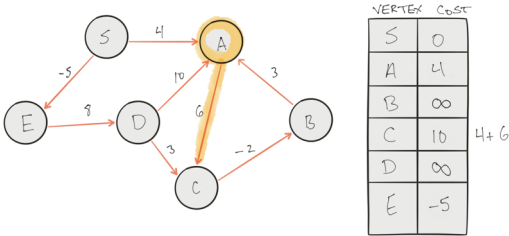
$$C_{se} = \delta(S, S) + \delta(S, E)$$

You can visualize this on the graph itself. We're using the cost of the highlighted edges below and adding them to the initial cost of S to S (which is 0):



If this seems complicated, let it wash over you. As we move through this it will make more sense.

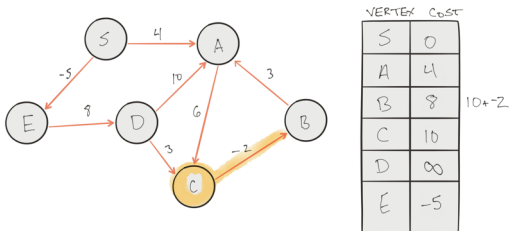
OK, we're still in our first iteration of calculating the costs between S and the rest of the vertices. Let's now move to A and calculate the outgoing edges from A:



A only has a single outgoing edge with a cost of 6, so we take that cost and add it to A's current cost (which is 4). This gives us a value of 10, which we add to our memo table.

Now we move on to B, which has a current cost of infinity. This means that we don't have a path from S to B yet, which means we can't calculate it in this iteration. So we skip it.

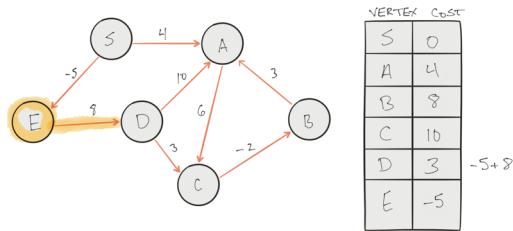
Next up is C, which does have a current cost of 10. It also has an outgoing edge to B which is good news as we'll be able to use that in the next iteration. For now, let's update our table:



The current cost of the path to C is 10, so adding -2 to that (which is the cost of the edge between C and B) gives B a current cost of 8.

Now we're up to D. What do you think we do here? The current cost is set to infinity, which means we haven't calculated a path to it yet, so we skip it.

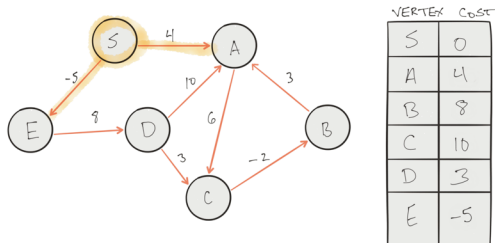
Finally, we close off this iteration by calculating the outgoing edges from E:



By now you should be able to reason what will happen. We know that E has a current cost of -5, so we add that to the cost of the edge between E and D, which is 8. This gives a cost from S to D equal to 3.

Great! We now have costs for all the vertices! It's time to do another iteration to see if we can relax these values a bit more.

Let's step through this a bit quicker this time. We'll start with S again, our source, and note that the cost of the outgoing edges does not improve on the costs we've recorded (4 and -5). This makes sense as these are single edges and there really is no way to improve the costs here:



If we move to A, again the only outgoing edge we can evaluate is A to C, which is 6. The cost from S to C remains the same at 10, so there's no improvement here and we can move on.

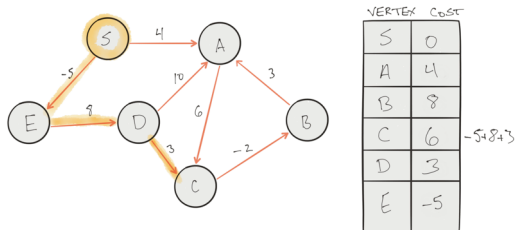
Next up is B, which now has a cost associated with it so we can use it in our calculations. The current cost to get to B is 8, so getting from B to A is 11 total, which is not an improvement over A's current cost of 4.

Then we come to C, which is 10. We've already calculated C to B as 8 and there's no improvement here, so we can move on again.

Seems a bit boring, doesn't it? We keep skipping things – but that's OK! It's about to get a bit more exciting when we consider D, our next vertex.

The current value of D is 3 and has two outgoing edges to A and C. The value of the edge from D to A is 10, so adding the current cost of D (3) to this edge cost of 10 would be 13. This doesn't reduce A's cost so we leave it. But what about edge D to C?

This produces a cost of 6, which means we can lower C's cost in our memo table to 6:

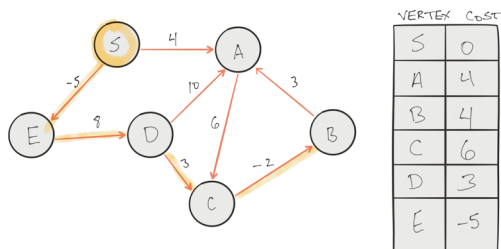


The excitement! Can you feel it! Now, as you might imagine, changing C like this means we'll be able to change more values in our table – but that will have to wait for the next iteration.

Speaking of, we're back to E now and there are no changes we can make here, so we'll skip on to the next iteration.

We skipped S and A last time and that's still the case this time: there are no changes we can make to improve our current costs for these vertices. Same for boring old B, again. We can't improve the cost of A, which is its only outgoing edge.

C, however, makes things interesting. The current cost of C is now 6, which means we can reduce the cost of B to 4:



Oh my that's so exciting isn't it! From this point we go to D and E just like before, skipping them as there's no way to improve their costs.

We can also use some more reasoning here. Since B was the only change we made from the last iteration, we can just evaluate its outgoing edge to see if it will improve A. Since B's current cost is 4 and A's current cost is 4, then no improvement will happen if we add the cost of the edge from B to A, which is 3.

So we're done! The shortest paths for each vertex are now calculated.

I'm sure you'll find some ways to improve this code, which is great! I've made it a little more verbose for clarity, which I think will help if you're still having problems getting your head around this algorithm.

```
//define the vertices - these can just be string values
var vertices = ["S", "A", "B", "C", "D", "E"];

//our memoization table, which I'll set to an object
//defaulting as described
var memo = {
  S : 0,
  A : Number.POSITIVE_INFINITY,
  B : Number.POSITIVE_INFINITY,
  C : Number.POSITIVE_INFINITY,
  D : Number.POSITIVE_INFINITY,
  E : Number.POSITIVE_INFINITY
}
```

As you can see we have a simple array and a JavaScript object to serve as our memo table.

Now we need to define the graph itself. For this I simply

need to track which vertices are involved and the costs associated with the relationship:

```
//this is our graph, relationships between vertices  
//with costs associated  
var graph = [  
  {from: "S", to: "A", cost: 4},  
  {from: "S", to: "E", cost: 6},  
  {from: "A", to: "C", cost: 6},  
  {from: "B", to: "A", cost: 3},  
  {from: "C", to: "B", cost: 2},  
  {from: "D", to: "C", cost: 3},  
  {from: "D", to: "A", cost: 10},  
  {from: "E", to: "D", cost: 8}  
];
```

Looking good! At this point you should be able to reason how we'll use these data structures to iterate over our graph. In short, we need to:

- Iterate over the vertices array
- Using the current vertex from our vertices array, we select the outgoing edges from the graph array.
- Once we have the outgoing edges, we run a quick calculation using our memo object. The calculation is straightforward: we take the cost of the current vertex and add it to the cost of the current outgoing edge. If that value is less than the cost of the current edge, we update the memo for the current edge.

Translating that word salad to code:

```

//represents a full iteration of Bellman-Ford on our graph
const iterate = () => {
  //do we need another iteration?
  //decided below
  let doItAgain = false;
  //loop all vertices
  for(fromVertex of vertices){
    //get the outgoing edges
    const edges = graph.filter(path => {
      return path.from === fromVertex;
    });
    //loop the outgoing edges
    for(edge of edges){
      const potentialCost = memo[edge.from] + edge.cost;
      //reset the cost as needed
      if(potentialCost < memo[edge.to]){
        memo[edge.to] = potentialCost;
        //if the cost was changed we need to loop again
        doItAgain = true;
      }
    }
  }
  //return the flag
  return doItAgain;
}

```

Great! Now all we need to do is to iterate over our iterate function and we're good to go:

```

for(vertex of vertices){
  //loop until no changes
  if(!iterate()) break;
}
console.log(memo);

```

You'll notice that I'm only iterating to `vertices.length - 1`.

Can you reason why? If you run this code (using Node), you should see:

```
{ S: 0, A: 4, B: 4, C: 6, D: 3, E: -5 }
```

These are the exact values of our exercise above. Nicely done!

This is dynamic programming in action. Dividing the objective problem (finding the shortest paths) into smaller problems (calculating the costs between each vertex). We then recursed over the smaller problems (the **iterate** function) and used memoization to derive the answer.

There is room for improvement, however. Think about the process we went through in the first section. I only needed 3 total iterations to derive the shortest paths. The code I wrote, however, required 5. How would you optimize this?

In addition, I'm not using recursion in a code sense. I am calling the same function repeatedly, but there's probably a tighter, cleaner way to do this using a true recursive function. Can you see how?

Finally, here's something to ponder: does the order in which we evaluate the vertices matter? If yes, why? If no ... why not? Rearrange the code a bit and see if you come up with a different answer than you see here.

The Bellman-Ford algorithm is quite effective, as we can see, but it can also take a long time to run. In terms of complexity, the algorithm runs in $O(V * E)$ time, where V is the number of vertices and E is the number of total edges. It can work well for simple graphs, but for more complex (or dense) graphs, it is not the most efficient algorithm.

Dijkstra

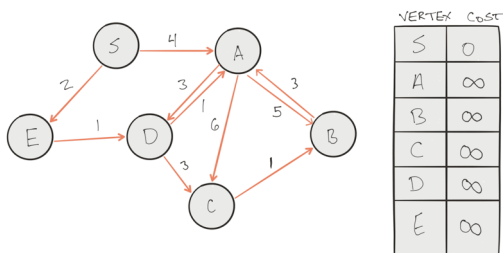
In the last section we iterated over a simple graph using Bellman-Ford to find the shortest paths from a single vertex (our

source) to all other vertices in the graph.

The complexity of Bellman-Ford is $O(|V|E)$, which can approximate $O(n^2)$ if every vertex has at least one outgoing edge. In other words: *it's not terribly efficient*.

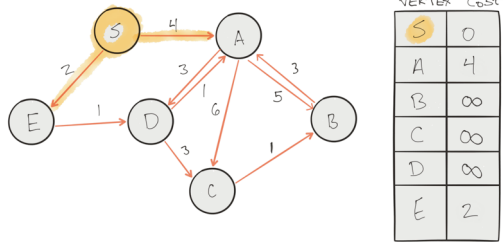
Dijkstra's algorithm requires only one iteration, however and has a complexity of $O(|V| \log V)$, which is much more efficient. Let's see why.

As with Bellman-Ford, we'll use a directed, weighted graph with 6 vertices. In addition, we'll setup a memo table with our source S set to 0 and the rest of the vertices set to infinity:



There is a difference here, however, and it's critical! Dijkstra doesn't work with negative edge weights! I have adjusted this graph so that we don't have any negative weights, as you can see. Specifically, the edges between S and E as well as C to B . In addition, I've added a few edges to show that the algorithm will scale easily regardless of the number of edges involved.

The first step is to evaluate the source, S . We do the same thing as before, with Bellman-Ford, where we tally up the cost of the outgoing edges and store them in the memo table. In addition, we'll mark S as complete:

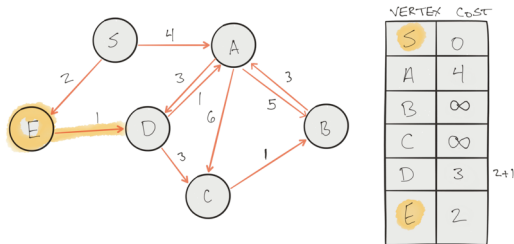


This is another way that Dijkstra differs from Bellman-Ford: each vertex is visited only once.

The next vertex is chosen using these rules:

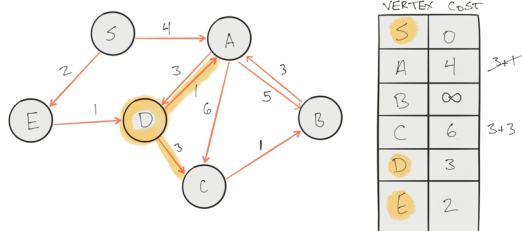
- It must not have been visited previously
- It has the smallest cost of the remaining unvisited vertices In our case, this would be vertex E.

The next vertex that we'll visit is E. The cost to reach E is 2, which is less than 4, and E has not been visited previously:



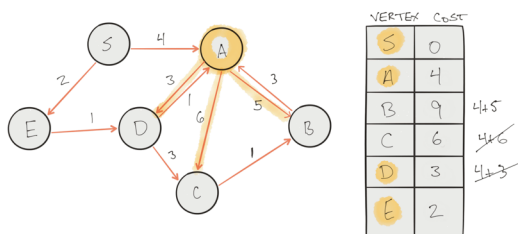
We then update the memo table, setting D to 3 (which is the cost of S to E plus the cost of E to D). We then mark E as visited.

The next vertex in our table that is unvisited with the lowest cost is D, so we calculate that next:



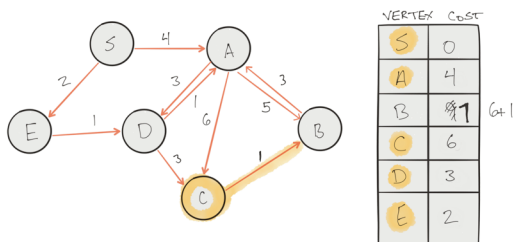
D has two outgoing edges: to A and to C. The cost of A would be $3+1$ which is 4 and no improvement, so we leave A as it is in our table. C has no cost, so we update it to 6, which is the current value of D (3) + the cost to get to C.

The next vertex we'll visit is A. It has the least cost and has not been visited before:



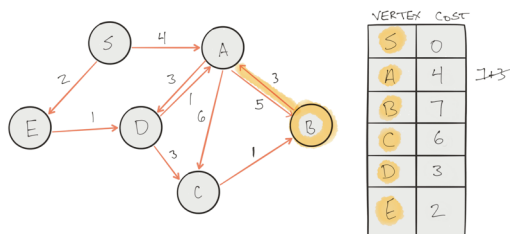
The current cost of A is 4 and the edges going out of A go to vertices D, C and B. The cost of reaching D through A is 7, which is not an improvement of D's current cost of 3, so we leave it. Same with C: reaching C through A does not reduce C's cost, so we leave it. B, however, is still infinity so we'll set it to 9, which is A's cost plus the cost to reach B, which is 5.

C is the next vertex we'll choose as its current cost is 6 and it hasn't been visited:



C has only one outgoing edge: to B. The current cost of C is 6 and adding 1 to it would be less than the current cost of B, so we'll update B's cost to 7.

There is only one vertex left, which is B:



The only candidate for improvement is A, with a current cost of 4. This is less than the tentative cost of B to A, which is 10, so we leave it as it is. We're done!

All vertices are visited, so we're done! We can be sure we calculated the shortest path by using our Bellman-Ford code from the last section to make sure it matches, and it does!

The code you're about to see is probably a lot more verbose than you might write it. I like clarity, mostly for my own sake, because you can bet I'll be looking over this page and this code quite a few times in the future! I want to remember what I was thinking.

To start with, let's alter our implementation of Bellman-Ford in the last section to have a memo table with a little more smarts:

```
//The memoization table, which needs to have some smarts
//using an ES6 class here
class MemoTable{
  //build the table using the passed-in vertices
  constructor(vertices){
    //set the root manually
    this.s = {name: "S", cost: 0, visited: false};
    this.table = [this.s];
    //add the vertices, defaulting cost to infinity and visited to false
    for(var vertex of vertices){
      this.table.push({name: vertex, cost: Number.POSITIVE_INFINITY, visited: false});
    }
  };
  //all non-visited vertices
  getCandidateVertices(){
    return this.table.filter(entry => {
      return entry.visited === false;
    });
  };
  //lowest cost, non-visited vertex
  nextVertex(){
    const candidates = this.getCandidateVertices();
    //if there are candidates, find the one
    //with lowest cost
    if(candidates.length > 0){
      return candidates.reduce((prev, curr) => {
        return prev.cost < curr.cost ? prev : curr;
      });
    }else{
      //otherwise return null
      //this will help determine if we need to
      //iterate
      return null;
    }
  };
  //update current cost
  setCurrentCost(vertex, cost){
    this.getEntry(vertex).cost = cost;
  };
  setAsVisited(vertex){
    this.getEntry(vertex).visited = true;
  };
  getEntry(vertex){
    return this.table.find(entry => entry.name == vertex);
  };
  getCost(vertex){
    return this.getEntry(vertex).cost;
  };
  toString(){
    console.log(this.table);
  }
};
```

I've added the logic for retrieving a given entry as well as updating its values. In addition I've added logic for determining the next vertex to traverse to based on the rules of Dijkstra that we saw above.

Next, we have our graph:

```
//the vertices for our memo table
//I could also use a filter or map on the graph below
//to avoid duplication; but this is nice
//and clear
const vertices = ["A", "B", "C", "D", "E"];
//our graph
const graph = [
  {from : "S", to : "A", cost: 4},
  {from : "S", to : "E", cost: 2},
  {from : "A", to : "D", cost: 3},
  {from : "A", to : "C", cost: 6},
  {from : "A", to : "B", cost: 5},
  {from : "B", to : "A", cost: 3},
  {from : "C", to : "B", cost: 1},
  {from : "D", to : "C", cost: 3},
  {from : "D", to : "A", cost: 1},
  {from : "E", to : "D", cost: 1}
];
```

This array represents the visual graph we worked with above. Now we just need to evaluate our graph using our **MemoTable** functionality:

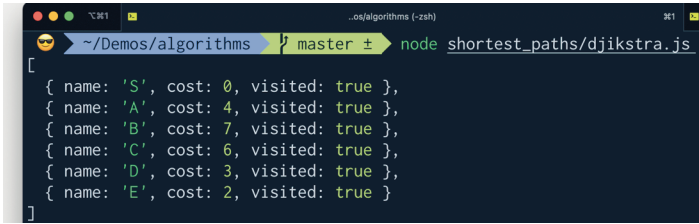
```

//create the table
const memo = new MemoTable(vertices);
//let's do this!
const evaluate = vertex => {
  //get the outgoing edges of the vertex
  const edges = graph.filter(path => {
    return path.from === vertex.name;
  });
  //loop the edges...
  for(edge of edges){
    //calculate the costs
    const currentVertexCost = memo.getCost(edge.from);
    const toVertexCost = memo.getCost(edge.to);
    const tentativeCost = currentVertexCost + edge.cost;
    //if we can improve the cost to the
    //connected vertex...
    if(tentativeCost < toVertexCost){
      //do it!
      memo.setCurrentCost(edge.to, tentativeCost);
    }
  };
  //set this vertex as visited
  memo.setAsVisited(vertex.name);
  //get the next vertex
  const next = memo.nextVertex();
  //if there is a next vertex, let's do this
  //again...
  if(next) evaluate(next);
}
//kick it off from the source vertex
evaluate(memo.S);
memo.toString();

```

The code looks a bit familiar – it follows what I did (for the most part) with Bellman-Ford but this time I’ve added a few tweaks to accommodate setting a vertex as visited, and I’m also using recursion off the memoization table instead of a loop.

What do you think? Can you improve this without losing its clarity? Let’s run it and make sure it works. Again, using Node:



```
~/Demos/algorithms master ± node shortest_paths/dijkstra.js
[
  { name: 'S', cost: 0, visited: true },
  { name: 'A', cost: 4, visited: true },
  { name: 'B', cost: 7, visited: true },
  { name: 'C', cost: 6, visited: true },
  { name: 'D', cost: 3, visited: true },
  { name: 'E', cost: 2, visited: true }
]
```

Right on! That's the exact answer we got above!

COMPILATION

ASSEMBLING CODE AND TURNING IT INTO
SOMETHING THAT CAN BE EXECUTED

Compilation is a set of five discrete steps: lexical analysis, parsing, semantic analysis, optimization and code generation. Every compiler does this exact same thing, but some are more capable than others. The main differences between compilers is in the optimization and code generation steps.

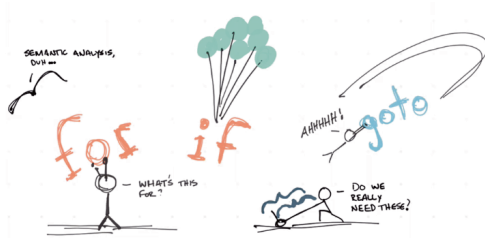
A stack is a way to store program data and is a *per thread process*. The stack is very fast and is used for *value types*. The heap is a *per-application process* and is used for *reference types*. You can make your application more efficient by understanding how these two interact.

Garbage collection is the process of freeing up application memory (the heap) automatically. This is not a free process and requires system resources. There are several strategies used for garbage collection (GC), but tracing is the most widely used.

How a Compiler Works

A compiler simply brings different things together and makes them one thing. In a programming sense, a compiler is a program that reads the code you write and generates something that the runtime engine can execute.

Compilation can happen on command – for instance using a make file. It can happen just in time (JIT) or at runtime with a thing called an interpreter.



They can compile code to different languages (CoffeeScript to JavaScript, e.g) or down to byte code ... or something in between. C#, for instance, compiles to an intermediate language (MSIL), which is then compiled to native code upon execution. This has the advantage of portability – meaning that you could create a compiler for different platforms (Windows 32-bit, 64-bit, Mac, Linux, etc.) without having to change the code.

But how do these things work? Let's look at a high level.

The Compilation Process

Compilation in a computer is just like compilation in your head when you read these words. You're taking them in through your eyes, separating them using punctuation and

white space, and basing the meaning of those words on emphasis.

These words are then turned into meaning in your mind, and at some point sink into your memory ... causing (hopefully) some type of action on your part.

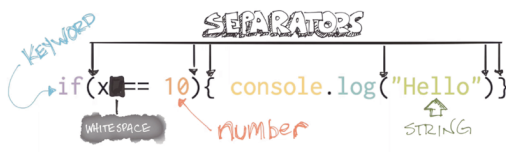
A compiler does the same things, but with slightly larger words. The main compilation steps are:

- Lexical Analysis
- Parsing
- Semantic Analysis
- Optimization
- Code Generation

Every compiler goes through these steps.

Lexical Analysis simply analyzes the code that's passed to the compiler and splits it into tokens. When you read this sentence, you use the whitespace and punctuation between the words to "tokenize" the sentence into words, which you then label.

A compiler will scan a string of code and separate it into tokens as well, labeling the individual elements along the way:



The program within the compiler that does this is called the lexer. So, using our code sample, the lexer will generate these pseudocode tokens (using tuples):

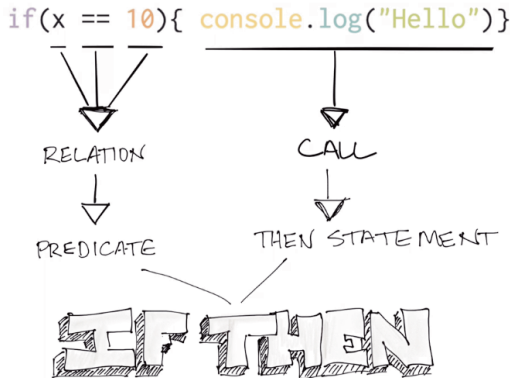
```
{keyword, "if"}
{paren, "("}
{variable, "x"}
{operator, "=="}
{number, "10"}
{left-brace, "{"}
```

The thing being analyzed by the lexer is the lexeme. A token is what's produced from this analysis. Yay for more random words to know!

After the lexer has tokenized the incoming code string, the parser takes over, applying the tokens to the rules of the language – also known as a *formalized grammar*.

Pushing this back into the realm of written language: the words you're reading now are tokenized using whitespace and punctuation – the next process is to parse their meaning and, essentially, what they're supposed to mean.

A parser analyzes the tokens and figures out the implicit structure:



We know, at this point, that we have an if statement. This is just one line of code, of course, however our entire code-

base would be parsed in exactly this way: turning tokens into language structures.

Now we get to the fun part. Semantic Analysis is where the compiler tries to figure out what you're trying to do. We have an if block for our code, but can the predicates be resolved to truthy/falsey expressions? Do we have more than one else associated with it? Consider this sentence:

Kim and Jolene want to go to her party

We can reason that “Kim and Jolene” are the subjects, “to go” is the verb and “party” is the indirect object. But who is her? When a compiler goes through semantic analysis, it must reason through the same thing. For instance:

```
var x = 12;
var squareIt = function(x){
    return x * x;
};
y = squareIt(x);
console.log(y);
```

What will y evaluate to? If you're thinking 144 – you'd be right. Even though we've reused x here, the JavaScript interpreter figured out what we meant, even though we tried to confuse it.

```
var x;
console.log(x);
x = "Hello!";
```

If you're Brendan Eich (creator of JavaScript) and it's 1995 and you have 10 days to create a language – what choices do you make when it comes to semantic analysis?

I'm sure many of you are thinking “hang on – JavaScript is not a compiled language – it's interpreted” and you'd be correct. They still go through the same steps.

I bring up JavaScript because it's precisely these decisions, made at the semantic analysis level, that have caused developers so much confusion over the years. If you've used the language, you'll know what I mean.

Most object-oriented languages are lexically scoped, which means the scope of a variable extends to the block of code that contains it. So, in C#, for instance:

```
public class MyClass {  
    public MyClass(){  
        var x = 100;  
    }  
}
```

The scope of `x` in this case is only within the constructor. You cannot access `x` outside it.

Let's change the code so we can rely on lexical scoping to make `x` available to all properties and methods in `MyClass`:

```
public class MyClass {  
    int x;  
    public MyClass(){  
        x = 100;  
    }  
}
```

All we needed to do was to declare the variable within the `MyClass` code block.

JavaScript, however, does things differently. Scopes in JavaScript are defined by function blocks. So, strictly speaking, JavaScript is sort of lexically scoped.

Consider this code:

```
if(1 === 1){  
    var x = 12;  
}  
console.log(x); //12
```

If lexical scoping was enforced here we should see undefined. But what happens in C#? Let's try it:

```
public class ScopeTest  
{  
    [Fact]  
    public void TheScopeOfXIsLexical()  
    {  
        if(1 == 1){  
            var x = 12;  
        }  
        Console.WriteLine(x);  
    }  
}
```

If I run this test, I get the expected response:

error CS0103: The name 'x' does not exist in the current context

The reason for the difference? A different semantic analysis for the C# compiler vs. the JavaScript interpreter.

There are more issues that I'm sure you're aware of: hoisting, default global scope for variables with **var**, confusion about **this**. There are many things written about these

behaviors and I don't need to go into them here. I bring all of "this" up simply to note the choices made by semantic analysis.

Once the compiler understands the code structures that are put in place, it's time to optimize. Of all the steps in the compilation process, this is typically the longest.

There are almost limitless optimizations that can occur; little tweaks to make your code smaller, faster, and use less memory and power (which is important if you're writing code for phones).

Let's optimize that previous sentence: compiler optimization produces faster and more efficient code. Reads the same, doesn't it? The same meaning, anyway – compilers don't care about creative expression.

Which is a very important point. Modern languages are leaning more on syntactic niceties and shortcuts (aka "*syntactic sugar*"). This is great for developers like me, who enjoy reading code where the intent is clear. It's not so great for the optimizer.

Ruby and Elixir are prime examples of this. Several constructs in these languages are optional (parentheses for example), and a compiler must work through various syntactic shortcuts to figure out the instructions. **This takes time.**

Does a **meaningful_variable_name** need to be 24 characters long. Not to the compiler, which will often rename the variable to something shorter. List operations as well – often you'll see arrays substituted for you, behind the scenes.

How about this rule – does this rule make sense?

$$X = Y * 0 :: X = 0$$

Basically, any time you see a number multiplied by 0, replace it with 0. Seems to make sense ... but ...

```
public class MultiplyingByZero
{
    [Fact]
    public void UsingNaN()
    {
        double x = Double.NaN;
        Console.WriteLine(x * 0); // NaN
    }
}
```

Yeah that won't work because **NaN** \neq 0. It will work if **x** is an integer, however.

Compiler optimizations are at the center of “what makes a language good” – something we'll get into more, later on.

Our code has been parsed, analyzed, and optimized – we're ready for output! This is one of the simpler steps: *we just need to package it up and off we go.*

Intermediate Language Output

Several platforms out there will compile code text files into an intermediate structure, which will be further compiled later on. Code that runs in a virtual machine (like Java) will do this, and later compile it down to machine-level code.

C# does this as well, and compiles into an actual second language called IL (or MSIL). People don't typically read or write IL directly (unless you're Jon Skeet) – but it is possible to dig down into it to see what's going on.

When a C# program is run for the first time, the IL is recompiled down to native machine code, which runs quite fast. It's also JIT compiled every time it's run, after that. The

term JIT stands for *Just In Time* – which usually means “last possible moment”

Getting The JITters

When discussing multistep compilation, you'll often hear other developers talk about “the JITter” and what it will do to your code.

In some cases it will output code optimized for an executable that will then run it. Other times it will produce native byte code that runs at the processor level.

Interpreters

Dynamic languages, like Ruby, JavaScript, or Python, typically use a different approach. I say typically because multiple runtime engines have been created that will produce byte code for you from each of these languages. More on that in a later section.

You can think of these languages as “scripting languages” – in other words, they are scripts that are executed by a runtime engine every time they are invoked.

Ruby, for example, has the MRI (Matz's Ruby Interpreter) which will read in a Ruby file, do all the things discussed above, then execute things as needed. You have to go through the full set of compiler steps *every time a routine needs to be executed*. This is not as fast as native code, obviously, which has already been analyzed, parsed and optimized.

JavaScript is interpreted on the fly in the same way as earlier versions of Ruby, depending on where you use it. The browser will load in and compile any script files referenced on a page, and will hold the compiled code in memory unless/until you reload the browser.

Node works in the same way: it will compile your source

files and hold them in memory until you restart the Node runtime.

LLVM

One of the biggest names in the compilation space (if not the biggest) is LLVM:

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines, though it does provide helpful libraries that can be used to build them. The name “LLVM” itself is not an acronym; it is the full name of the project.

So you can use LLVM “stuff” to build your own compiler. One that you’ve probably heard of is CLANG which is an LLVM-based compiler for C.

You know when you go to install a Node module or Ruby gem and you get some nasty message about native bindings or build tools required? The reason for this is that some of these modules use C libraries that need to be compiled to run. You see this a lot with database drivers, for instance, which want to be superfast.

With Node, you’ll usually see a reference to node-gyp rebuild, which is a module specifically created for compiling native modules.

This can cause headaches, especially when you have other developers working on Windows. The workaround, typically, is to install Visual Studio, referencing C++ build tools during the installation. Installing these things on a Windows

Server is one of the most frustrating things about using Node on Windows in production.

GCC

The GCC project is from GNU:

The GNU Compiler Collection includes front ends for C, C++, Objective-C, Fortran, Java, Ada, and Go, as well as libraries for these languages (libstdc++, libgcj,...). GCC was originally written as the compiler for the GNU operating system.

GCC and LLVM do the same things... sort of. They're both compiler toolchains that also provide compilers for C, C++, Java and Go (among others).

The main differences have been speed (with GCC typically being a bit faster) and licensing. GCC uses the GPL and LLVM uses a license based on MIT/BSD which is a little more permissive.

Apple moved from the GCC to LLVM back in 2010/2011 and it caused a lot of code to break, especially for Rails developers. Those darn native gems!

Garbage Collection

Garbage collection (GC) is the process of cleaning up the heap. It's a facility of managed languages such as C# (and other CLR-based languages), Java (and anything that runs on the JVM) as well as dynamic languages such as Ruby, Python and JavaScript (and many others).

Some languages, like Objective-C, do not have garbage collection directly. This was a choice Apple made to keep their phones as fast as possible. You can, if you want, implement Automatic Reference Counting (ARC) which is a feature of LLVM.

When you write code in Objective-C, you allocate memory as you need and specify pointers vs value types explicitly. When you're done with the variable, you deallocate it yourself.

In previous versions of this book, I made the claim that “most developers don't use ARC”, which is false. I made this claim based on a conversation I had 5 years ago with an iOS developer friend. At that time Objective-C developers needed to know how to manually manage memory, and some didn't trust ARC to work perfectly.

This has changed. Use of ARC is standard these days. A comment from a reader (Adrian Kosmaczew) sums it well:

... after ARC was introduced in 2010 (I was in the room during the WWDC in San Francisco when they announced it) Apple rebuilt pretty much every single app in Mac OS X with it, and ended up removing the garbage collection (which had been added to Objective-C in Mac OS X Leopard, back in 2007) a few years later. iOS never had the GC for the reasons of performance you mention in your book; that is correct. Having a “mark and sweep” GC was too much to handle for the small CPU of the original iPhones, and as such we devs ended up using “retain” and “release” calls to make sure objects didn't die before we had a chance to use them.

The original Objective-C runtime (originally designed for the NeXTSTEP operating system in 1989)

used a simpler GC scheme based in resource counting. What ARC does is to perform a static analysis of the source code and to insert the calls to “retain” and “release” automatically where needed. The final binaries are 100% compatible with older versions of OS X and iOS, because there's no runtime library needed.

Now the Objective-C GC is 100% deprecated, and everybody in the “Apple galaxy” uses ARC these days. There are only two known cases of retain cycles one has to take care of when using it (that is, from a lambda to an object holding it, and from an object to another through strong references) but apart from that, it works beautifully well. Actually, the concept was quite revolutionary, it trades a bit of a longer compilation cycle with binaries that literally never leak.

There are several things to know about GC:

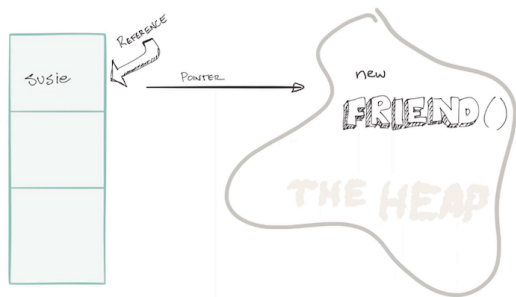
- It's not free. Determining what objects and memory are subject to collection requires overhead and can slow things down
- It's undecidable. This is as Turing illustrated with the Halting Problem: it's just not possible to know if a process will complete, therefore it's not possible to know if memory will ever not be needed
- It's the focus of a lot of amazing work. Speeding up GC means speeding up the runtime, so language vendors spend much time on it.

Saying that GC “cleans up the heap” is not going to satisfy the curious, so let's dive into it a bit.

As with all things computer science, there are multiple ways to go about reducing the memory size of the heap and they all center on probability analysis. Some are sophisticated, some are very outdated and buggy.

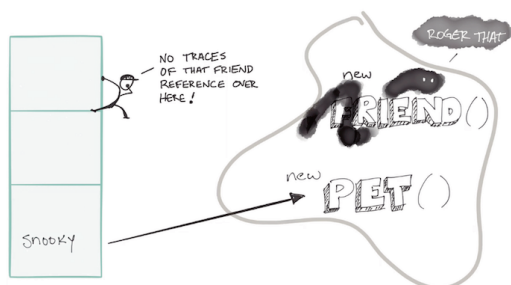
The most common strategy for GC is tracing, to the point that unless you say otherwise, people will assume this is what you mean.

Tracing is the most widely used method of garbage collection. When you create an object on the heap, a reference to it exists on the stack, as we've discussed.



As your program runs, the garbage collector will take a look at a set of “root” objects and their references, and trace those references through the stack. Objects on the heap can refer to other objects, so the trace can be much more complex than what is represented here.

As the trace runs, the GC will identify objects that are not traceable – meaning they aren’t reachable by other objects that are traceable. The untraceable objects are then deallocated and the memory freed.



The advantages of tracing are:

- It's accurate. If objects aren't referenced they are targeted for deallocation and that's that.
- It's easy. You just have to find the objects!
- The disadvantages are:
- When will the GC get around to executing?
- What happens when there are multiple threads?
The stack for one thread may not have any references, but what about the other threads running?

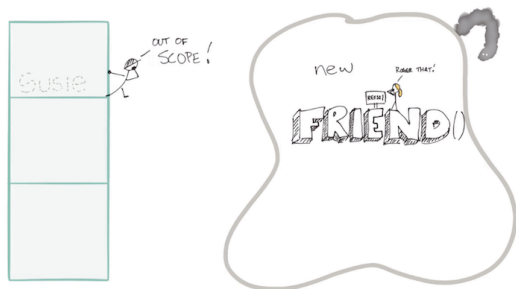
Obviously, it gets tricky. Within the tracing strategy there are various algorithms for marking and chasing down memory to be cleaned up, with different levels of speed and complexity. Both Java and .NET use tracing for GC, both implementing various flavors of generational (or ephemeral) implementations.

This is a deep topic, full of algorithms, probability and statistical analysis ... and I could fill up many pages on the smallest details. Alas I have the rest of the book to write and so I need to clip the discussion here.

If you want to know more, Google away!

Reference counting works almost exactly like tracing, but

instead of running a trace, an actual count of references is made for each object. When the count goes to 0, the object is ready to be deallocated.



The advantages to reference counting are:

- It's simple. Objects on the heap have a counter which is incremented/decremented based on references to that object (from both the heap and the stack) rather than a trace algorithm.
- There's less guesswork as to "when" GC will happen. When local reference variables fall out of scope, they can be decremented right away. If that count goes to 0 then GC can happen shortly thereafter
- The asymptotic complexity (Big-O) of reference counting is $O(1)$ for a single object, and $O(n)$ for an object graph

Sounds simple and obvious, doesn't it? What about this code (pseudocode):

```

public class Customer{
    public int ID {get;set;}
    //...

    public Order CurrentSalesOrder{get;set;}
    public Customer(int id){
        //fetch the current order from the db
        this.CurrentSalesOrder = db.getCurrentOrder(id);
    }
}

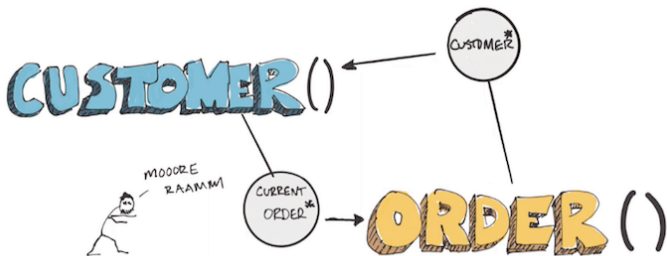
public class Order{
    public int ID {get;set;}
    //...

    public Customer Buyer{get;set;}
    public Order(GUID key){
        this.Customer = db.getCustomerForOrder(key);
    }
}

```

This code is quite common. What we have here is a circular reference on the heap. You could create an instance of the **Customer** and along with it comes a reference to an **Order** ... which has a reference back to the **Customer**.

The reference count for these objects will always be > 1 , so they are, in effect, little memory bombs eating up RAM.



If you have a sophisticated enough compiler, it should be able to analyze what variables need what memory, where, and (possibly) for how long. Without it, Objective-C developers have had to manage memory themselves, allocating and deallocating within the code (much like C).

The XCode compiler analyzes the code written and decides the memory use upfront, freeing the runtime from the overhead of GC. It does this by using the notion of strong, weak, and unowned objects, each with an explicit level of protection that would, again, take me pages to explain properly (along with some goofy drawings).

The essence is this: using strong variables keeps them in memory longer, based on other strong references they're related to. If you have a weak reference to an object it tells the compiler "no need to protect the referenced object for longer than now". Finally with unowned you're making sure that a strong reference won't keep an object alive for longer than you want – so you explicitly say "make sure this object goes away".

If you want to know more about this, there's a great article [here](#) that discusses the ideas in terms of the human body:

A human cannot exist without a heart, and a heart cannot exist without a human. So when I'm creating my Human here, I want to give life to him and give him a Heart. When I initialize this Heart, I have to initialize it with a Human instance. To prevent the retain cycle here that we went over earlier (i.e. so they don't have strong references to each other), your Human has a strong reference to the Heart and we initialize it with an unowned reference back to Human. This means Heart does not have a strong reference to Human, but Human

does have a strong reference to Heart. This will break any future retain cycles that may happen.

OBJECT-ORIENTED DESIGN PATTERNS

THE ESSENTIAL GANG OF FOUR PATTERNS WITH
EXAMPLES

People have been writing code in object-oriented languages for a long time and, as you might guess, have figured out common ways to solve common problems. These are called design patterns and there are quite a few of them.

In 1994 a group of programmers got together and started discussing various patterns they had discovered in the code they were writing. In the same way that the Romans created the arch and Brunelleschi created a massive dome – the Gang of Four (or “GoF” as they became known) gave object-oriented programmers a set of blueprints from which to construct their code. The Gang of Four are:

- Erich Gamma
- Richard Helm
- Ralph Johnson
- John Vlissides

The Code

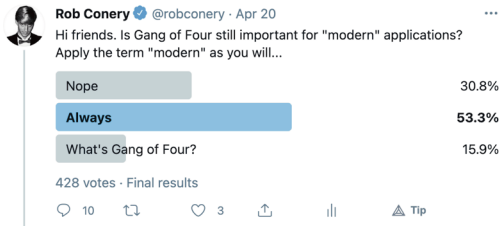
The code for this chapter is, once again, shown in screenshots. If you'd like to play along or have access to the code, you can view it or download it from our GitHub repo. I also have 10 hours of video walking you through the concepts in this book.

Rethinking Gang of Four

It's 2021 and I'm revisiting this chapter and redoing a chunk of it. Software has evolved and so have programming languages. A large part of our community no longer conceives of a software program as a singular structure as the result of a group effort. Instead, we're composing elements in smaller pieces and orchestrating them using containers, "pods" and services. The "modern" developer is paying the same price as the classical one when ignoring what's come before.

To that end, *what's come before* must be understood *now*. Only, however, **if it's useful**.

In terms of *usefulness*, I went to the oracle to ask if GoF is still useful. Consider the result:



Twitter, of course, is Twitter. But if I consider my undertakings over the last 5 years I wonder if there is a reckoning afoot. I think about these patterns a lot, but if I'm honest, I tend to plow ahead with *The Way I Do Things* (which I'll share with you at the end of the following chapter). This is hard to admit and I feel like many of my experienced friends will be clutching their pearls...

To quote Morrissey: *has the world changed or have I changed?*

In previous incarnations of this book, I opened this chapter with this warning:

This entire chapter will be argumentative. I hate to say it, but there's just no escaping it: how we build software is still evolving. You might disagree with what you read here, which is fine. I'll do my best to present all sides – but before we get started, please know this: I'm not arguing for or against anything.

These concepts exist; you should know they exist. You should know why people like or dislike them and, what is more important, what they argue about when they argue. And oh how they do.

That's not exactly a strong opening. As I read it now, I'm feeling the weight of the hatchet in my hand. The Bridge Pattern, as complex as it is, is simply no longer considered by the bulk of programmers out there. It could occupy a dusty shelf in a programmer's curio shop, I suppose, but that's not what I'm offering with this book. I want to cleave it (and many other patterns) from this book because they're part of a past that I'm not sure is still relevant.

This isn't a good place to be in, as a writer. I *need* to write about history in this book but I also *need* to be sure I take

care with what I include, omit and revise. You never know what the future holds in the CS world – I (seriously) just read that Fortran is making a come back. FORTRAN! I don't even know what's real anymore...

Here's the question: I feel like GoF is fading, but should I write about it anyway? Are these things going to be important to you tomorrow, next week, next month?

Who knows, maybe implementing a pattern or two will solve a massive problem you have! Alternatively: you might read this and say “wow Bridge is super cool I'll give this a go” and you write a bunch of code which slows down your app and confuses your team and you get fired. People do weird stuff with Gang of Four.

Let's approach this two ways. The first being, of course, that knowing them is better than not because you might spot one in a code review and be able to ask the person who implemented that pattern why they did why they solved a particular problem that way and then proceed with a thoughtful conversation. The second is that knowing about patterns broadens your mind into how developers in decades past have solved problems (aka “history”) – as long as you know it's historical. This is always valuable. Knowledge is a good thing, so let's proceed thus.

I'm simultaneously glad that I know GoF patterns and also sad that I had to live through the Patterning Carousel during the enterprise heyday of Java and .NET. To me, patterns are useful, but if I could impart only a single thing in this brief introduction: **patterns don't make your code correct.**

You do that. That's what you're paid for. No pattern is going to fix your software problems.

- Rob Conery, Hanalei HI, May 2021

Creational Patterns

When working with an OO language you need to create objects. It's a simple operation, but sometimes having some rules in place will help create the correct object, with the proper state and context. You're familiar with this if you've used any object-oriented language so I'll be brisk.

Most OO languages have a built-in way of creating an instance of a class and it typically involves the keyword **new**. I will assume you know how this works if you've written code.

In most languages you can implement a constructor that takes in the parameters it needs to exist:

```
class User {  
  constructor({name, email}){  
    this.name = name;  
    this.email = email;  
  }  
}  
  
//our constructor  
const user = new User({name: "Rob", email: "rob@bigmachine.io"});
```

This is perfectly usable. As with all the GoF patterns, their utility (or lack of) becomes apparent as the program grows. For instance: there's no context here. Our user is being created and returned with a simple parameter assignment. While that's fine for small apps just getting off the ground, more complex apps benefit from something more concrete.

That's where our next pattern comes in. This is also where I will be switching to a more pattern-prone language, C#. If you don't know C#, don't despair! If you know JavaScript or any C-family languages it should be quite familiar.

Sometimes instantiating an object can be quite involved, and might require a little more clarity. This is where the Factory Pattern comes in.

For instance: our **Customer** class might have some defaults that we want set:

```
using System;
using System.Collections;
using System.Collections.Generic;

public class Order{}

public class Customer
{
    public string Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public string Status { get; set; }
    public List<Order> Orders { get; set; }

    public static Customer FromDefaults ()
    {
        var customer = new Customer { Status = "unregistered", Name = "Guest" };
        customer.Orders = new List<Order> ();
        return customer;
    }
    public static Customer FromExisting (IDictionary values)
    {
        var customer = new Customer ();
        //populate the values on the class, validating etc.
        return customer;
    }
}
```

Our **Customer** class isn't terribly complex, but you do gain some clarity by calling **Customer.FromDefaults()**. This can become important as your code base grows because it's not clear what's going on if you simply use **new Customer()**.

For complex class construction you could create a dedicated factory class. You see this often in Java. For instance, we could pull the instantiation logic completely out of our **Customer** and into a **CustomerFactory**:

```

public class CustomerFactory
{
    public Customer FromDefaults ()
    {
        var customer = new Customer { Status = "unregistered", Name = "Guest" };
        customer.Orders = new List<Order> ();
        return customer;
    }

    public Customer FromExisting (IDictionary values)
    {
        var customer = new Customer ();
        //populate the values on the class, validating etc.

        return customer;
    }
}

var customerFactory = new CustomerFactory();
var customer = customerFactory.FromDefaults();

```

Again: this is a bit simplistic. You can do a lot more with a factory class, such as deciding which object to create altogether. Your application might have the notion of an **Administrator** that inherits from **Customer**:

```

public class Administrator : Customer {
    //specific fields/methods for admins
}

```

You can use a variation of the Factory Pattern (called the Abstract Factory Pattern) to decide whether an **Administrator** should be returned or just a **Customer**:

```

public class AbstractCustomerFactory
{
    public Customer FromDefaults ()
    {
        var customer = new Customer { Status = "unregistered", Name = "Guest" };
        customer.Orders = new List<Order> ();
        return customer;
    }

    public Customer FromExisting (IDictionary values)
    {
        if (values.Contains ("Email")) {
            if (values ["Email"].ToString () == "admin@example.com") {
                var admin = new Administrator ();
                //populate values
                return admin;
            } else {
                var customer = new Customer ();
                //populate the values on the class, validating etc.

                return customer;
            }
        } else {
            return null;
        }
    }
}

var customerFactory = new CustomerFactory();
var customer = customerFactory.FromDefaults();

```

This pattern is useful, but it can spiral on you if you are really into patterns. Consider this question on [StackOverflow](#):

What is a good name for class which creates factories?
(FooFactoryFactory sounds silly imo)?

This happens with C# as well:

I make extensive use of the interface-based Typed Factory Facility in Windsor, but there are times when I

must pass a lot of arguments to a factory around with the factory itself. I'd much prefer to create a factory factory with these arguments so that I don't need to muddy up the constructors of objects more than I need to.

BUILDER

The Factory pattern can only do so much until it becomes too convoluted. This usually happens with very complex objects. Many developers consider this a “code smell” (when you find yourself needing it, it means there's a simpler way). Overly complex objects are ripe for bugs and, typically, means you've probably over-thought your solution.

There are times, however, that a Builder makes sense. Consider a class that .NET developers use all the time: **System.Text.StringBuilder**.

Strings are immutable in C#, so if you try to build a string from many string fragments, you can run into the memory problem as seen here:

```

public class NaiveStringBuilder {
    IList<string> _strings;
    public NaiveStringBuilder(){
        _strings = new List<string>();
    }
    public void Append(string val){
        _strings.Add(val);
    }
    public override string ToString(){
        var result = "";
        foreach (var s in _strings) {
            // a new string is built each time
            result = result + s + " ";
        }
        return result;
    }
}

var naiveBuilder = new NaiveStringBuilder();
naiveBuilder.Append("This");
naiveBuilder.Append("could be");
naiveBuilder.Append("very long");
naiveBuilder.Append("and blow up");
naiveBuilder.Append("your program...");
var result = naiveBuilder.ToString(); //BOOM

```

If you ever find yourself writing a string concatenation routine in a loop, stop. It's a memory problem just waiting to happen.

The good news is that the C# team contemplated this and decided to help out, using the Builder pattern with **System.Text.StringBuilder**:

```

var goodBuilder = new System.Text.StringBuilder();
goodBuilder.Append("This ");
goodBuilder.Append("won't ");
goodBuilder.Append("blow up ");
goodBuilder.Append("my program ");
var result = goodBuilder.ToString(); //yay!

```

If you're curious about how the **StringBuilder** works, you can view the source code online. There's a lot going on in there! The thing to take away, however, is that an instance of an object (**System.String**) is being built for us in a very specific way to avoid problems. This is what the Builder Pattern is good for.

There is a more elegant way of doing this, however, which we'll see next.

Instead of calling `stringList.Add(...)` or using a **StringBuilder** directly, you can encapsulate what you're doing into a class that uses a fluent interface, otherwise known as Method Chaining:

```
public class Message
{
    System.Text.StringBuilder _stringBuilder;
    public Message (string initialValue)
    {
        _stringBuilder = new System.Text.StringBuilder ();
        _stringBuilder.Append (initialValue);
    }
    public Message Add (string value)
    {
        _stringBuilder.Append (" ");
        _stringBuilder.Append (value);
        return this;
    }
    public override string ToString ()
    {
        return _stringBuilder.ToString ();
    }
}

var message = new Message("Hello")
    .Add("I might be")
    .Add("a really long string")
    .ToString();
//Hello I might be a really long string
```


A Singleton is a class that only allows one instance of itself. It's not an easy thing to do correctly and many blog posts have been written about the perils of Singletons and threading or multiple processes.

You should know the pattern, however. Here's a rather naive one in C#:

```
using System;
namespace Singleton
{
    public class SingleThing
    {
        //single instance holder
        private static SingleThing _instance;
        //disallow calling constructor directly
        protected SingleThing () { }
        //access to the instance
        public static SingleThing Instance ()
        {
            if (_instance == null) {
                _instance = new SingleThing ();
            }
            return _instance;
        }
    }
}
```

The problem with this code is that it will likely work fine *most* of the time. Until it gets used more and the Instance method is called simultaneously, and a nasty collision happens. Or if, more likely, someone decides to use your code in a threaded environment.

Interestingly, in JavaScript land, Node implements Singletons for its moduling system but it's almost by accident. Node is single threaded and when it starts it loads every module into memory, caching it, which creates a Singleton.

Running Node in production, however, usually means

spinning up multiple instances of Node to handle the load of your application. I'm sure your Singleton code has considered all the other Singletons running in the other instances, yes? Your database connection, for instance, when executing a transaction would *never* do a row-level lock on data your other Singleton needs!

If you find yourself trying to lean on a Singleton the best advice I can give is, *please stop*. You're going to get it wrong. Everyone does and the realization usually comes in the middle of the night during a frantic phone call at 2am. Yes, I know you *think* you can do it, but I would urge you to consider a different approach. You might be wondering at this point: *what of Node then?*

It's true, you're going to need to deal with multiple instances of your application running in memory, if you've used Node. It's at this point where we discuss database consistency rules, row and table locks and whether you need to think about all of this in your application.

I say you do. We'll get into this in a later chapter but throwing a deadlock on a table is one of the first issues that multi-process (or threaded) apps must deal with. I'm going to ask you to hold tight. We're going to discuss concurrency soon and I'll bring this back up.

Structural Patterns

Code needs to have some structure, so we use things like methods, modules, and classes. As code becomes more complex, these modules and classes might also need some structure to reduce confusion and unneeded complexity. That's what we'll look at here.

The Adapter Pattern is all about making one interface work with another. You see them used often with data access tools (ORMs) where the abstracted query interface needs to

work against different databases, such as PostgreSQL, SQL Server, etc.

For instance, we might create a way of working with a database that we really like, so we'll abstract it into a basic interface of some kind:

```
public abstract class GroovyQuery{
    //groovy interface
    //find
    //fetch
    //save
}

public class GroovyPostgresAdapter: GroovyQuery{
    //implements groovy interface for PostgreSQL
}

public class GroovySQLServerAdapter: GroovyQuery{
    //implements groovy interface for SQL Server
}
```

You just need to pick the correct adapter for the database you're working with. This can be done manually or by way of configuration and some higher-level patterns which we'll see later on.

The Bridge Pattern is quite subtle and tends to look a lot like the Adapter Pattern, but it's one step up the abstraction curve. You use the Bridge Pattern when your abstraction gets complicated enough that you need to split things out.

People really like our **GroovyQuery** tool and we want to add a feature: document queries. It turns out that you can store JSON happily in PostgreSQL and also in SQL Server – so we decide to implement a document API that handles parsing and so on:

```

public abstract class GroovyQuery{
    //groovy interface
    //Find
    //Fetch
    //Save
    public abstract T GetDocument<T>();
    public abstract T SaveDocument<T>();
    public abstract IList<T> FetchDocuments<T>();
    //etc
    //etc
}

```

This is a very interesting idea! The problem is that we now have to go and implement it for every adapter. Unless we abstract the document interface and bridge it to our **GroovyQuery**:

```

public abstract class GroovyQuery{
    //groovy interface
    //Find
    //Fetch
    //Save
    public IDocumentQueryable Documents();
    //etc
}

//a document query interface
public interface IDocumentQueryable{
    T Get<T>();
    T Save<T>();
    IList<T> Fetch<T>();
}

```

Here's how we might implement it:

```

//implementation of the document query interface for
//relational systems.
public class RelationalDocumentQueryable : IDocumentQueryable{
    GroovyQuery _adapter;
    public RelationalDocumentQueryable(GroovyQuery adapter){
        this._adapter = adapter;
    }
    //implement Get, Save, Fetch
}

//our SQL Server adapter
public class GroovySQLServerAdapter: GroovyQuery{
    public GroovySQLServerAdapter(){
        this.Documents = new RelationalDocumentQueryable(this);
    }
    //implement Get, Save, Fetch
}

```

The neat thing about this new structure is we can change our **IDocumentQueryable** interface and the implementation, without breaking any of our adapters.

The Composite Pattern deals with parent-child relationships that are composed to create a whole object. They can grow and shrink dynamically, and child objects can move between parents.

Our **GroovyQuery** tool is really picking up steam! People are really happy with it, mostly because we have a cool document abstraction, they can use next to your typical ORM interface. The problem is we need more speed!

It turns out that some of the drivers we've been using don't implement connection pools – basically a set of 10 or so open connections to the database that we keep alive so we don't need to take the time establishing a connection for each query.

We can create our own incredibly naive implementation using the Composite Pattern. We'll start by defining a **Connection**:

```

public class Connection
{
    public bool CheckedOut { get; set; }
    public Connection (string connectionString)
    {
        //connect
    }
    public void Close ()
    {
        //close the connection
    }
}

```

Now we manage that class with a **ConnectionPool**:

```

public class ConnectionPool
{
    public IList<Connection> Pool;
    public ConnectionPool (string connectionString)
    {
        this.Pool = new List<Connection> ();
        for (var i = 0; i < 10; i++) {
            this.Pool.Add (new Connection (connectionString));
        }
    }
    public void Checkout ()
    {
        //grab a list of connections which aren't checked out
        //return the first
    }
    public void Checkin ()
    {
        //tick the boolean
    }
    public void Drain ()
    {
        foreach (var connection in this.Pool) {
            connection.Close ();
        }
        this.Pool = new List<Connection> ();
    }
}

```

I hesitated to show a **ConnectionPool** example as I'm sure many of you will be poking holes in it (as you should)! Pooling is a hard thing to do, and I don't recommend writing your own. I include it here because it's a real-world example that's easily understood (as opposed to the mind-numbing Foo and Bar nonsense you see everywhere).

If the **ConnectionPool** goes away, so do all the connections. If there are no children (in other words the **IList<Connection>** is empty, there is no **ConnectionPool**. The parent and children work together to provide functionality.

If you work in an IDE (such as Visual Studio or Eclipse) – each of the UI elements you see is a component that has a parent. This, again, is the Composite Pattern.

The Decorator Pattern adds behavior to an object at runtime. You can think of it as “dynamic composition”.

We could use the Decorator Pattern as an alternative to the Bridge Pattern above for our **GroovyQuery** engine. We still have the same core bits:

```
public abstract class GroovyQuery
{
    //groovy interface
    public abstract T GetDocument<T> ();
    public abstract T SaveDocument<T> ();
    public abstract IList<T> FetchDocuments<T> ();

    public IDocumentQueryable Documents;
    //etc
}

public interface IDocumentQueryable
{
    T Get<T> ();
    T Save<T> ();
    IList<T> Fetch<T> ();
}
```

But now we get to add Decorators

```

//implementation of the document query interface for
//relational systems.
public class RelationalDocumentDecorator : IDocumentQueryable
{
    GroovyQuery _adapter;

    //Find, Fetch, and Save use the _adapter passed in
    public RelationalDocumentDecorator (GroovyQuery adapter)
    {
        this._adapter = adapter;
    }

    public IList<T> Fetch<T> ()
    {
        throw new NotImplementedException ();
    }

    public T Get<T> ()
    {
        throw new NotImplementedException ();
    }

    public T Save<T> ()
    {
        throw new NotImplementedException ();
    }
    //implement Get, Save, Fetch for Documents below
}

```

With our **RelationalDocument** Decorator we're able to "decorate" the **GroovyQuery** base object with the ability to work with JSON documents, for instance.

A Facade hides implementation details so clients don't have to think about it. We can use a Facade for our **GroovyQuery** to pick an adapter for the calling code, so they don't need to worry about how to wire things together.

We start by defining the classes we'll need:


```
//abstract base class
public abstract class GroovyQuery
{
    public GroovyQuery (string connectionString) { }
}

//implementation for PostgreSQL
public class PostgreSQLQuery : GroovyQuery {
    public PostgreSQLQuery (string connectionString) : base (connectionString){}
}

//implementation for SQL Server
public class SQLServerQuery : GroovyQuery
{
    public SQLServerQuery (string connectionString) : base (connectionString) { }
}
```

We then decide which class to using the Façade pattern, which amounts to a bunch of **if** statements or, sometimes, a **switch**:

```
//a simple class that hides the selection details
public class QueryRunner
{
    string _connectionString;

    //Find, Fetch, and Save use the _adapter passed in
    public QueryRunner (string connectionString)
    {
        _connectionString = connectionString;
    }

    public void Execute ()
    {
        GroovyQuery runner;
        if (_connectionString.StartsWith ("postgresql://", StringComparison.InvariantCultureIgnoreCase)) {
            runner = new PostgreSQLQuery (_connectionString);
        } else if (_connectionString.StartsWith ("sqlserver://", StringComparison.InvariantCultureIgnoreCase)) {
            runner = new SQLServerQuery (_connectionString);
        } else {
            throw new InvalidOperationException ("We don't support that");
        }

        //execute with the runner
    }
}
```

In the initial versions of **GroovyQuery** we decided it would be very useful to introspect our database whenever a

write needed to happen (insert or update query). We did this because knowing more about each table (data types, primary key fields, column names, and default values) would be extremely helpful in crafting up a very usable API.

Unfortunately, this became very slow when the system came under load, so we opted to implement the Flyweight Pattern.

Now, when **GroovyQuery** starts up, it runs a single query that introspects every table in our database, and then loads up a series of small objects that can be used throughout our application:

```
//our Flyweight class
public class Table
{
    public string Name { get; set; }
    public string PrimaryKeyField { get; set; }
    //column and data type information...
}

public abstract class GroovyQuery
{
    //the API as we've come to know it
    List<Table> _tables;
    public void Initialize ()
    {
        _tables = new List<Table> ();
        //query the database for meta information
        //load up the _tables list
    }
}
```

Now, whenever we run an insert or update query we can reuse one of the **Table** instances we have in memory, avoiding the need to make a special query call on each write operation. This pattern can scale to a large number tables, and helps to scale our app to thousands of write operations per second.

Behavioral Patterns

We've figured out various ways to create our **GroovyQuery** class as well as how to enable functionality by structuring things a certain way. Now let's see how we can use patterns to simplify how clients can use our **GroovyQuery** operations.

Of all the patterns, Behavioral are the most complex. They are also the most useful *when you need them*. When I asked on Twitter that started this chapter (whether GoF patterns are still useful), my friend Jimmy Bogard replied thus:



Jimmy Bogard 🍌
@jbogard

...

Replying to @robconery

I still use the behavioral patterns but there are a lot more variations and language features as options

Jimmy creates large, complex applications so he needs these things. I'll get more into large applications in a later chapter, however, if you find yourself on a project creating a stock exchange, you'll need to know these things.

We've decided to implement validations for our User and must orchestrate a bit of an approval chain. We can use the Chain of Responsibility Pattern for this, which is focused on moving data through a set of handlers.

There Is A Better Way

Moving objects through a process chain can be subject to many high-level patterns that are, frankly, much better than this one. I'm showing you this example because you should know the pattern – but when it comes to validations there are better ways to do this.

The first thing to do is to create an abstract handler class:

```
//our handler class
public abstract class UserValidator {
    protected UserValidator Successor = null;
    public void SetSuccessor (UserValidator successor)
    {
        this.Successor = successor;
    }

    public abstract void Validate (User user);
    public void HandleNext (User user)
    {
        if (user.IsValid && this.Successor != null) {
            this.Successor.Validate (user);
        }
    }
}
```

This handler will allow us to define our **Validate** routines as well as any successors that we might have. Now we can create individual validations, starting with a **NameValidator**:

```
public class NameValidator : UserValidator
{
    public override void Validate (User user)
    {
        user.IsValid = !String.IsNullOrEmpty (user.Name);

        if (user.IsValid) {
            user.ValidationMessages.AppendLine ("Name validated");
        } else {
            user.ValidationMessages.AppendLine ("No name given");
        }
        HandleNext (user);
    }
}
```

We can also implement an **AgeValidator**:

```

public class AgeValidator : UserValidator
{
    public override void Validate (User user)
    {
        user.IsValid = user.Age > 18;
        if (user.IsValid) {
            user.ValidationMessages.AppendLine ("Age validated");
        } else {
            user.ValidationMessages.AppendLine ("Age is invalid - must be over 18");
        }
        HandleNext (user);
    }
}

```

One of the nice things about using Chain of Responsibility is that we can formalize our validations into classes that target a single use case, rather than writing a ton of validation code onto our **User**.

Speaking of, let's create our **User** class and orchestrate the validations:

```

public class User
{
    public string Name { get; set; }
    public int Age { get; set; }
    public bool IsValid = false;
    public System.Text.StringBuilder ValidationMessages;

    public User ()
    {
        this.ValidationMessages = new System.Text.StringBuilder ();
        this.ValidationMessages.AppendLine ("Pending save");
    }

    public void Validate ()
    {
        var nameCheck = new NameValidator ();
        var ageCheck = new AgeValidator ();
        nameCheck.SetSuccessor (ageCheck);

        //kick it off
        nameCheck.Validate (this);
    }
}

```

The **IsValid** and **ValidationMessages** properties will let

us know if we can save this user, and what's happened during our validation process.

The user's **Validate** routine is where the orchestration is at. We instantiate the name and age validators, and then decide which goes first. We could use Method Chaining here – but I think this is clear enough.

Now we just need to kick it off:

```
var user = new User { Name = "Larry", Age = 22 };
user.Validate();
user.IsValid //true
user.ValidationMessages.ToString() //Pending Save, Name validated, Age validated

var user2 = new User { Name = "Larry", Age = 16 };
user2.Validate();
user2.IsValid //false
user2.ValidationMessages.ToString() //Pending Save, Name validated, Age is invalid - must be over 18
```

As you can see, our validations have gone off and we have errors we need to correct.

The Command Pattern formalizes requests from one API to the next.

Our data access tool, **GroovyQuery**, is all about writing and reading data from the database. It does this by creating SQL statements that our adapter then executes. We could do this by passing in a SQL string and a list of parameters – or we could formalize it into a command.

We'll start by creating a class for our query parameters:

```

public class QueryParameter
{
    public QueryParameter (string name, string value)
    {
        this.Name = name;
        this.Value = value;
    }
    public string Name { get; private set; }
    public string Value { get; private set; }
}

```

Next we'll create an interface that describes a query command as well as a class that implements it:

```

public interface IQueryCommand
{
    string SQL { get; set; }
    IList<QueryParameter> Parameters { get; set; }
    IDbCommand BuildCommand ();
}
public class QueryCommand : IQueryCommand
{
    public string SQL { get; set; }
    public IList<QueryParameter> Parameters { get; set; }
    public IDbCommand BuildCommand ()
    {
        //return a command that can be executed
        //...
        return null;
    }
}

```

This is our new **GroovyQuery** class, which takes any **IQueryCommand** and executes it:

```

public class GroovyQuery
{
    //the API
    //...
    public IDataReader Execute (IQueryCommand cmd)
    {
        //build the command and execute it
        var dbCommand = cmd.BuildCommand ();
        //...
        return null;
    }
}

```

One thing about formalizing a request like this is that we can scale it to specific needs:

```

public class CreateUserCommand : QueryCommand
{
    public CreateUserCommand (string name, string email, string password)
    {
        this.SQL = @"insert into users(name, email, hashed_password)
            values(@1, @2, @3)";

        this.Parameters = new List<QueryParameter> ();
        this.Parameters.Add (new QueryParameter("@1", name));
        this.Parameters.Add (new QueryParameter("@2", email));
        this.Parameters.Add (new QueryParameter("@3", SomeHashingAlgorithm (password)));
    }
    private string SomeHashingAlgorithm (string val)
    {
        //some solid hashing here...
        return "";
    }
}

```

A little naive, perhaps, but this command encapsulates what it means to add a **User** to our system for SQL and the parameters required.

For what it's worth, I really like the Command pattern. It is a nice alternative to the Repository Pattern which we'll talk about later. In fact, there is a whole "sect" (for lack of better words) in the OO world that uses an architectural pattern called "Command Query Responsibility Separation", aka

CQRS. The idea is a simple one: rather than convoluting routines in multiple classes (models, services and so on) you divide things up into simple commands and queries. We'll get into this more in the Architectural Patterns chapter coming up soon.

We want to formalize our document storage capabilities, however adding methods and abstractions to our **Groovy-Query** will make the API more complex, which goes against some programming principles we'll discuss in a later chapter.

In short: simplicity is our goal. We want our class abstractions to do one thing and to do it well.

Let's formalize our document storage idea with the Mediator Pattern. A Mediator is simply a class that sits between two other classes, facilitating communication. It's often used in message-based applications, but we can use a simplified version here:

```

public class DocumentStore {
    GroovyQuery _adapter;
    public DocumentStore (GroovyQuery adapter) {
        _adapter = adapter;
    }
    public T Save<T> (T item) {
        //parse and save the object
        return item;
    }
    public T Get<T> () {
        //pull the record, dehydrate
        return default(T);
    }
    public IList<T> Fetch<T>() {
        //pull the list, dehydrate
        return new List<T>();
    }
    string Dehydrate<T>(T item) {
        //turn the object into JSON
        return "";
    }
    T Hydrate<T> (string json) {
        //resolve
        return default(T);
    }
}

```

Here, we're mediating between our database adapter and any class type of **T**. The adapter doesn't need to know anything at all about **T**, and **T** knows nothing about the adapter that's passed to it other than the fact that it's an abstract **GroovyQuery**.

This is one of those patterns that would cause me to check myself and what I'm doing with my application. I've never had need of a pattern like this but I'm sure it exists somewhere. I have several .NET friends that use this kind of thing regularly. I prefer simpler and smaller and if that means breaking a single application into smaller ones, so be it.

The Observer Pattern facilitates event-based program-

ming. You use this pattern whenever you wire up events in a language like C# or JavaScript (using the **EventEmitter** in Node or listening to DOM events in the browser).

Many frameworks have the mechanics for observation already built in, but let's look at how we can construct an observer by hand by adding methods to our **GroovyQuery** that get fired when certain events occur. These are commonly referred to as callbacks:

```
public interface IListener
{
    void Notify<T> (T result);
    void Notify ();
}
public abstract class GroovyQuery
{
    //API methods etc
    //...
    public IList<IListener> Listeners { get; set; }
    public GroovyQuery ()
    {
        //constructor stuff
        //...
        this.Listeners = new List<IListener> ();
    }
    public virtual IDataReader Execute (IDbCommand cmd)
    {
        //the execution stuff
        //notify all listeners
        foreach (var listener in this.Listeners) {
            listener.Notify (); //optionally send along some data
        }
        return null;
    }
}
```

There are other ways to do this in C# – namely using

virtual methods that inheriting classes can implement directly.

A variation on this pattern is the “hook” which you see in Node frameworks like Sequelize or Feathers (a real-time web framework). Ruby on Rails has hooks for **ActiveRecord** as well. Hooks are explicit observers that you can override on a per-instance basis, such as **onSave**, **onDelete** and so on. The whole idea here is that you’re watching a class instance (observing it, if you will) and reacting to things.

Observers are incredibly useful – almost as much as they are annoying when you’re trying to track down bugs. The question comes down to this: would you rather put your logic in a single “service” class, or orchestrate a set of events?

Consider a simple case of blog post. When a comment comes in we need to do a few things:

- Run some light moderation, checking for things we’d rather not have in our post comments
- Save the comment to the database
- Notify the blog owner that there’s a new comment

There are likely more, but let’s go with these for now. We could implement this logic using a simple service class, which we could call **Feedback**. That class could have a method called **newComment** which does the things we need done. We can test that class and verify everything works.

We could also implement these requirements using observers. Our **PostObserver** could see that a new comment has been added with a status of “pending”, which means it needs to be moderated. When the status changes to “moderated” the **onModerated** hook fires which means we can save the comment in the database. This, in turn, fires the **newComment** hook which executes a notification routine that sends out an email.

I've done both things both ways and you can see a slightly more complex example here, where I handled deep callbacks using Node's **EventEmitter**. This was in the good old days before promises and `async/await`.

I like using events but, as I hope you can see, they can quickly cause confusion in a larger application.

The State Pattern changes an object's behavior based on some internal state. Often this is done by creating formalized state classes. You often hear an implementation of this pattern called a "State Machine", which is a fascinatingly complex way of handling process flow.

For instance, you might have an Order that transitions from pending to paid and then to fulfilled. With each transition, the Order might have specific rules in place such as:

- Refunds are only possible if an order is in the "paid" or fulfilled statuses.
- If an order is in the "refunded" state it can't be changed.
- If an order is "pending" it can only transition to "paid" or "voided"

It's easy to see how the State pattern can help orchestrate these concerns.

To continue with our running example, we want to know what current state our **QueryCommand** is in – if it's new, succeeded, or failed. Let's create some classes that tell us this. We'll start with the **QueryCommand** and a base class for the state:

```

public class QueryCommand
{
    public QueryState State { get; set; }
    //...

    public QueryCommand ()
    {
        this.State = new NotExecutedState (
            "Query has not been run"
        );
    }
    //pass execution off to the state bits
    public T Execute<T> ()
    {
        return this.State.Execute<T> (this);
    }
}

```

Here we've added a class to explicitly handle the idea of a **QueryState** and set it as a property on our **QueryCommand**. The initial state, when the **QueryCommand** is first created is **NotExecutedState**, which is another class we'll need to create.

```

//our base class
public abstract class QueryState {
    protected string Message { get; set; }
    public QueryState (string message) {
        this.Message = message;
    }
    public abstract T Execute<T> (QueryCommand cmd);
}
//an explicit state for not executed
public class NotExecutedState : QueryState {
    public NotExecutedState (string message) : base (message) { }
    public override T Execute<T> (QueryCommand cmd) {
        try {
            //run query execution... and if it works
            var results = cmd.Execute(); //pretend it goes off
            cmd.State = new SuccessState (results, "Query executed successfully");
        } catch (Exception x) {
            //on error
            cmd.State = new FailState (x.Message);
        }
        //return query results
    }
}

```

As you can see, the execution of a query is only available in the `NotExecutedState`, which makes perfect sense. If the execution goes off without a problem, the `QueryState` is transitioned to `SuccessState` which is something we'll need to create. Otherwise we'll set a `FailureState` with an error message.

Something to notice as well is that results are only available in a `SuccessState`, which again makes good sense.

Great. Let's create a **FailState**, for when the query fails and a **SuccessState** for when execution goes off just fine:

```

public class FailState : QueryState {
    public FailState (string message) : base (message) { }
    public override T Execute<T> (QueryCommand cmd) {
        throw new InvalidOperationException (
            "This query already failed execution"
        );
    }
}

public class SuccessState : QueryState {
    public IList<T> Results {get; set;}
    public SuccessState (IList<T> results, string message) : base (message) {
        //assign the results
        this.Results = results;
    }
    public override T Execute<T> (QueryCommand cmd) {
        throw new InvalidOperationException (
            "This query already executed successfully"
        );
    }
}

```

This is obviously simplified but I hope you get the idea. We're handling transitions from one state to the next based on the results of our command behavior and we're being explicit about it. That clarity can be very helpful, but it can also be incredible overkill. We have a lot of code here to convey a simple process: the execution of a query.

With an Order, however, the idea of a *State Machine* becomes a little more understandable. With a State Machine, we transition from one state to the next in a very prescribed way. If you're thinking "hey wait, I read about Finite State Machines a few chapters ago..." and yes! That's exactly what I'm talking about!

The Strategy Pattern is a way to encapsulate "doing a thing" and applying that thing as needed. Code is the easiest way to explain this pattern, as it's quite simple and useful.

Our document query capability is working well, but it turns out that SQL Server has excellent support for XML, and some users have asked that we support that along with JSON storage.

We can do this using the Strategy Pattern. We'll start by defining our **GroovyQuery** class and the interfaces we need:

```
public interface IDocumentQueryable
{
    T Get<T> ();
    T Save<T> ();
    IList<T> Fetch<T> ();
}
public abstract class GroovyQuery
{
    //groovy interface
    public abstract T GetDocument<T> ();
    public abstract T SaveDocument<T> ();
    public abstract IList<T> FetchDocuments<T> ();

    public IDocumentQueryable Documents;
    //etc
}
```

We're going to change things a bit now by using a storage *strategy* which we'll describe using an interface:

```
public interface IStorageStrategy
{
    T Hydrate<T> (string document);
    string Dehydrate<T> (T item);
}
```

When we store something we either Hydrate it (into an object) or Dehydrate it (into storable XML or JSON). Now we just need to implement them:

```

public class JsonStorageStrategy : IStorageStrategy {
    public string Dehydrate<T> (T item) {
        //turn the object into JSON, return the JSON
    }
    public T Hydrate<T> (string json) {
        //resolve from JSON
    }
}
public class XmlStorageStrategy : IStorageStrategy {
    public string Dehydrate<T> (T item) {
        //turn the object into XML
    }
    public T Hydrate<T> (string xml) {
        //resolve from XML
    }
}

```

Finally we implement the **DocumentStore** with the needed strategies:

```

public class DocumentStore {
    GroovyQuery _adapter;
    IStorageStrategy _parser;
    public DocumentStore (GroovyQuery adapter) {
        _adapter = adapter;
        _parser = new JsonStorageStrategy ();
    }
    public DocumentStore (GroovyQuery adapter, IStorageStrategy parser) {
        _adapter = adapter;
        _parser = parser;
    }
    public T Save<T> (T item) {
        var document = _parser.Dehydrate (item);
        //parse and save the object
    }
    public T Get<T> () {
        //pull the record, dehydrate
        //get the results
        return _parser.Hydrate<T> (result);
    }
    //...
}

```

We’ve “decoupled”, sort of, the storage format from our **DocumentStore** so our users can store XML or JSON. But how and where do you tell the **DocumentStore** which

strategy to use? An ENV variable? Configuration file somewhere?

In the .NET world this would likely be done using Inversion of Control as the application starts up and things are configured. We're going to talk about that and a few other things in the following chapter. What we're focusing on here is just the strategy part – the implementation is something we'll deal with later.

In The Real World...

Many of you will likely notice that I left a few patterns out of the above list – namely the Visitor Pattern, Memento, Template, etc. These are useful patterns to know about, but their use is rare.

For instance, the Visitor Pattern – it's useful if you're parsing tree structures (like Expression Trees in C#) but in everyday code, this is rare. For me, at least.

Also: as you implement patterning as we've done here, the code you write tends to become more generalized and you end up writing a lot more of it just to do a simple operation. This is not what these patterns are for.

A design pattern should make things simpler. If you implement one, have a look at your code before and after, and see if it makes more sense or less. Think about the people inheriting your code in a year, two years and five years. I'll be honest with you and say that if I inherited code that implemented a strategy pattern (or Mediator etc.) I would want to know why and then I would likely see if there was a simpler, more direct way of doing the thing needed.

For instance: our Strategy pattern example is useful because in the future we might want to store documents in an entirely different format – perhaps YAML! I know I run the risk of creating a straw-man argument here, but I do

think this perspective is valuable: *at what price do we implement this pattern?*

Let's say we implemented a 20-line switch statement to handle the different storage formats we'd be using. Or maybe we have explicit methods in place like `SaveYaml()` and `GetYaml()`. In terms of pure lines of code written – I would wager we'd be pretty close. Our `DocumentStore` would indeed grow, but we would lose all the Strategy pattern stuff.

The question is: *which one is easier to hold in your head?* What about your teams' heads? My point is this: implementing a pattern doesn't automatically mean your code is correct and maintainable, though they often do help with that.

OBJECT-ORIENTED DESIGN PRINCIPLES

SOLID, YAGNI, SEPARATION OF CONCERNS
AND MORE

As you build applications using the patterns we learned in the previous chapter, you begin to see some common side effects or, as they are otherwise known: “code smells”. These monikers are ambiguous and don’t readily communicate what the actual problems might be. Not like the term “carcinogen” or “repellant”.

Understanding what these terms mean as well as why people say them can take years. Often you come to understand that many people who utter these phrases are just repeating what’s been said to them, usually misunderstanding the reasons why and completely missing the point altogether.

There’s a term for this kind of behavior and it’s called “cargo culting”:

A **cargo cult** is a millenarian belief system in which adherents perform rituals which they believe will cause a more technologically advanced society to deliver goods. These cults were first described in Melanesia in the wake

of contact with allied military forces during the Second World War.

Programmers do this kind of thing all the time, believing if they use a certain pattern, toolset or framework then their application will *just work* where the term “work” means it will be scalable, easily maintainable and solve the client’s problem.

Given this line of thinking, the absence of a pattern or process means that said application will *not* work correctly.

All of that said, a more centered mind can find the process of implementing a design pattern to be very enlightening. The Strategy, Adapter, Mediator and Bridge Patterns, for instance, can lead you to think critically about how much Class A knows about Class B, a term we call “coupling”. Perhaps the Façade pattern will push you to consider how well Class A actually defines the properties and behavior of an A, whatever an “A” might be. This is something we call *cohesion*.

These two ideas: Software Patterns and Design Principles, work together. One does not precipitate the other in any meaningful way. Design Principles help us think about *what* we’re coding and *why*. Software patterns help us assemble that code into a proper structure.

Obviously, this is not a small topic. In this chapter we’ll discover the key principles you should understand, who came up with them, and why. These amount to philosophy dreamed up by people who’ve done this sort of thing for a very long time. Some are a bit older, some are newer – all are relevant.

The code for this chapter is, once again, shown in screenshots. If you’d like to play along or have access to the code, you can view it or download it from our GitHub repo.

Coupling and Cohesion

You've likely heard these terms before, they're thrown around a lot and have straightforward definitions:

- **Cohesion** applies to how well you've thought out the concepts (and concerns, for that matter) of your application. In other words: how related are the functions of each module or class? You put your **Membership** code into a **Membership** module and your **User** code in a **User** model. The functionality here is cohesive (meaning the ideas bound together logically).
- **Coupling** is the opposite of cohesion. When you couple two or more things, their separate notion becomes one. In our code above we had an example where **Membership** created a **newUser** during the registration process. This coupled **Membership** to **User**. If we moved/renamed/got rid of the notion of a **User** we'd have an error in our **Membership** code. This is tight coupling.

You want high cohesion, low coupling. Your classes and modules should make sense for isolating ideas, and not rely on each other to exist. This is the goal, at least.

These ideas were invented in the 60s by Larry Constantine and later formalized in a white paper called Structured Design (Yourdon and Constantine, 1979):

For most of the computer systems ever developed, the structure was not methodically laid out in advance – it

just happened. The total collection of pieces and their interfaces with each other typically have not been planned systematically. Structured design, therefore, answers questions that have never been raised in many data processing organizations.

It's a fascinating and easy read, and I highly suggest it.

These ideas are *foundational* to writing clear, understandable code and should be in your mind *constantly* as you go about your programming day. A tightly coupled class can be a nightmare to change, especially when the things it relies upon change, which they often will. When one class changes and another one breaks due to that change – that's "tight coupling" and it means you need to fix something pronto.

Cohesion, on the other hand, is more of a "guiding light" if you will. At some point in your career you've probably (or will do so at some point) had a situation where you were coding a class of some kind thinking something along these lines: "hang on... why is my Post class handling the moderation comments?"

Indeed, a Post is a blob of text assembled in a given way. A User with the role of Moderator is probably a better idea – a more *cohesive* an idea – to handle the task.

Breaking this down even further one could say that this is the act of programming in an object-oriented language distilled to its essence. Ideas resolve into aspects of the real world and we hope that those aspects are what are program is indeed trying to convey. If they are, we're happy programmers. If they aren't... we've made a big mess. The trick is to keep asking ourselves "is this really what an A (in my Class A) would do?" and, correspondingly, "does Class A really need Class B to exist?"

Separation of Concerns

Separation of Concerns is about slicing up aspects of your application, so they don't overlap. These are typically thought of (by developers) as horizontal concerns (they apply to the application as a whole): such as user interface, database, and business logic. The term can equally (and confusingly) be applied to more abstract ideas, such as authentication, logging and cryptography.

Finally, there are vertical concerns, which deal with more business-focused functionality such as Content Management, Reporting, and Membership.

The whole idea of a “concern” in programming has been convoluted, diluted and elocuted to the point of meaninglessness. It reminds me of implementing an API where one of the variables has the name **Context**. It means *nothing* and could represent just about any idea the developer had at the time.

I had a discussion once with a developer who suggested I separate SQL from my data access code so I can have a “cleaner separation of concerns”.

Another time it was suggested to me that dividing my .NET code into separate library projects (basically moving files around on disk) was a great way to *separate the concerns* of my application, instead of having all those files together in one place.

Ruby on Rails (which is responsible for the spread of the term) famously suggested that the Model View Controller approach they used was a great “separation of concerns” as it decoupled data access and business logic from HTML. The reality is it did **exactly the opposite**. A Rails view is HTML strewn with artifacts (which are Models) created in a Controller that deal directly with data access.

There is no separation there. Ruby on Rails version 3.0 tried to get there with more generic implementations (so

called Railties), but this ended slowing everything down and making a mess in the name of architectural purity. I'm not saying it was *wrong*, I'm saying it was slow and lost a lot of compelling reasons to use Rails in the first place.

The term "Separation of Concerns" has become a marketing catchphrase that has lost its original meaning. There's power in the original idea and it's an expansion on the notion of *coupling*. Instead of decreasing the coupling of your classes, however, Separation of Concerns forces you to consider decoupling your *application*.

The best example of this that I've seen is the Django web framework. It pushes the idea of building smaller apps that work together, rather than one large app. Microservices, to use a loaded term, tries to do the same thing but has been cargo-culted into a punchline. We'll talk more about microservices in a later chapter.

For now, empty your mind and let's take a trip to the past...

The origin of the term comes from this quote from one of my favorite computer science people, Edsger W. Dijkstra:

Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so: why, the program is desirable. But nothing is gained — on the contrary! —

by tackling these various aspects simultaneously. It is what I sometimes have called “the separation of concerns”, which, even if not perfectly possible, is yet the only available technique for effective ordering of one’s thoughts, that I know of. This is what I mean by “focusing one’s attention upon some aspect”: it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect’s point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.

Every application we build is composed of a vertical subset of processes and rules that try to solve a business need. An eCommerce application will have a sales aspect, a membership aspect, accounting, and fulfillment. These are concerns of the application.

Can we apply this type of thinking to more horizontal ideas? In other words, can we study the notion of logging? Or data access? I’m sure friends of mine would argue that I have, indeed, done the latter many times! Studying these horizontal aspects of our application might make the application better, but I don’t think it will make it more correct.

Now this is where we come up against the weight of history and a little trick that every politician knows: **If you say something long enough it become true**. I think it’s the same with the phrase “separation of concerns”. It’s reminiscent of the phrase “I could care less” or “hone in on”. These phrases make no sense at all, but for some reason popular American English vernacular has twisted them to mean something and, as time goes on, they get adopted.

I think the same is true with Separation of Concerns. It’s a catchall phrase which means “I’m trying to do the right thing”, whatever that thing may be. Perhaps it’s an effort at

file organization or using one of Fowler's enterprise patterns (which we won't be discussing in this book) to abstract away a part of your application ... at this point *it doesn't matter*.

So: when in conversation and someone invokes this trite little phrase, perhaps ask them for some detail. After a few years of doing this, you should have some fun tales to share.

YAGNI and DRY

I remember when I started learning Ruby. I loved the simplicity of the language as well as its dynamic design which, I know, many people dislike. You had to have some rigor and much care when building programs with Ruby because you didn't have a compiler and static type checking.

This was freeing, and it was also a little scary.

Part of this rigor was learning a new set of jargon. YAGNI (You Aint Gonna Need It) and DRY (Don't Repeat Yourself) – neither of which came from the Ruby community – started becoming more popular precisely because the practices you needed to adopt to write good Ruby code leaned squarely on you, as developer, rather than your tooling.

More (and better) testing replaced compiler checks. Test-driven Development (TDD, which we'll get to in a later chapter) helped in this regard as well – forcing you to justify the code you needed to write with a set of tests. In other words: if a test didn't mandate some code's existence, you didn't need that code.

Don't Repeat Yourself is something that most developers understand. Duplicated code is difficult to maintain. With large applications, this is almost impossible – but it is an important idea to keep a focus on.

If you have an application with 300 classes and 50,000 lines of code, you're bound to have duplication. The trick is

to spot it and, hopefully, to simplify your future by abstracting it in some way.

In January 2016 Sandi Metz wrote a great article about the perils of focusing too much on DRY. The punchline being:

... duplication is far cheaper than the wrong abstraction... prefer duplication over the wrong abstraction...

The article sprang from a talk she gave at RailsConf in 2014.

The main idea is that you don't want to create abstractions solely for the sake of avoiding repetition; it needs to fit your overall approach. If you can't fit it, just let the duplication be.

Tell, Don't Ask

Another Rubyism that I quite like came from Ruby's inspiration: Smalltalk. Whenever you invoke a method on a Ruby class you send it a message. You tell that instance that you need it to do something, or that you need some data back of some kind.

If you ask an object instance a question, then you'll need to know something about that object or its state, which breaks the notion of encapsulation.

If you think back to our **GroovyQuery** from a previous chapter, imagine this as our API:

```

using System;
using System.Data;

public class GroovyQuery
{
    public bool IsCommandValid (IDbCommand cmd)
    {
        //logic
    }
    public bool IsConnectionAvailable ()
    {
        //check connection pool to see if one is ready
    }
    public IDataReader Execute (IDbCommand cmd)
    {
        //execution
    }
}

```

To use this API effectively I would need to ask two questions and finally get around to telling the class what to do (**Execute**). In short: I need to know way more about the API than is needed.

A better way to do this is by moving a few things around:

```

public class GroovyQuery2 //Telling
{
    bool CommandIsValid (IDbCommand cmd)
    {
        //logic
    }
    bool ConnectionIsAvailable ()
    {
        //check connection pool to see if one is ready
    }
    public IDataReader Execute (IDbCommand cmd)
    {
        var commandIsValid = CommandIsValid (cmd);

        if (ConnectionIsAvailable () && commandIsValid) {
            //execution
        } else {
            throw new InvalidOperationException ("Can't run this query");
        }
    }
}

```

The responsibility for deciding whether the query can run is now within **GroovyQuery**, which is where it should be.

Law Of Demeter (or: Principle of Least Knowledge)

The Law of Demeter (LoD, or “Deep Dotting”) is an offshoot of loose coupling. In short: you shouldn’t have to “reach through” one object to get to another. This can be further nuanced to mean you shouldn’t have to reach deeply into one object to do the thing you need to do.

Let’s examine both.

Our **Membership** system is working well, but some users aren’t behaving themselves so we need to give them a bit of a timeout. With our first go we decide to drop a **Suspend** method on **User** because we’re telling them they’re suspended:

```
public class DB {
    public User GetUser (int id) {
        //call to the DB, getting record
        //returning an empty user for now
        return new User ();
    }
}

public class User {
    public String Status { get; set; }
    public void Suspend() {
        this.Status = "suspended";
    }
}

public class Membership {
    DB _db;
    public Membership () {
        _db = new DB ();
    }
    public User GetUser (int id) {
        //get the user
        return _db.GetUser (id);
    }
}
```

To suspend a user we need to access them from the Membership module and then suspend them:

```
var membership = new Membership();  
membership.GetUser(1).Suspend();
```

This is a violation of LoD. We had to reach through **Membership** to get to the **User**. You might be wondering ... so what?

It's a subtle point, sure, but the more you think on it the more you realize how you're muddying the principles we've been reading about.

In essence: *we've punched a hole in our membership abstraction* by dividing the responsibility for changing the user between two different classes. Cohesion is breaking down and coupling is going up.

It doesn't make sense to involve **Membership** at all here, except for the fact that we need to get at the **User**. So let's make a choice, and it's a simple one: **Membership** has the responsibility of adding and retrieving users (aka changing them) so let's have it update the user's status as well:


```

public class DB {
    public User GetUser (int id) {
        //call to the DB, getting record
        return new User ();
    }
    public void Save (object item) {
        //save to DB
    }
}

public class User {
    public String Status { get; set; }
}

public class Membership {
    DB _db;
    public Membership () {
        _db = new DB ();
    }
    public User GetUser (int id) {
        //get the user
        return _db.GetUser (id);
    }

    public void SuspendUser (int id) {
        var user = this.GetUser (id);
        user.Status = "suspended";
        _db.Save (user);
    }
}

```

Note: we're being pushed in a particular direction as we consider these design principles. Can you see what it is? We're removing "external", for lack of better words, functionality from our User class. It can now stand on its own with no external dependencies. Our DB, on the other hand, needs to know what a User is in order to work. This is possibly more coupling than we want and we'll discuss it in the Architecture chapter.

Some developers will focus on "dot counting", claiming that the use of too many dots is, all by itself, a violation of LoD. Sometimes it is, sometimes not.

Consider this API:

```

var liTag = new Html.Helpers.HtmlTags.Lists.ULTag.LiTag();

```

This API is kind of ridiculous, I must say. However, it's an organizational choice and not necessarily a violation of LoD. This could be horrible namespacing for all we know! Or it could be a lack of imagination. We don't know the inner workings of the Html helper library (and trust me, you don't want to), so it's not exactly accurate to call for a violation just by looking at dots.

I was reading my friend Phil Haack's blog while researching this subject, and he had a great quote from Martin Fowler:

I'd prefer it to be called the Occasionally Useful Suggestion of Demeter.

I hate to leave you with vagary, but hopefully you can see how "deep dotting" and LoD aren't always the same thing.

Dependency Injection

One way to loosen up your code is to send in the dependencies that a class needs through its constructor. The best way to see this is with some code.

Our **Membership** class is using the database to retrieve and save a **User**:

```

public class Membership{
    DB _db;
    public Membership(){
        _db = new DB();
    }
    public User GetUser(int id){
        //get the user
        return _db.GetUser(id);
    }
    public void SuspendUser(int id){
        var user = this.GetUser(id);
        user.Status = "suspended";
        _db.Save(user); //coupling
    }
}

```

This couples the **Membership** class to the **DB** class which is responsible for data interactions. We've read *The Imposter's Handbook*, so we know that coupling is bad – but how can we change this?

The simple answer is to inject the dependency through the constructor rather than to invoke it in place:

```

public class Membership {
    DB _db;
    public Membership (DB db) {
        _db = db;
    }
    public User GetUser (int id) {
        //get the user
        return _db.GetUser (id);
    }
    public void SuspendUser (int id) {
        var user = this.GetUser (id);
        user.Status = "suspended";
        _db.Save (user); //Coupling
    }
}

```

Now our class doesn't need to know how to instantiate **DB**, which is one step in the right direction. There's still a bit too much coupling, however as our **Membership** class cannot be used unless a **DB** instance is passed in.

Let's see how we can loosen this up a bit more.

Interface-based Programming

Many languages support the idea of interfacing with an ability, rather than a type itself. With C#, Java and now TypeScript these are called *Interfaces*. With languages such as Swift and Elixir this is done with Protocols. For our purposes I'll use interfaces, so translate as you need.

The goal of working with interfaces is to describe an ability of your application. In our case all we care about is that we can retrieve and save a record from a data store. It's important to keep this interface light because doing so will actively increase cohesion and drive down coupling:

```
public interface IDataStore {
    public void Save<T>(T item);
    public T Get<T>(int id);
    public IList<T> Fetch<T>();
}
```

This is a good start. We can now use our new interface:

```
public class Membership {
    IDataStore _db;
    public Membership (IDataStore db) {
        _db = db;
    }
    public User GetUser (int id) {
        //get the user
        return _db.Get<User> (id);
    }
    public void SuspendUser (int id){
        var user = this.GetUser (id);
        user.Status = "suspended";
        _db.Save (user);
    }
}
```

Much better. We still have coupling to the notion of an **IDataStore**, but it's unavoidable at this point (unless we want to work directly with eventing, but that's probably overkill). We can now implement an **IDataStore** to do all kinds of things for us, such as:

- Store data in a relational system
- Store data in a NoSQL system
- Store data directly in memory for testing purposes

Using interfaces like this is a cornerstone of object-oriented programming. Injecting them, as we're doing here, is a great way to keep your code isolated.

It does come at a price.

Inversion of Control

As you build out your application, paying attention to interfaces and dependency injection, you will start to see the number of dependencies for a given class begin to spiral a bit out of control. The best way to see this is with some code.

In the real world, our **Membership** class will probably need quite a few external dependencies:

- A hashing library for password storage
- An email library for sending a new user a note
- A rules module for accepting new users
- A logger module for logging

This means our constructor is going to grow:

```

public class Membership{
    IDataStore _db;
    IEmailer _email;
    ICrypto _crypto;
    ILogger _logger;
    IRulesEngine _rules;
    public Membership(IDataStore db,
        IEmailer email,
        ICrypto crypto,
        ILogger logger,
        IRulesEngine rules){
        _db = db;
        _email = email;
        _crypto = crypto;
        _logger = logger;
        _rules = rules;
    }
}

```

Yikes. Registering a user isn't much fun either. Here's the **Register** method on the same **Membership** class:

```

public void Register(IRegisterable user){
    //validations etc
    if(_rules.CanRegister(user){
        user.Status="Registered";
        user.HashedPassword = _crypto.HashPassword(user.Password);
        _db.Save(user);
        _email.SendWelcome(user);
        _logger.Info("New user added: " + user.Email);
        return user;
    });
}

```

This is nuts. Every time we want to use **Membership** we'll need to create instances of its dependencies which, themselves, likely have dependencies of their own we'll need to create (and then inject). This is simply sweeping the

dependency coupling somewhere else.

This is where Inversion of Control comes in. With Inversion of Control you have a separate mechanism (called a “container”) which is responsible for creating and injecting all the dependencies you need and then giving those injected objects to you when you need it.

Here is some pseudocode for an IoC container modeled after my friend Nate Kohari’s excellent Ninject Project:

```
//our app start
public void Main(){

    Container container = new Container();
    container.Bind<IMembershipStore>().To<PostgreSQLAdapter>();
    container.Bind<IEmailSender>().To<MailgunSender>();
    container.Bind<ILogger>().To<Log4Net>();
    container.Bind<ICrypto>().To<SuperCryptoThingy>();
    container.Bind<IMembership>().To<Membership>();

    //get an instance of Membership
    var membership = container.Get<IMembership>().InSingletonScope();

}
```

We have our interfaces mapped to concrete implementations in a single place. If we ever need to change anything, we just change it here.

When we need an instance, we simply need to access our container, which needs to be global to our application¹. The container then orchestrates the instantiation of the classes we want. A bonus to this is that we can set a “scope” on the object. For instance, in the example above I’m setting the lifecycle to a Singleton.

You can do many other things with Inversion of Control containers – and they are quite useful.

You might be getting the sense that we’re creating a bit of a “meta” programming system here, where object instantia-

tion is removed from the language constructs themselves and into this separate...mechanism of our own creation.

This is where we start getting subjective. There are quite a few developers out there who see patterns like the above as flaws in the language or, more broadly, as flaws in object-oriented programming itself. This is a great quote from Lawrence Krubner in his essay *Object-oriented Programming is a Disaster and Must End*:

I have seen hyper-intelligent people waste countless hours discussing how to wire together a system of Dependency Injection that will allow us to instantiate our objects correctly. This, to me, is the great sadness of OOP: so many brilliant minds have been wasted on a useless dogma that inflicts much pain, for no benefit.

Now that we understand a bit more about dependency injection and inversion of control containers – do you think you’ll waste “countless hours”? To be honest: *yes, I have*. But it was my fault. That last bit doesn’t make me feel any better and is the chorus sung by most framework adherents when you criticize their framework of choice: “PEBCAK” or *Problem Exists Between Chair and Keyboard*. It’s always the user’s fault innit?

Keeping your containers happy and working properly in an IoC sense is not as simple as it seems. As your application grows and becomes more complex, it becomes easier to find yourself creating circular dependencies. For instance: we might decide to create an **ILogger** implementation that saves logs to a database. We decide to reuse our **IDataStore**, which requires an instance of **ILogger** which requires an instance of **IDataStore**...

These problems, as you might be sensing, typically have to do with application design rather than object-oriented programming. Which seems to be a recurring problem in our industry.

If a language, platform or framework leads you down a snarled path of bad design, it's usually your fault. Or is it?

I'd like to leave this chapter with a great quote from my friend Gary Bernhardt, which he offered during his amazing talk *The Birth and Death of JavaScript*:

The behavior that you see a tool being used for is a behavior that tool encourages.

It's easy to dismiss recurring structural problems as ignorance on the programmer's part. If the same problem occurs throughout the development community, however, is it really a problem of ignorance?

I don't have an answer.

SOLID

In object-oriented programming circles it's almost impossible to escape Martin/Feathers/Meyer's SOLID principles, which are:

- **Single Responsibility Principle (SRP):** a class should do one thing and have only one reason to change.
- **Open/Closed Principle:** a class should be open for extension but closed for modification.
- **Liskov Substitution Principle:** a class should be able to be substituted with its subclass.

- **Interface Segregation Principle:** small, role-based interfaces are better than a massive object.
- **Dependency Inversion Principle:** higher level classes shouldn't depend on lower-level ones; abstractions shouldn't depend on details - rather details should depend on abstractions..
- SOLID has been around for many years and is considered to be something you just don't argue with. Which is why I'm going to start this chapter doing just that.

I've struggled with SOLID. On the face of it, SOLID makes sense doesn't it? Smaller classes and clear thinking when creating your domain objects - what's wrong with that? Nothing, really, especially if you want to build a career as a coding guru - writing books and booking speaking engagements based on an acronym which is otherwise *common sense*.

That's a strong opinion, but it's an important one. Let me explain.

I used to work in rather large development groups. Nothing was more infuriating than a condescending grunt from a code reviewer who would point at some code saying "that violates SOLID". As if that makes what you've written *wrong*.

This is my main critique of SOLID: *it gives people a tower to climb and a robe to hide behind*. Your code is wrong because it violates a doctrine that was created to propel a career, not your project.

This is where the "science" part of "computer science" needs to come into play: *How you've built something needs to be considered separately from how it works*. It's easy to read that sentence and think "oh wow Rob is suggesting developers become sloppy" - but I'd like for you to consider the opposite of that opinion.

How many times have you looked over a nicely SOLIDified codebase and thought "this is pure crap". An explosion of classes, interfaces and abstractions all thrown together to creat a todo list. Ceremony for the sake of ceremony and escaping the dreaded "this class violates Liskov..."

There's a good idea behind SOLID, but the idea should be the focus. The inspiration to keep things as simple as you can so your future self (or another developer who has to support what you've created) doesn't get overwhelmed and confused. This has a practical aspect as well as egotistical: *if they can't figure out what you've done, they'll probably throw it away*. Just like you've done many times with other people's code.

I like how Dan North puts it: *just write simpler code*. This is a powerful idea because "simple" doesn't mean "wrong" - in fact it means the opposite. Think about world-class athletes - how easy do they make their sport look? The best interface designs and hardware form factors all strive for the same thing: *blissful, wonderful simplicity*.

I invite you to follow the link above and have a look at Dan North's presentation on "Why every element of SOLID is wrong". It's good stuff, if only to make sure we don't fall into the trap of adhering to doctrine over good design.

Dan, by the way, is the creator of Behavior-driven Design and is a rather well-known figure in the development world. This talk (which I was at) was a lightning talk and was intended to be for laughs as much as for thought - but to me it was the best talk I had seen that entire week.

A class should have a single responsibility to your application or, as Martin puts it, *a single reason to change*. This sounds simple, but it's a little tricky to grasp.

Most applications have a class for a **User**:

```
public class User
{
    public string Name { get; set; }
    public string Email { get; set; }
    public string Status { get; set; }
    public User ()
    {
        this.Name = "Guest";
        this.Status = "Anonymous";
    }
}
```

This class changes when the data about the **User** changes. But how does that information actually change?

Let's say we want to register the user into our system. We could do something like this:

```
public class User {
    public string Name {get;set;}
    public string Email {get;set;}
    public string Status {get;set;}

    public User() {
        this.Name = "Guest";
        this.Status = "Anonymous";
    }

    public void Register(string name, string email) {
        this.Name = name;
        this.Email = email;
        this.Status = "Registered";
        //save to the DB or something else
    }
}
```

This class is now doing two things: describing a **User** based on some data and registering a user into the system. We can keep with SRP if we move the registration responsibility off to another class:

```

public class User {
    public string Name { get; set; }
    public string Email { get; set; }
    public string Status { get; set; }
    public User ()
    {
        this.Name = "Guest";
        this.Status = "Anonymous";
    }
}

public class Membership {
    public User Register (string name, string email)
    {
        //validations etc
        var newUser = new User { Name = name,
                                Email = email,
                                Status = "Registered" };

        //save to the DB or something else
        return newUser;
    }
}

```

Open/Closed

This principle may seem obvious to you, but at the time it was created and refined (late 80s, early 90s) it addressed a real problem.

Open/Closed says that a class or module should be open for extension, closed for modification. Or, put another way: let people override/extend your code without needing to modify it.

Let's say we're working in Node, and we need to install a module from NPM. We run our npm install command and a **node_modules** directory appears with our module.

It turns out that we find a bug in one of the methods! We could go and fix the bug directly inside of **node_modules**, but that would violate Open/Closed! The good news is that the developer left the module extensible, so we can go in and

override the bug with a fix. This module was open for our extension but closed for our modification.

With object-oriented languages, this kind of thing is solvable if you allow users of your classes and modules to inherit and extend key bits of functionality that you're providing.

Liskov Substitution

Liskov is a subtle principle but can catch some very serious bugs that creep into your application. The principle has to do with inheritance and how inheriting objects behave. It says:

if S is a subtype of T, then objects of type T may be replaced with objects of type S

Let's say we have a User and Administrator class:

```
public class User{
    //...
}
public class Administrator : User{
    //...
}
```

My program should work if I pass **Administrator** to any routine that expects a **User**.

The classic example of breaking LSP is the **Square** and **Rectangle** analogy:

```

public class Rectangle {
    int _height;
    int _width;

    public virtual void SetHeight (int height) {
        _height = height;
    }
    public virtual void SetWidth (int width) {
        _width = width;
    }
}
public class Square : Rectangle {

    public override void SetHeight (int height) {
        this.SetHeight (height);
        this.SetWidth (height);
    }
    public override void SetWidth (int width){
        this.SetHeight (width);
        this.SetWidth (width);
    }
}

```

This code is obviously confusing in that we're coding our way around a problem in our implementation. While a square, in reality, is a rectangle (if we're talking math) – implementing it as a **Rectangle** in our code causes us to do some weird things.

Moreover, passing a **Square** around as if it were a **Rectangle** could cause some very strange things to happen in our video game code later on.

Interface Segregation

The Interface Segregation Principle is all about targeted, simple API creation so code is easy to use and implement. Rather than create a small set of large interfaces, favor a larger set of smaller, more generic interfaces.

Let's think about registering our **User** again:

```

public class User {
    public string Name { get; set; }
    public string Email { get; set; }
    public string Status { get; set; }
    public User () {
        this.Name = "Guest";
        this.Status = "Anonymous";
    }
}

public class Membership {
    public User Register (string name, string email) {
        //validations etc
        var newUser = new User { Name = name, Email = email, Status = "Registered" };
        //save to the DB or something else
        return newUser;
    }
}

```

This works fine, but the notion of a **User** is bound to our **Membership** class. We could solve this with an interface (something like **IUser**), or we could lean on ISP and focus on what's really needed. Here we can use **IRegisterable** with our **User** class:

```

public interface IRegisterable
{
    string Name { get; set; }
    string Email { get; set; }
    string Status { get; set; }
}

public class User : IRegisterable
{
    public string Name { get; set; }
    public string Email { get; set; }
    public string Status { get; set; }
    public User ()
    {
        this.Name = "Guest";
        this.Status = "Anonymous";
    }
}

```

Now our **Membership** class is a bit cleaner:


```
public class Membership {
    public IRegisterable Register (IRegisterable user) {
        //validations etc
        user.Status = "Registered";
        //save to the DB or something else
        return user;
    }
}
```

We could extend this notion further with **IAuthenticatable** as well, if we want. Abstracting your code like this helps hide implementation details – which means making changes in the future becomes a lot easier.

Dependency Inversion

Dependency Inversion is all about loosening up the relationship between classes and modules in your code. The definition is a bit wonky:

- High-level modules should not depend on low-level modules. Both should depend on abstractions
- Abstractions should not depend upon details. Details should depend upon abstractions

Our **Membership** module is a high-level module, and we need to depend on some low-level modules in order for things to work.

For instance: we need to save our **IRegisterable** object to the database. We could go the route of creating a new database connection right in our **Register** code, if we want:

```

public interface IRegisterable {
    string Name { get; set; }
    string Email { get; set; }
    string Status { get; set; }
}
public class User : IRegisterable {
    //...implement interface
}
public class Membership {
    public IRegisterable Register (IRegisterable user) {
        //validations etc
        user.Status = "Registered";
        //save to the DB or something else
        var db = new PostgreSQLAdapter ();
        db.Save (user);
        return user;
    }
}

```

But this is coupling or binding our **Membership** class directly to our **PostgreSQLAdapter**. This will solve our problem now, but in the future, we will likely come across some problems:

- We might want to switch data access tools later on, maybe moving from our homespun routines to an ORM
 - We might move to a document database, or something hosted (like Amazon's Dynamo DB)
 - Our adapter might change the way it's constructed – perhaps moving to a Factory pattern or the like.
- Or maybe we'll have a Mediator...

In short: our **Membership** class knows way too much about the construction and execution of our adapter – it's too tightly *coupled*. Let's invert this dependency, shall we?

```

public interface IMembershipStore {
    void Save (object item);
}

public class PostgreSQLAdapter : IMembershipStore {
    //implement interface
}

public class Membership {
    IMembershipStore _store;
    public Membership (IMembershipStore store) {
        _store = store;
    }
    public IRegisterable Register (IRegisterable user) {
        //validations etc
        return _store.Save (user);
    }
}

```

Now we're free to change our storage approach however we like in the future. Our high-level module doesn't depend on a lower-level one (our database bits) and the abstractions we're using don't depend on the details of the implementation.

These are not the same thing, though they sound alike and, in many cases, look alike. Dependency Inversion is simply structuring our code to work with interfaces in a particular way (as described above). Dependency Injection is how these interfaces are provided to the classes that need them.

Something I Find Useful

Here we arrive at the end of my research into patterns and principles and the beginning of my opinion on the matter. It's important to me that you recognize that what's to follow is like every other thing you'll read about architectural strategies in software: *just my opinion, not necessarily factual*.

I do think it's important to share our experience, but I

don't like positioning that experience as “The Way”. Of course it's not The Way. How could it be? Yet, there is value in learning about how others have built things (so you can come up with your own “Way”) so I decided to change course in this book, for this one section, and share with you my *opinion*.

So here you go. All of what you're about to read is, in effect, how I've been assembling programs for the last decade². Some things felt right, some didn't and I have been able to tweak the things that didn't feel right as the result of writing this book. Now they feel very good indeed.

I remember buying book after book in the late 90s and early 2000s as I was trying to improve my skills as a developer. I read about domain-driven design (DDD), TDD, BDD and was constantly reading Martin Fowler's blog, digging for guidance on how to put things together. I was convinced that knowing my patterns and principles would help me write better software. I was wrong.

It's easy to get vapor-locked on doctrine, focusing on that instead of solving the problems you're getting paid to solve. This is a common thing but I'm here to suggest a way to avoid this entirely: come up with your own personal, architectural “baseline”. The thing you do when you're not sure what else to do.

Here's mine.

My goal is to keep models as light as possible and the dependencies between *everything* as light and simple as I can make them. Here are the rules:

- Models are simple classes, inheriting *nothing* and can only change their own state. As you'll read more in the next chapter, this is a functional approach with the goal of “code purity”.
- Decisions are made in “service” classes, which

model a process. These classes are the most susceptible to coupling, which is something I try to avoid at all costs.

- Service classes are handed everything they need to execute, taking the smallest dependency footprint possible. This is achieved by making sure I inject interfaces or lightweight structs whenever and wherever I can.

That's the core of it really. To be more concrete about it, let's use the shopping cart analogy one more time:

- A **ShoppingCart** class is concerned with one thing only: holding items for a buyer and representing that data.
- A **Checkout** class might represent the idea of closing a sale, accepting a cart, payment processor and repository as arguments and using these to execute business logic.
- The cart, payment processor and repositories would be represented by interfaces, not the actual classes. If I'm using something like Ruby or JavaScript, these would be represented by lightweight structs or hashes.

I feel like I'm waiving my arms around some and I really want to show you what I mean with code, but I think it's more valuable if you could *feel* this approach and visualize it mentally.

When checking out, who cares about the cart? What you really care about are the items in the cart so in that sense all I really need to do is to have the SKUs and quantity of each item. The total will be determined by the checkout routine

which will go and look up the products using the repository interface.

A lot of frameworks (Rails, Django and others) provide models that are database aware which means that in order to test them, you have to boot up the entire application to have the context for that model. That's some serious coupling, and also very slow. It's great for smaller applications, absolutely blows for bigger ones.

I favor models that inherit nothing, have the fewest number of dependencies possible and fit on a single page (aka "no-scroll models").

Using C#, for instance, we might be tempted to create a ShoppingCart class that holds ShoppingCartItems and has a checkout method:

```
class ShoppingCartItem{
    string sku {get;set;}
    int quantity {get;set;}
    string name {get;set;}
    int price {get;set;} //pennies
}
class ShoppingCart{
    IEnumerable<ShoppingCartItem> items;
    constructor(){
        this.items = new List<ShoppingCartItem>();
    }
    addItem(Product product){
        //adding logic
    }
    checkout(){
    }
}
```

This will work just fine in most cases, but I've come to *loathe* dependencies of any kind. My ShoppingCart depends on Product and ShoppingCartItem and the checkout method

will surely depend on a payment processor of some kind as well as a database.

This is going to get yucky, fast. For instance - our Checkout process might be used in the future to process a Subscription ... which means writing code to get around the type restriction in our addItem method. This means that we'll likely be breaking things.

What if, instead, we used an interface to define this relationship:

```
interface IBuyable{
    string sku {get;set;}
    int price {get;set;} //pennies
    int quantity {get;set;}
}

class Product: IBuyable {
    //interface bits
}

class ShoppingCart{
    IEnumerable<IBuyable> items;
    constructor(){
        this.items = new List<IBuyable>();
    }
    addItem(IBuyable item){
        //adding logic
    }
}
```

The only thing my cart cares about, really, is a SKU, price and a quantity. We're now free to add other things that our users might want to buy - like gift purchases, gift cards or support time.

Also notice that I removed the checkout method. That's a business process and, to me, belongs in a service class:

```

class Checkout {
    ICatalog catalogRepo {get;set;}
    IPaymentProcessor processor {get;set;}
    constructor(ICatalog catalog, IPaymentProcessor processor){
        this.catalogRepo = catalog;
        this.processor = processor;
    }
    process(IEnumerable<IBuyable> items){
        //pull the products from the catalog
        //validate the order
        //process the sale
        //save it
    }
}

```

This is where the decisions happen. Creating a checkout requires the tools to execute the checkout - a payment gateway and some type of repository of products (which I'm calling a catalog).

The process method executes the transaction. This is where the heavy lifting happens! Note that since we're using interfaces here, we can mock them easily for testing, which is a huge bonus.

You might be wondering: *all these interfaces! Where do you instantiate your actual types?* I tend to push that as far out as I can. A lot of people will orchestrate these things using Inversion of Control containers (see above) so that types and instances are defined and dealt with in one place.

For me, I like to keep things exceedingly simple. In this case, I might have the Controller (if I'm creating a web application) or a config file somewhere create the actual classes for me:


```
//pseudo code
class CheckoutController {
  constructor(){
    this.stripe = new StripeGateway();
    this.catalog = new ProductRepository();
  }
  checkout(post){
    var checkout = new Checkout(this.catalog, this.stripe);
    checkout.run(post.items)
  }
}
```

This is pseudocode, but hopefully you get the idea. When the controller is created, it creates the classes that I will then pass to an instance of my Checkout class. This, to me, is about as complex as I like to keep things.

If you don't have interfaces, let's say you're using JavaScript, you could implement something like this:

```
class Checkout {
  constructor(repo, {charge}){
    this.repo = repo;
    this.chargeMethod = charge;
  }
  process(items= []){
    //...
  }
}
```

We're doing the same thing but, fortunately or not, we're leaning on the dynamic nature of JavaScript to help us out. All we care about for our Checkout class is that we have a repository passed in (repo) and something with a charge method, which would be our Stripe gateway.

The bare minimum needed to run, that's what we want to focus on.

I find this way of doing things useful for two main reasons:

- It keeps my models extremely lightweight and easy to test.
- It allows for easier testing with my service classes given that I can mock, or stub, the things my service classes need.
- It keeps my tests fast since I don't need to boot up an entire application context, with a database connection, in order for them to run.

Of course things could get more complex as time goes on but this chapter isn't about architecture nor how to scale the complexity of your application. I did think it would be nice, however, to show you the patterns I use on a daily basis.

TEST DRIVEN DESIGN

WRITING TESTS THAT DICTATE YOUR APPLICATION'S DESIGN

I am, strictly speaking, not a practitioner of Test-driven Design (TDD) or Behavior-driven Design (BDD). I know what these things are and I *feel* why they are useful. I also know that people like to argue about what they think it is and what they think it is not.

This leaves me with a problem. How do I explain what these things are without being over-opinionated in one direction or another?

So here's my plan: *just show the essence of the idea*. I think we can all agree on that, can't we? For this chapter I approached several friends and asked them about what they considered the essence of testing to be, and how they think of using it. By the way – each of them hedged their opinions with a variation of “this isn't strictly TDD (or BDD)... but...”.

Testing requires discipline and you're not alone if you sort of do it. As long as you're testing your code!

As with many of the chapters in this book, the code for everything you're about to read is at GitHub. Clone or download if you want to play along.

Some Opinion About Testing In General

Before we begin, let's have a think about testing your code, aside from TDD. No matter what: **test your code**. There really is *no excuse* to not test what you create, to make sure it's correct.

I will say this, directly: if you're not testing the code that people are paying you money to write *you deserve to be fired*. I won't justify that remark - if you don't believe me you should move on. Better yet: close this book and rethink your career. The fact that we should test is a given. It's *how* (and in some cases when) we test that is the subject of this chapter.

In that spirit, let's engage with Test-driven Design (TDD) and Behavior-driven Design (BDD) focusing on the fun parts. It's my hope you can see how creating your tests with a design pattern in mind can be extremely useful.

We'll start with TDD and then move on to BDD. After that, we'll discuss some of the filigree that goes into testing code in general.

The Nuts and Bolts of TDD

Unless you've been living under a rather large rock over the last 10 years, you've heard of TDD. Maybe in good ways, possibly in bad ones. At its core it's a simple practice:

- You think about what you need to create
- You write a simple test to get yourself started
- You run that test and watch it fail
- You write some code to make the test pass
- You write another test for the next step, and repeat the process

As you go along, you're constantly refactoring what you've written – and this is the somewhat goofy part: you write the barest minimum to make a test pass.

As Close To a Real Example As I Can Get

A few years back I recorded a video with my friend Brad Wilson and the idea was to capture him doing “real” TDD. No to-do list, no fake blog demo example ... *real stuff*. Brad is the creator of XUnit (along with Jim Newkirk) and is an everyday practitioner of TDD. I couldn't imagine a better person for that video.

The video was for my former company, Tekpub, and it was entitled “Full Throttle: TDD With Brad Wilson”. Brad didn't know what I was going to ask him to do – so we sat together, I recorded his desktop, and then asked him to create a subscription billing system for me.

What Brad did next changed the way I thought about TDD. Unfortunately, the video is no longer available (it belongs to Pluralsight who has retired it) – but I will recount the highlights for you here.

Just Start

Brad used Visual Studio 2012 and C# 4.5, creating a library project (BillingSystem) and a test project (BillingSystem.Tests) in a matter of seconds. He added XUnit to his test project and then paused to think about a few things:

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Xunit;

namespace BillingSystem.Tests
{
    public class BillingDoohickeyTests
    {
        //Monthly billing
        //Grace period for missed payments
        //Not all customers are necessarily subscribers
        //Idle customers should be automatically unsubscribed
    }
}

```

Two things about this code struck me. The first is Brad's use of comments (lines 11 through 14) to get the concepts in his head out onto the screen and, more importantly, he's not getting in his own way thinking about names and structure.

Brad had no idea what to call his test suite just yet, so he called it **BillingDoohickeyTests** in part for fun, but also to remind himself to rename it once things started rolling.

What comes next? What classes should we create right off the bat? This plagues many developers who can't even get past this point.

Here's the thing with TDD that causes anxiety almost immediately: it takes rigor, and it feels pretty silly, if I'm honest. So far nothing we've done is overly goofy (apart from the naming thing) – but in a second, you'll see what I mean.

You can do a little thinking upfront, but TDD tries to discourage over-engineering by pushing you to let your tests tell you what to write. That's where we're going to start.

We're building this system so we can charge customers, so why not start there? This is exactly what Brad does:

```
namespace BillingSystem.Tests2
{
    public class BillingDooohickeyTests
    {
        //Monthly billing
        //Grace period for missed payments
        //Not all customers are necessarily subscribers
        //Idle customers should be automatically unsubscribed
    }

    public class Customer
    {
    }
}
```

He just put the class to test right there, next to his test code. Why not? TDD is a rigorous process, but it doesn't need to be slow.

OK, so we have a **Customer**, now we need to charge the customer on a monthly basis. At this point: *stop thinking*. Let's put this idea in motion with a test:

```

namespace BillingSystem.Tests3 {
    public interface ICustomerRepository { }
    public interface ICreditCardCharger { }
    public class BillingDoohickeyTests3 {
        [Fact]
        public void Monkey () {
            var repo = new Moq.Mock<ICustomerRepository> ();
            var charger = new Moq.Mock<ICreditCardCharger> ();
            BillingDoohickey thing = new BillingDoohickey (repo.Object, charger.Object);
            thing.ProcessMonth (2016, 8);
        }
        //Monthly billing
        //Grace period for missed payments
        //Not all customers are necessarily subscribers
        //Idle customers should be automatically unsubscribed
    }
    public class BillingDoohickey {
        public BillingDoohickey (ICustomerRepository repo, ICreditCardCharger charger){}
        public int ProcessMonth (int year, int month) {return 0;}
    }
    public class Customer{}
}

```

A lot just happened here. Let's step through it.

You'll notice that Brad isn't concerning himself, again, with names. We have **Monkey** and **thing**, which might be making you cringe – but for Brad, he's removing obstacles to his design process.

Which is what TDD is supposed to be: a **design process**.

Next, he's using mocks as you can see on lines 12 and 13, provided by the Moq project (fake classes for testing) upfront so he doesn't need to think about implementation just yet – he's leaving that for later. We'll discuss mocks and stubs (also known as “test doubles”) more in just a minute.

At this point we have a little machinery to play with, but we still don't know what we're doing completely. When I write tests, the very first thing I do is to create what some people call the happy path: one or more tests that pass when everything works as we expect it to work.

In other words, if we're building a registration system then our happy path would be something like **User_is_regis-**

tered_with_a_valid_login_and_password. This test should always pass.

Once our happy path is set, we go about trying to break it. We'll do that later. Right now, let's create a "happy path" for ourselves to get us off the ground, renaming Monkey to focus ourselves:

```
namespace BillingSystem.Tests4 {
    public interface ICustomerRepository { }
    public interface ICreditCardCharger { }
    public class MonthlyChargeTests {

        [Fact]
        public void Customers_With_Subscriptions_Due_Are_Charged () {
            var repo = new Moq.Mock<ICustomerRepository> ();
            var charger = new Moq.Mock<ICreditCardCharger> ();
            BillingDoohickey thing = new BillingDoohickey (repo.Object, charger.Object);
            thing.ProcessMonth (2016, 8);
        }
        //Monthly billing
        //Grace period for missed payments
        //Not all customers are necessarily subscribers
        //Idle customers should be automatically unsubscribed
    }
    public class BillingDoohickey {
        public BillingDoohickey (ICustomerRepository repo,
                                ICreditCardCharger charger) {}

        public int ProcessMonth (int year, int month) {
            return 0;
        }
    }
    public class Customer{}
}
```

In this code I've renamed a few things because I now have an idea what I'm doing. The test suite is called **Monthly-ChargeTests** and my test name makes it clear what it's going to test for. This name is far too broad, but it will change at some point.

The simple renaming, however, has forced me to consider a few more bits of functionality:

- What is a Subscription?
- What does it mean for a Subscription to be Due?

- A Customer, apparently, needs to have a Subscriptions property

I stop here – thinking about YAGNI ¹(You Aint Gonna Need It). It's tempting to plow ahead and add a **Subscription** class and a **Subscriptions** property to my **Customer**... but do I need to just yet? It does seem obvious, but this is the rigor part.

Let's focus on the test, add an assertion, and move on from there:

```
[Fact]
public void Customers_With_Subscriptions_Due_Are_Charged () {
    var repo = new Mock<ICustomerRepository> ();
    var charger = new Mock<ICreditCardCharger> ();
    BillingDoohickey thing = new BillingDoohickey (repo.Object, charger.Object);
    var processed = thing.ProcessMonth(2016,8);
    Assert.True(processed > 0);
}
```

This is where we venture into silly land. Our first goal is to make sure this code can compile – so I've added the interfaces and classes that we need. I also added a **ProcessMonth** method to **BillingDoohickey** and, for now, I'm returning 0 because I don't know what else to return.

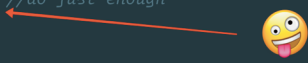
If we run this test, it will fail. We'll also probably feel a bit badly about ourselves because we might not have any customers with subscriptions due ... so what then? I'll get to that in a minute – for now we can compile our code and run this test: **watching it fail**.

That's a critical aspect here – our initial test needs to fail because we don't want to accidentally write a test that passes! Which does happen.

Now, let's get our test to pass:

```
public class BillingDoohickey {
    public BillingDoohickey (ICustomerRepository repo,
                           ICreditCardCharger charger){}
    public int ProcessMonth (int year, int month){
        return 1; //do just enough
    }
}

public class Customer{}
```



Ugh. Our tests pass and, as dumb as it seems, *this is TDD*. We will fix this code and, in fact, the dumber it feels the better it is because it forces you to write more tests just to get this kind of thing out of your code!

For now, our happy path is set. Let's blow it up.

The sad path is all about trying to blow up the happy path. It's "what happens when I do this!" The obvious first thing is to write a test that is in complete opposition to our happy path:

```
[Fact]
public void Customers_With_Subscriptions_Due_Are_Charged () {
    var processed = thing.ProcessMonth (2016, 8);
    Assert.True (processed > 0);
}

[Fact]
public void Customers_With_No_Subscriptions_Due_Are_Not_Charged () {
    var processed = thing.ProcessMonth (2016, 8);
    Assert.True (processed == 0);
}
```

I moved some declarations around in my test because I want to keep things DRY ²(Don't Repeat Yourself) – a messy test suite is one you'll want to stay away from – so I moved everything up top and into the constructor.

Next, I created the opposite test, asserting that customers without subscriptions would not be charged – which fails

because I've hard-coded the result into the **BillingDoohickey**.


Now I need to think about a few things. Specifically: *what does it mean to have no subscriptions?* **Who cares!** For now, let's get this test to pass.

I'll start by setting the mock for **ICustomerRepository** available for orchestration using the **repoMock** variable:

```
ICustomerRepository repo;
ICreditCardCharger charger;
BillingDoohickey thing;
Mock<ICustomerRepository> repoMock;

public MonthlyChargeTests () {
    repoMock = new Mock<ICustomerRepository> ();
    repo = repoMock.Object;
    charger = new Mock<ICreditCardCharger> ().Object;
    thing = new BillingDoohickey (repo, charger);
}
```

Next I'll add a method called **Customers** to the **ICustomerRepository** interface because my test told me I needed to:



```

[Fact]
public void Customers_With_Subscriptions_Due_Are_Charged(){
    repoMock.Setup (r => r.Customers())
        .Returns (new Customer [] { new Customer () });

    var processed = thing.ProcessMonth (2016, 8);
    Assert.True (processed > 0);
}

[Fact]
public void Customers_With_No_Subscriptions_Due_Are_Not_Charged (){
    repoMock.Setup (r => r.Customers())
        .Returns (new Customer [] { });

    var processed = thing.ProcessMonth (2016, 8);
    Assert.True (processed == 0);
}

```

Now that I've done this, I can refactor the **BillingDoohickey.ProcessMonth** method to return a count of the records in the repo. This is a prime example of “just doing enough to get the tests to pass”:

```

public class BillingDoohickey{
    ICustomerRepository _repo;
    public BillingDoohickey (ICustomerRepository repo,
                           ICreditCardCharger charger) {
        _repo = repo;
    }
    public int ProcessMonth (int year, int month){
        return _repo.Customers().Length;
    }
}

```

This kind of thing is fun when pair coding. I remember pairing with a friend once and laughing so hard at just how creative she was at writing the dumbest code possible to get my tests to pass:

You keep writing tests like that, I'll keep writing code like this.

It's fascinating to split your personality when testing like this. See if you can outsmart yourself with a more interesting test – something to break the happy path for once and for all! Then stave off the attack with some goofy way around it, like we just did here, returning the count of users.

It's tempting to scrap this test and try to write something more concise, which you're welcome to do. I typically just write another test specifically so I can get this crappy code out!

How about this:

```
[Fact]
public void A_Customer_With_Two_Subscriptions_Due_Is_Charged_Twice ()
{
    var customer = new Customer ();
    customer.Subscriptions.Add (new Subscription ());
    customer.Subscriptions.Add (new Subscription ());

    repoMock.Setup (r => r.Customers())
        .Returns (new Customer [] { customer });

    var processed = thing.ProcessMonth (2016, 8);
    Assert.True (processed == 2);
}
```

What do you think about this? You can play my pair who's writing code to get the tests to pass ... think you can make me sad?

Here's a way to do it:

```

public class BillingDoohickey
{
    ICustomerRepository _repo;
    public BillingDoohickey (ICustomerRepository repo, ICreditCardCharger charger) {
        _repo = repo;
    }
    public int ProcessMonth (int year, int month) {
        var customer = _repo.Customers().FirstOrDefault();
        if (customer == null) {
            return 0;
        } else {
            return customer.Subscriptions.Count ();
        }
    }
}

public class Customer {
    public IList<Subscription> Subscriptions { get; set; }
    public Customer () {
        this.Subscriptions = new List<Subscription>();
    }
}

```

Ha! With that test I was able to push the code so that two new concepts could be added: a **Subscription** and a property on **Customer** called **Subscriptions**. You were still able to write silly code to get the tests to pass – so you win too!

Notice the ongoing *failure, pass, refactor ... failure, pass, refactor* that we're doing here. That's exactly what TDD is all about and despite the way many people make it seem – it can be extremely fun. Especially if you have a pair to code with you – real or imaginary.

A final thing to notice, *I did one step at a time*. I didn't write out a set of tests upfront, which would defeat the idea of challenging myself as I go with YAGNI.

Test Doubles and Your System Design

As your test suite grows, you're going to find yourself up against the "Integration Wall", which can present problems. The most typical problem you'll face is *persisting data* and how to treat things that need to be saved to a database as part of a test run.

Using our subscription payment scenario, let's say that a payment comes in as a webhook ping from Stripe, the popular payment processor which handles recurring billing. A typical process might be:

- The ping is received and saved with some kind of initial status.
- A Payment is created from this ping and the ping's status is updated
- A Subscription is found based in information on the Payment, some form of customer ID for instance. The ping is once again updated with the subscriber info
- The Customer for that Subscription has their status reset with a payment recorded.
- An email is sent to the customer thanking them for their continued support
- The ping status is set to "resolved" so we know it was handled OK

A process like this needs to be thoroughly tested, of course, but how do you do it without hitting the database repeatedly?

This is where mocking and stubs can greatly help, but they can also hurt tremendously if we do them wrong.

Don't Mock What You Don't Own

I can't tell you how many times I've fallen back on this rule when my testing strategy was falling apart. It's tempting to think "well I'll just mock the X driver/service so I don't hit the database", but that's disastrous for a simple reason: *you'll never be able to match what someone else has done*. It almost always comes back to bite you!

The code you “own” is the code you’ve written for your application. Feel free to mock it however you want, but wrap the code you don’t own (like an ORM, for instance) in a service class that you can mock or stub later on³.

I tried to do this once with Nodemailer and I’ll never do it again. It seemed like such a small thing but once I went down this path I found I had to mock out some very deep internal method calls and it made me want to cry.

Databases are also a problem in this way. Mocking an ORM, for instance, is something that at least every developer must try at least once. There is just no way you’ll ever be able to do this properly and I feel like you’re going to try anyway, which is how we learn isn’t it? When you’re mocking out your 20th method response (incorrectly) perhaps you’ll remember the phrase *don’t mock what you don’t own* and try something different.

What’s the answer then? That’s a big question, but the short answer is to rethink your system design and what each object is doing. That’s why they call testing in these ways a “design process”. It’s a happy coincidence that a testable application is typically one with an architecture that lends itself to change. We like this as software architects, but at this point I’m going to stop because this book is *not* about architecture!

Avoiding Common Problems

It’s always simple to talk about TDD and BDD in the abstract, but when your test suite tips 300 different specifications and tests... things can be a bit difficult to manage. You find this out quickly when one change in your codebase causes you to update 50 different tests.

These are called “brittle” tests and can really be a pain and I hate to say it, but there really is no avoiding it: *it’s going*

to happen at some point. Some IDEs excel at refactoring (such as Visual Studio) so if you change a property name, for instance, the change ripples throughout the code. If your tests break due to a property name change, however, you might want to think about why that is.

Obviously tests that focus on the property you changed *should* break. This is a natural part of refactoring. When unrelated tests break it's often because you're doing too much or involving too many external factors for a given test.

Testing our `MonthlyBilling` service would be one such candidate for brittle tests. Knowing this from the start, however, gives you an advantage! How much do you need to involve your `Customer` and `Subscription` classes in the `MonthlyBilling` service? If you change `Customer.firstName` to `Customer.first` – will that break your `MonthlyBilling` tests? I sure hope not! In that sense you might want to work with a lighter interface for the customer rather than the entire thing. Perhaps an `IBillable` or something?

This is a great discussion to have with your colleagues and is precisely why testing can guide your architecture to good places. The simple reason is this: **if your tests are hard to refactor, your code will be 10 times harder!**

Like I said above: *I get lazy sometimes.* OK *many* times. I'll get into this in more detail in the chapter on Behavior-driven Design (BDD - coming up next) but I tend to use tests as more of a checklist. Things I expect to work, etc.

I often get carried away and I just keep coding – which I know is bad. I pay for this choice often when I'm deleting code that took a while to get to work, but that I ultimately don't need.

So, I take a deep breath, maybe I'll go for a walk or get some tea. When I come back, I clean up my tests and refocus myself. This is TDD to me – more of a battle with myself than anything.

BEHAVIOR DRIVEN DESIGN

THINKING OF YOUR APPLICATION OUTSIDE-IN

Behavior-driven Development (BDD) is basically the same process as TDD, but you have a specific focus: *behavior of the application*. It's a subtle shift, but an important one.

Getting Started

In the last section on TDD we began to build out a **Billing-System** using TDD which works, but it's slightly mechanical. In other words: Widget X will return Y when I pass in Z. This defines what we expect to happen as developers, not what we expect as humans.

Let's shift this to focus on a story instead, with some scenarios. We'll start with what our application will do when a payment is received for a monthly subscription:

```

using System;
using Xunit;

namespace BillingSystem.Specs {
    [Trait ("Monthly Payment Is Due", "Payment Is Received")]
    public class PaymentReceived
    {
        [Fact]
        public void An_Invoice_Is_Created(){}

        [Fact]
        public void Subscription_Status_Is_Updated(){}

        [Fact]
        public void Next_Billing_Is_Set_1_Month_From_Now(){}

        [Fact]
        public void A_Notification_Is_Sent_To_Subscriber(){}
    }
}

```

As you can see, I’m thinking in terms of how the system will react when an event happens. Another way to put this is “this is the specified behavior of a feature”. The mental approach to this idea is outside in, meaning that we typically come at this type of testing from a user’s perspective. Therefore you’ll sometimes hear this type of testing as “acceptance testing”.

BDD, like so many things, has been jargoned to death and is also subject to “cargo-culting”, which means if you call your file “something_spec” and the thing you’re testing a “feature” given a “context” then you’re doing BDD.

This is not the case. BDD is about behavior of your system when a thing happens, that’s all.

For instance, what happens when a payment fails? Let’s spec it out¹:

```
[Trait ("Monthly Billing", "Payment Fails")]
public class SubscriptionPaymentIsDue
{
    [Fact]
    public void An_Invoice_Is_Not_Created(){}

    [Fact]
    public void Next_Billing_Is_1_Day_From_Now(){}

    [Fact]
    public void A_Notification_Is_Sent_To_Subscriber(){}
}
```

This code is doing the same basic thing as TDD: *testing our code*. This time, however, we're doing it in the form of "how does our application respond when this thing happens". In other words: **behavior**.

Philosophy

This approach is different from strict unit testing, which tends to be more clinical. In other words, you might put a certain class under test, vary the input data to see where it fails and then refactor until it succeeds. This is fine, but has some disadvantages, which are:

- The tests are bound to the design of your class by definition. That is the point of TDD. If you change your design, you have to change your tests and your code. This can be quite frustrating.
- The focus is on engineering, not application experience. When you're focused on code, the code wins. When you're focused on behavior, however,

you're focused on the user's experience and the business wins.

- Testing proliferates. The tendency with TDD and unit testing is to have as much code coverage as possible. When you use BDD, you typically write few tests which are more targeted to application experience.

With BDD you tend to write your tests detailing what the application will do under certain circumstances. Given this, BDD fans will call their tests “specifications”, as they tend to read as if dictated directly by the client. In the example above, I'm using XUnit's **Trait** attribute to decorate my scenarios so they're a little more readable in the test runner.

I've also made sure that my test names rely completely on the test class itself (called the scenario). When you run this test, you see this (or something like it):

```
[Monthly Billing]: Payment Received
- An_Invoice_Is_Created
- Subscription_Status_Is_Updated
- Next_Billing_Is_Set_1_Month_From_Now
- A_Notification_Is_Sent_To_Subscriber

[Monthly Billing]: Payment Fails
- An_Invoice_Is_Not_Created
- Next_Billing_Is_1_Day_From_Now
- A_Notification_Is_Sent_To_Subscriber
```

This is the XUnit runner output, which you can jiggle in Visual Studio if you want. If you're using a framework in another language (like Mocha for Node or RSpec for Ruby) you can have a more readable output.

Features, Scenarios, Expectations

I try to focus on the notion of *Feature*, *Scenario*, *Expectations*. I know this might seem like a syntax dance to you, but it's incredibly easy to lapse back into unit testing mode – not that there's a problem with that! Unit tests are indeed needed in some cases (testing utility code, parsers, etc.).

Let's revisit our example code:

```
//...
namespace BillingSystem.Specs {
    [Trait("Monthly Billing", "Payment Is Received")]
    public class PaymentReceived{
        //... the constructor prepares the test data
        [Fact]
        public void An_Invoice_Is_Created(){
            //...
```

In this code you can see the features directly by examining the **Trait** attribute on the test class. The first element is the feature (“Monthly Billing”), the second is the scenario (“Payment Is Received”). Your testing library might have a different way for specifying these things.

So What?

You might be wondering, at this point, why all of this even matters? BDD does have several advantages:

- **Readability.** It sounds idealistic but being able to print out a test run and read (in common language) what's going on is powerful.

- **Ubiquity.** I hate that word, but it's applicable here: you know what these tests describe, and your client/boss will know as well. In this, you're speaking the same language.
- **Focus.** If you're focused on how your application behaves, you're aligning yourself with the business goals. This is important for programmers! You can watch your application evolve into something exciting and understand why it's doing what it's doing. You might even have some questions about this, which means you can contribute your genius to the application's design.

There's obviously some jargon to muddy up what is, otherwise, an elegant development practice. I brought this up before so let's find out what I mean.

On Jargon and Cargo Culting

BDD tests are typically called specifications or "specs". In the theoretical world you could sit with your client, create a list of specifications for various aspects of your application, and then translate that directly into your test suite.

In the real world I've found that my clients have never cared about my tests. Maybe it's just my clients – not sure – but even when I worked at Microsoft and tried to show progression using my test suite I was laughed out of the room.

Clients like to see results, not test runs. It is good, however, to use the same language in your tests as you do with them in email or on the phone. Trying to align your thinking is important.

To that end, you could have this conversation and you could, using BDD, translate it directly into some tests:

So let's recap the conversation: given a successful monthly billing – the billing system should generate an invoice, set the next billing date to exactly one month from now, set the user's subscription to active and then email the user. Correct?

Say this out loud to yourself, as if in conversation. Feel free to use your own words.

Do you hear any problems? Your client probably will. Here's a reply I received once, when I said almost exactly that sentence to a startup client back in 1999:

Rob - yes for the most part this is correct but accounting handles the invoices so I think just notifying them will work. I don't think we should be creating our own invoices. As far as email goes, I'd want to loop in our marketing team as they own client communications...

Do you see what just happened there? Not only did I save myself some work, but I also opened a great conversation about the role of our application within the new company.

The feature we were discussing is the monthly billing run. The scenarios we created were payment received and payment failure; these are also called contexts. We describe the behavior of the application in response to a scenario with specifications:

```
Monthly Billing Run //feature
- Payment is successful //scenario, or context
- an invoice is created //specification
```

Is this jargon important? *Yes, and no.* It is important in the sense that you should be thinking in these ways when doing BDD. It's also important to know that just because you call a test class **MonthlyBillingFeature** and a method on that test class **SuccessfulPaymentScenario** does not mean you're doing BDD. BDD is a process of discovery for both you and your client.

That said, you can call a feature a **pancake**, a scenario a **lovelybutterfly** and each specification **larry**. The naming doesn't matter – as long as your team understands what it is you're doing and why.

Given, When, Then

Cucumber is a popular Ruby test tool that helps you focus on BDD. It popularized a certain syntax, called Gherkin:

```
Feature: monthly billing run
  Scenario: payment received
    Given a charge of 20 USD
    And today is the 1st of the month
    When the charge is applied to Subscription x
    Then that subscription is considered active
    And an invoice is created
    And the customer gets an email
```

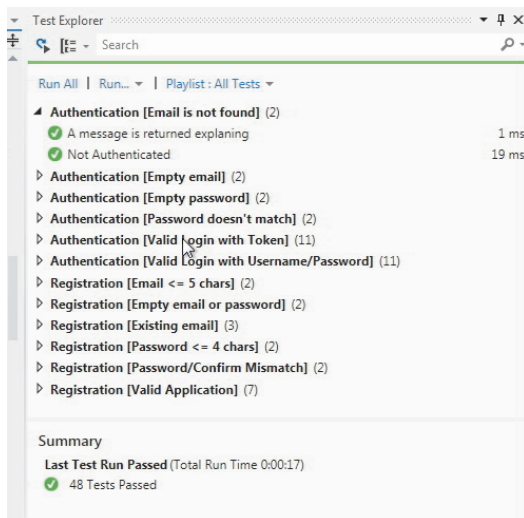
Gherkin is a specialized syntax you can use to get your head into the Given, When, Then syntax in a formal way. These rules might sound goofy but when you just start getting into BDD it can help get your mind in the right place.

It's effective but it does take time to break things down in this way. Think about an application you're writing now ... how would you detail the behavior of it? Give it a go!

In The Real World

You might be wondering what this really looks like once a project has been up and running for a few months. Is it possible to scale this idea? To keep it readable and focused?

This is a project I did five years ago, using .NET, Visual Studio and XUnit - no additional tooling:



The feature under test is the first item in the list (**Authentication**, for instance) and the scenario is the second. Underneath each feature/scenario is a set of specifications. This approach worked well for me.

If you want to check out the code, I put it up at GitHub. The code is a few years old now, but it should still work.

If you're feeling a little vague on the idea still, I don't blame you. It took me a while to get my tests (ahem, sorry: specs) to flow with behavior vs simple unit tests. The best thing I can offer you is to look at the difference between the

billing system code at the start of this chapter compared with the code from the last chapter on TDD.

This is important: *both are valid, and both are lovely*. Some people favor TDD, others BDD. Experiment!

FUNCTIONAL PROGRAMMING

IMMUTABILITY, CURRYING, CODE PURITY AND A
VERY BRIEF DISCUSSION ON MONADS

Some people love it and claim it's the only way to write software. Others see it as a fad and roll their eyes.

Let's come away from those extremes. Functional Programming is based in a foundational concept which we learned about a few chapters ago: *Lambda Calculus*. It's not magical, nor is it something you should ignore because you're amazing. It is simply something you need to understand.

Like most things in Computer Science, functional programming is full of jargon, idioms and practices that, at first, might be a bit opaque. If you take the time, however, to let it soak in... functional programming can change the way you write software if you come from an object-oriented background.

A Change In Thinking

I learned a functional language two years ago; one that I love using: *Elixir*. It has changed the way I think about programming. It's the only functional language I know, so I'll be using it to do some of the demos you're about to see.

For others, I'll be using JavaScript/ES6. I chose this language because 1) most programmers know it at least a little and 2) it has some functional characteristics - at least to the level where you can get your point across.

We'll cover four main topics in this chapter:

- Immutability (things that can't/don't change)
- Purity (functions rely only on what they're given)
- Side Effects (functions only operate on data they're given)
- Currying (breaking big functions down to little ones)

I also threw in a very brief discussion about functors and monads at the very end, something I barely understand but I think is critical to have a think on.

Off we go...

Immutability

The first word that comes to mind whenever you hear “functional programming” is usually “immutable”. As I am sure you know, it means “not changeable” in plain English, but how does that translate to programming? Moreover: *who cares?*

If you're an object-oriented (OO) programmer you're used to creating classes and then instantiating them. You set

their properties and send them messages which change their properties or tell you something about themselves. This is not possible with functional programming.

There are no objects, no classes, no “state” as you might think of with OO. There are only functions which transform things. You give a function what it needs, it hands you back what you want (hopefully). This might sound incredibly confining, but there are practices that go along with this idea that make it rather compelling. Let’s have a look.

If you want to follow along, you should have Elixir (version 1.3 or so) installed, as well as Node (version 6+).

Consider this assignment:

```
friend = %{name: "Clara"}
IO.inspect friend
#%{name: "Clara"}
```

This is a **map** in Elixir and it looks a lot like an object in JavaScript or a hash in Ruby. The difference is you can’t do this:

```
friend = %{name: "Clara"}
friend.name = "Mike"

*** (CompileError): cannot invoke remote function friend.name/0 inside match
[Finished in 0.159s]
```

The error message is a bit odd as it has to do with the way Elixir tries to match values (pattern matching), but essentially it means “you can’t change Clara’s name to Mike”. So, what do we do?

We ask the **Map** library to update the map. This is going

to look a bit weird, but once I explain it more you'll hopefully understand:

```
friend = %{name: "Clara"}
friend = Map.put(friend, :name, "Mike")
IO.inspect friend
#%{name: "Mike"}
```

I do not blame you if you're completely underwhelmed at this point... and more than a little confused. There's just no graceful way to go about this, so I'm throwing you in the deep end straight away.

Here's what just happened:

- We created a map with a name key set to "Clara"
- We asked the Map library to put the value "Mike" in Clara's place
- The Map library gave us back a completely new map ... sort of
- The friend variable was rebound to the new Map

Elixir provides some helpful features which you don't find in more "strict" functional languages. For instance: in Erlang (a functional language that Elixir is based on) you can't do what we just did (rebinding **friend**). You would need to create a whole new variable - something like **renamed_friend** to hold the result of the **Map.put/2** operation. Elixir is less strict, so it *looks* like we're changing the **friend** variable.

A natural thought that you might be having at this point is *wait a minute, if I need to update a map, I need a whole new one? Isn't this horrible for memory?* This is a very good question! The short answer is: *no, it's not*. This is because Elixir simply uses

a pointer under the covers. The initial **friend** map is still there and the updated map points back to it. This keeps things rather light and you don't fall into the immutability traps that you have in other languages (strings with .NET, for example). Furthermore: this kind of thing is only possible with a functional language.

That's the technical teardown, let's discuss something a bit more intangible: *this code is just ugly*. I would agree with you there too. Fortunately what I wrote is *not* idiomatic. I said at the beginning that functional programming is all about *transforming* data through a set of functions. Let's rewrite our code to support that idea:

```
%{name: "Clara"} |> Map.put(:name, "Mike") |> IO.inspect
```

Much better. That toothy thing is the “pipe” operator and works in much the same way as its Unix counterpart. The result of a given function is piped into a following one as its first argument, and so on.

Before we end our introduction to immutability, I want to underscore the notion of rebinding. Let's change our code a bit, and I'll reintroduce our **friend** variable:

```
friend = %{name: "Clara"}
friend |> Map.put(friend, :name, "Mike") |> IO.inspect
IO.inspect friend
```

Running this you'll see that our **friend** variable wasn't changed at all. **Map.put/2** simply passed back a new map with the updated key, which we piped into **IO.inspect** directly. Thus, we have two different outputs:

Formalizing Data with Structs

I don't want to leave you with the impression that data in a functional language like Elixir is just tossed around without any formalized rules. There are no classes in Elixir, but their close relatives, "Structs", do provide some of their utility.

Let's formalize our code, making our intentions clearer and giving our friends some structure:

```
defmodule Friend do
  defstruct name: "Clara", age: 0
end

defmodule Immutability do
  def change_name(friend, new_name) do
    Map.put(friend, :name, new_name)
  end
  def get_friend do
    %Friend{}
  end
end

Immutability.get_friend |> IO.inspect
```

This looks better don't you think? Structs allow you to set defaults for your data and give it a prescribed structure. Elixir also gives you some shorthand for updating a struct:

```
#...

friend = Immutability.get_friend
%{friend | name: "Mike"} |> IO.inspect
IO.inspect friend
```

Once again, we get the same result as we did with

Map.put/2 above as the rebinding returns an immutable new **Friend** struct:

```
%{name: "Mike"}  
%{name: "Clara"}  
[Finished in 0.22s]
```

Transforming Data

If you're new to functional programming ideas than none of this is probably convincing. It takes a while to get into the functional groove, so let's write some more code.

Three things to focus on for this section are:

- We are transforming data by passing it through a set of functions
- We like smaller functions
- We can treat functions the same as values

The latter comes straight from Lambda Calculus. In fact all of this does - that's where functional programming has its roots in case that wasn't obvious. In addition, things work better overall if we focus on smaller, clearer functions.

To see this in action, let's do some math, shall we?

```

defmodule Ops do
  def square_it(num), do: num * num
  def double_it(num), do: num + num
  def root_it(num), do: :math.sqrt(num)
  def print_it(num), do: IO.inspect num
end

4 |> Ops.square_it
|> Ops.double_it
|> Ops.root_it
|> Ops.print_it

# 5.656854249492381
# [Finished in 0.274s]

```

What we have done here is fairly typical in the functional world. Small, concisely-named functions are arranged as needed, and we can simply pass some data along. This type of approach can work with anything, you just have to change your thinking a bit.

Now, you might be thinking about how certain things might be solved with functional programming, maybe translating problems you work on every day? It's an interesting thing to ponder, but from my experience it takes a good two weeks to really hit that AHA! moment... at least for me.

What you need is a *reason to care about it all* to pull you through the process of changing your thinking. Let's see if I can help with that.

First: *no, functional programming is not perfect nor the answer to everything*, of course. But it does greatly expand your ability to think through a solution!

The first advantage of functional programming is its immutability. Quite a few bugs in OO are caused by the state of something not being correct, or what you wanted. You might have quite a few tests that show that yes, indeed, this

code *should work* provided X, Y and Z conditions are met, but then condition Σ comes along and screws the whole thing up!

If you write code that depends on anything going on outside it, you're prone to these types of errors. It's the same reason most developers loathe global variables - one change and it ripples throughout your program, causing things to break.

Functional programming is different in that, ideally, a function should receive everything it needs to do its job, when asked. There are all kinds of safeguards you can attach to these functions to prevent them from being called if the data isn't correct. As opposed to being a burden, it's quite freeing! To get it right, however, you have to change your thinking, which takes time and effort.

Speaking of effort, let's get back to it. We'll pick up this conversation again later.

A Real Example: A Shopping Cart

Let's compare and contrast a functional style vs. an OO style, and I'll do that by creating a **ShoppingCart** in Elixir and also in JavaScript.

Let's start with JavaScript:

```

class Cart{
  constructor(){
    this.items = [];
  }
  addItem(item){
    //make sure it's a proper item
    //and then...
    this.items.push(item);
  }
}

const cart = new Cart();
cart.addItem({sku: "SOCKS", price: 12.00})
cart.addItem({sku: "MORESOCKS", price: 18.00})

```

Lovely. A very, very simple cart but it captures the idea: we have a class, an instance and we change the state of our instance by adding items to it.

Elixir is a bit different. As a first pass you might be tempted to do something like this:

```

defmodule Cart do
  items = []
  def add_item(item) do
    %Cart{items: item}
  end
end

```

Which wouldn't work. Functional languages don't have the notion of state, so holding on to an **items** array would cause an error. We need to pass everything to the **Cart** that it needs to operate, *including the items*:

```
defmodule Cart do
  def add_item(items, item) do
    items ++ item
  end
end
```

This works and, once again, underscores the notion of rebinding. Let's tweak this to look more like functional programming:

```
defmodule Cart do
  def add_item(items, item) do
    items ++ item
  end
end

items = []

Cart.add_item(items, %{sku: "SOCKS", price: 12.00})
|> IO.inspect

# %{sku: "SOCKS", price: 12.00}
# []
```

Much cleaner. Still a bit... *wonky* however. We're passing around arrays and maps without knowing what's going on. Let's clean this up a bit.

When you're forced to do without state and objects, you begin to think in terms of data and things that happen to that data. In our example we're using OO thinking by trying to represent a **Cart** as an object, which isn't very functional of us. Instead, we should think about it in terms of data moving through a *process*.

The **Cart** is our data, and our process is, more accurately, described as **Shopping**:

```

defmodule Cart do
  defstruct items: [], total: 0, count: 0
end

defmodule Shopping do
  #use pattern matching to guarantee data we need
  #the _ ignores the data, we just want the pattern
  def add_item(%Cart{} = cart, %{sku: _, price: price} = item) do
    %{cart | items: cart.items ++ [item], total: cart.total + price, c
    ount: cart.count + 1}
  end

  #get from the DB?
  def get_cart, do: %Cart{}

  #save to DB?
  def save_cart(cart), do: cart
end

```

Now, to add an item, we *transform* the **Cart** using the **Shopping** process:

```

# add an item to the cart
Shopping.get_cart
|> Shopping.add_item(%{sku: "SOCKS", price: 12.22})
|> Shopping.save_cart
|> IO.inspect

```

I threw a bit more Elixir at you in this go, hope you don't mind. Of note is how I'm able to use pattern matching to guarantee the data this function needs, which is a **Cart** and a map with a **sku** and a **price**.

From end to end, I'm able to retrieve a **Cart** from somewhere (assume it's a database for now), add an item to it and then put it back. It's clear what this process is by reading the code and the only change of state is that of our database, which is acceptable in functional programming realms. Barely.

Side Effects and Purity

There are two terms that you often hear in discussions about functional programming: *purity* and *side effects*. Both terms stem from the idea of immutability.

The whole notion of interacting with a system outside the scope of the function you're in is called a "side effect" - something that happens as a result of your function being invoked. Working with a database, for instance, is referred to as a "necessary side effect" because you're changing the state of something outside the scope of your function.

The more you do this, the less "pure" your code is. *Purity* is not a term dedicated to functional programming - it refers to the level of interaction any code has with the outside world. You can write "pure" code in OO programming just as you can with functional, it just happens to have a little more focus in the functional world.

Functional programmers like purity. It's easier to test a function that doesn't change behavior based on some external setting or function call. It's also easier to debug. The downside is that you end up with more functions that are (typically) a lot smaller than what you might write in OO land.

There's a way to work with those, too, as we're about to see.

Currying

Functions, functions, functions. They're everywhere! Organizing a program full of them can feel overwhelming, especially when you're just starting out with functional programming.

Let's look at one of the very first practices you'll want to

take advantage of: *currying*. Currying is the act of using smaller, single arity functions in a chain rather than a larger function with multiple/complex arguments. It's easier to understand using code.

Consider date night with your partner:

```
const dateNight = (who, what, where) => {
  return `Out with ${who} having fun ${what} at ${where}`;
};
```

This function takes 3 arguments (or, in more programmy speak, has an arity of 3). If we split these functions into a smaller set of chained, single arity functions, we're currying:

```
const nightOut = who => what => where => {
  return `Out with ${who} having fun ${what} at ${where}`;
};
```

Looks a weird, doesn't it? Especially if you're not used to lambdas. But how might we use this function? Like this:

```
const funTime = nightOut("Dancing")("wife")("Club 9");
console.log(funTime);
//Out with wife having fun Dancing at Club 9
```

Does this look familiar to you? It should, this is the *exact* thing we did when applying values to functions in Lambda Calculus a few chapters back.

I know what you're thinking: *calling functions like this is a bit goofy*, and I would agree. This is where the idea of *partial application* comes in:

```
const dancing = nightOut("Dancing");
const dancingWithWife = dancing("wife");
const funTime = dancingWithWife("Club 9");
console.log(funTime);
//Out with wife having fun Dancing at Club 9
```

This is the power of currying. Let's see something a bit more concrete, however.

I worked with a very nice, very opinionated developer recently who loves her object-oriented programming. I was trying to explain how I thought it would be fun to add some more functional ideas to the data access code we were writing, and she was rather resistant. Finally, she said “just show me what you mean”.

So I did. I started out with a curried function set for building a basic select query:

```
const selectQuery = table => (where, params) => order => limit => {
};
```

At this point I simply needed to evaluate what was sent in to each function call:

```
const selectQuery = table => (where, params) => order => limit => {
  const whereClause = where ? ` where ${where}` : "";
  const orderClause = order ? ` order by ${order}` : "";
  const limitClause = limit ? ` limit ${limit}` : "";
  if(params.length > 0) params = [params];
  const sql = `select * from ${table}
    ${whereClause}${orderClause}
    ${limitClause}`; //wrapped this for readability
  return {sql: sql, params: params};
};
```

I've written a lot of code for building SQL statements, and this is probably the smallest I've ever created. Using this is even more fun:

```
const usersQuery = selectQuery("users");
console.log(usersQuery()());
//{sql: "select * from users", params: []}
```

You probably wouldn't want to create a query in that way, however, and this is where partial application comes in. You could create a module (let's call it a repository for fun) which built up the following:

```
const usersQuery = selectQuery("users");
const allUsersQuery = usersQuery()();
const usersByEmail = email => usersQuery("email = $1", email)();
console.log(usersByEmail("test@test.com"));
//{sql: "select * from users where email = $1", params: ["test@test.com"]}
```

I think this is interesting, and a nice way to reuse functionality. Let's round this out by plugging in pg-promise and having it actually do something. I'll be using the Chinook test database for this:

```

const pgp = require('pg-promise')();
const db = pgp("postgres://localhost/chinook");
//our selectQuery as before
const selectQuery = table => (where, params=[]) => order => limit => exec => {
  const whereClause = where ? ` where ${where}` : "";
  const orderClause = order ? ` order by ${order}` : "";
  const limitClause = limit ? ` limit ${limit}` : "";
  if(params.length > 0) params = [params];
  const sql = `select * from ${table}${whereClause}${orderClause}${limitClause}`;
  const query = {sql: sql, params: params};
  //should we execute?
  if(exec){
    db.many(sql, params).then(res => exec(res)).catch(err => console.log(err))
  }
  //if not, just return the query
  else return query;
};
const albumQuery = selectQuery("album");
const albumSearch = title => albumQuery("title LIKE $1", `${title}%`)();
const rockAlbums = albumSearch("Rock");
//fine it up!
rockAlbums(res => {
  console.log(res);
})

```

If you're playing along and run this code, you should see all the albums in the Chinook database with the term "Rock" in the title.

When I showed this code to my friend, her response was wonderful:

...oh my god, that's equal measures ingenious and horrifying :D Being able to reuse selectQuery, albumQuery, etc. almost seems like a twisted take on models at an arbitrary scope...

Twisted indeed. Personally, I dig it!

A Very Brief Discussion About Functors and Monads

There's a great line from Douglas Crockford about monads:

In addition to it being useful, it is also cursed and the curse of the monad is that once you get the epiphany, once you understand - "oh that's what it is" - you lose the ability to explain it to anybody.

From "Monads & Gonads", December 2012

Very, very true. That's why I want to keep this discussion as brief as possible and invite you to do your own investigation. You should have a little bit of an itch in your brain right now, wondering *just how am I supposed to orchestrate things beyond simply currying my way to insanity?*

This is where structures like *functors* and *monads* (among others) come in.

When you're slinging functions around as we have, you might want to have a higher level of abstraction for working with them. Something that might *wrap* those functions and handle certain situations for you the way you want.

One thing that you might want to do is to interrogate/iterate over some values, such as an array or a struct like a **user**. You could write a loop or do some type of recursion, or you could orchestrate the effort using a set of functions.

A popular way of doing this is to create a function which allows you to map values from a given object:

```
class Monkey{
  constructor(val){
    this.__value = val;
  }
  map(fn){
    return fn(val);
  }
}
```

This is simplistic, but hopefully you get the idea. This simple construct has a lofty name: it's a *functor*. It's only purpose in life is to run **map** over things that are *mappable*.

Functors have big brothers that will implement some logic on top of that mapping, namely returning some alternate values based on the data contained in your mappable object. For instance, you might want to access a **name** field on your **user** object, but don't want bad things to happen if there is no **name** field.

This is where monads come in, and where I start gracefully backing my way to the door. They're easy to understand in concept, and when you first encounter them, it can be a little anticlimactic. I don't know why this is. Monads really aren't terribly scary, but it's like some beautiful gateway to programming hell: if you understand it, you're doomed. I've found Crockford's rule above to be spot on.

Let's see if you agree with me.

Assume that we're allergic to **if** statements. This is one lovely oddity that happens when you start working with functional languages. There are so many interesting constructs and ways of doing things (like pattern matching) that you find yourself avoiding complex conditional trees.

Our **selectQuery** is pretty good and I feel happy about it, but we wouldn't be allowed into Functional Club with this code:

```
const selectQuery = table => (where, params=[]) => order => limit => exec => {
  const whereClause = where ? ` where ${where}` : "";
  const orderClause = order ? ` order by ${order}` : "";
  const limitClause = limit ? ` limit ${limit}` : "";
  if(params.length > 0) params = [params];
  const sql = `select * from ${table}${whereClause}${orderClause}${limitClause}`;
  const query = {sql: sql, params: params};
  //...
};
```

Monads exist to handle just this case! Let's create the simplest one: the **Maybe** monad:

```
class Maybe{
  constructor(val){
    this.__value = val;
  }
  isNothing(){
    return (this.__value === null || this.__value === undefined);
  };
  map(fn){
    return this.isNothing() ? Maybe.of(null) : Maybe.of(fn(this.__value));
  };
  val(){
    return this.isNothing() ? "" : this.__value;
  };
}
Maybe.of = (val) => new Maybe(val);
```

That's not so scary is it! See what I mean? Ah the temptation... let's see if we can confuse ourselves a bit.

This class exists to do one thing: *handle conditional values* in an orchestrated, functional way. When you call **map()** you say “here's a function, use it if there's a value”. The trick, however, is that if there is a value for your monad then **map** hands you back *another Maybe monad* with its value set as the result of your mapped function call.

This means we can move data through a chained set of functions without worrying about **null** or **undefined** values. How fun.

We can now use our **Maybe** monad to evaluate our **whereClause**:


```
const whereClause = Maybe.of(when).map(w => `where ${w}`).val();
```

If **when** has a value then our lambda will be called and we'll get back another **Maybe** with its current value set to a where clause. We can then call **val()** if we just want this value directly, which we do. If when isn't set, we'll get back an empty string.

We can apply this to the other arguments as well:

```
const whereClause = Maybe.of(when).map(w => `where ${w}`).val();
const orderClause = Maybe.of(order).map(o => `order by ${order}`).val();
const limitClause = Maybe.of(limit).map(l => `limit ${limit}`).val();
```

We could, at this point, build our SQL statement as before by building a string with template literals, but when's the fun in that! Let's use our **Maybe** monad one more time:

```
const sql = Maybe.of(`select * from ${table}`)
  .map(sql => `${sql} ${whereClause}`)
  .map(sql => `${sql} ${orderClause}`)
  .map(sql => `${sql} ${limitClause}`)
  .val();
return {sql: sql, params: params};
```

Let's step through this. We start out with an initial value that we want to work with, in this case it will be our **select * from \${table}** statement. We move this value through a successive chain of functions, where it is applied to a function that is passed in via **map**. That value is then passed on to the next in the chain. If you're thinking that this looks a lot like Elixir's **|>** (pipe) operator, you'd be correct.

Note: if you're a functional programming person and you see a better way to do this, please do let me know!

Here's the final code:

```

const selectQuery = table => (where, params=[]) => order => limit => {

  const whereClause = Maybe.of(where).map(w => `where ${w}`).val();
  const orderClause = Maybe.of(order).map(o => `order by ${o}`).val();
  const limitClause = Maybe.of(limit).map(l => `limit ${l}`).val();

  const sql = Maybe.of(`select * from ${table}`)
    .map(sql => `${sql} ${whereClause}`)
    .map(sql => `${sql} ${orderClause}`)
    .map(sql => `${sql} ${limitClause}`)
    .val();

  return {sql: sql, params: params};
};

const albumQuery = selectQuery("albums");
const albumSearch = title => albumQuery("title LIKE $1", `${title}%`());
const rockAlbums = albumSearch("Rock");
console.log(rockAlbums());

//{
//  sql: 'select * from albums where title LIKE $1 ',
//  params: '%Rock%'
// }

```

Wahoo!

DATABASES

NORMALIZATION, ANALYSIS AND REPORTING, CAP
THEOREM, SHARING AND REPLICATION

There is an established way to design a transactional database: following the rules of normalization. This is the process of essentially turning a single, large spreadsheet into a set of related tables.

Transactional systems can back our applications, but analytical systems power decision-making. Data warehouses and data marts store the data and OLAP systems provide the analysis infrastructure.

Distributed database systems use the notion of *horizontal* scaling (adding more machines) vs *vertical* scaling (increasing machine size and power). Distributed systems are much different from traditional “big machine” databases in that they have to balance tradeoffs with data consistency, availability, and network problems.

Social media is driving the need to store gigantic, petabyte-sized data stores. This movement is called Big Data and it’s often difficult to grasp the sheer size of it all.

Normalization

A relational database consists mainly of a bunch of tables. These tables contain rows of data, organized by columns. There must be a method to this madness, and it's called normalization.

Database normalization is all about controlling the size of the data as well as preserving its validity. Back in the 70s and 80s you simply did not have disk space, but companies did have a ton of data that would fill it up. Database people found that they could reduce the size of their database and avoid data corruption if they simply followed a few rules.

Before we get to these rules, let me just add upfront that it's truly not that complicated once you grasp the main ideas. As with so many things computer science related, the jargon can be rather intense and off-putting, making simple concepts sound hard. We'll slowly work our way up to it.

Finally: rules are meant to be broken. In fact a DBA will break normalization quite often in the name of performance. I'll discuss this at the end.

We've decided to buy a food truck and make tacos and burritos for a living. We have our ingredients and drive up to our favorite street corner, opening our doors for business.

When the orders come in we put them into a spreadsheet – knowing we'll need to deal with it later on:

email	name	order_id	items	price
joe@example.com	Joe Tonks	1	Pollo Burrito, Diet Coke	\$12.50
jill@example.com	Jill Jones	2	Carne Asada, Sprite	\$14.50

Let's move this spreadsheet into the database, altering it as we go.

First normal form (1NF) says that values in a record need

to be atomic and not composed of embedded arrays or some such. Our items are not atomic – they are bunched together. Let's fix that.

email	name	order_id	items	total
Joe @ Example.com	Joe Tonks	1	Paolo Burrito Diet Coke	\$10.50
Jill @ Example.com	Jill Jones	2	Carne Asada Sprite	\$14.50



email	name	order_id	items	total
Joe @ Example.com	Joe Tonks	1	Paolo Burrito	\$8.00
Joe @ Example.com	Joe Tonks	1	Diet Coke	\$2.50
Jill @ Example.com	Jill Jones	2	Carne Asada	\$12.00
Jill @ Example.com	Jill Jones	2	Sprite	\$2.50

Lovely. Our new orders table is 1NF because we have atomic records. You'll notice that things are repeated in there, which we'll fix in a bit. Our next task is to isolate the data a bit.

Now that we're in 1NF, we can move on to 2NF because part of being in 2NF is complying with 1NF. Our task to comply with 2NF means that we need to identify columns that uniquely define the data in our table.

An order is a customer buying something. In our case the email field uniquely identifies a customer, and the **order_id** field uniquely identifies what they've ordered. You put that together and you have a sale – which could uniquely identify each row in our table.

The problem we have is that name does not depend on the **order_id** and items and price have nothing to do with email – this means we’re not in 2NF. To get to 2NF we need to split things into two tables:

email	name	order_id	item	total
JOE @ EXAMPLE.COM	JOE TONKES	1	PALO BUTTOS	\$ 8.00
JOE @ EXAMPLE.COM	JOE TONKES	1	DIET COKE	\$ 2.50
JILL @ EXAMPLE.COM	JILL JONES	2	CARNIE ACADA	\$ 12.00
JILL @ EXAMPLE.COM	JILL JONES	2	SPRITE	\$ 2.50



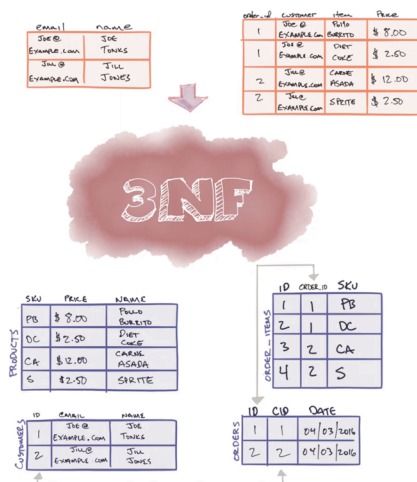
email	name
JOE @ EXAMPLE.COM	JOE TONKES
JILL @ EXAMPLE.COM	JILL JONES

order_id	customer	item	price
1	JOE @ EXAMPLE.COM	PALO BUTTOS	\$ 8.00
1	JOE @ EXAMPLE.COM	DIET COKE	\$ 2.50
2	JILL @ EXAMPLE.COM	CARNIE ACADA	\$ 12.00
2	JILL @ EXAMPLE.COM	SPRITE	\$ 2.50

Much better, but not quite there. Our customers table looks good, but our orders table is a bit of a mess.

3NF says that every non-primary field in our table must describe the primary key field, and no field should exist in our table that does not describe the primary key field.

Our orders table has repeated values for the key (which is a no-no) and the price of each item has nothing to do with the order itself. To correct this, we need to split the tables out again:



We now have 4 tables to support our notion of an order:

- **customers** holds all customer data
- **orders** holds metadata related to a sale (who, when, where, etc.)
- **order_items** holds the items bought
- **products** holds the items to be bought

You'll notice, too, that I replaced email with an integer value, rather than using the email address. I'll explain why in just a second.

Normalizing a database requires some practice. As programmers, hopefully you understand how to model classes and objects. It's almost the same process that we just went through: *what attributes belong to which concept?*

It's at this point that I get to tell you (with a sinister giggle) that the rules of normalization are more of a guideline, not necessarily law. A well-normalized database may be

theoretically sound, but it will also be somewhat hard to work with.


We managed to move a fairly simple spreadsheet with 2 rows and 5 columns into a 4-table structure with 3 joins and multiple columns! We only captured a very, very small fraction of the information available to us and our Taco Truck.

It's very easy to build a massively complex database with intricate lookups, foreign keys, and constraints to support what appear to be simple concepts. This complexity presents two problems to us, right away:

- Writing queries to read and write data is cumbersome and often error-prone
- The more joins you have, the slower the query is

The bigger the system gets, the more DBAs tend to cut corners and denormalize. In our structure here it would be very common to see a total field as well as **item_count** and embedded customer information.

To show you what I mean – have a look at the structure for the StackOverflow posts table:

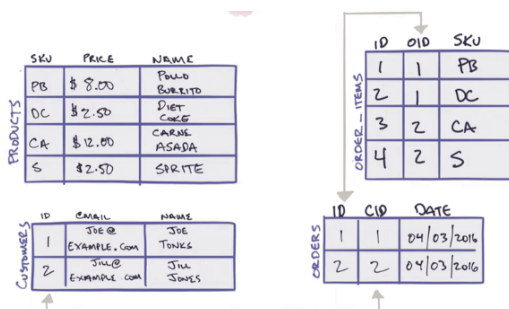
Name	Type	Length	Decimals	Dimen...	Not Null	Key
id	int4	0	0	0	<input checked="" type="checkbox"/>	
post_type_id	int4	0	0	0	<input type="checkbox"/>	
accepted_answer_id	varchar	255	0	0	<input type="checkbox"/>	
parent_id	int4	0	0	0	<input type="checkbox"/>	
created_at	varchar	255	0	0	<input type="checkbox"/>	
deleted_at	varchar	255	0	0	<input type="checkbox"/>	
score	int4	0	0	0	<input type="checkbox"/>	
view_count	varchar	255	0	0	<input type="checkbox"/>	
body	text	0	0	0	<input type="checkbox"/>	
owner_user_id	int4	0	0	0	<input type="checkbox"/>	
owner_display_name	varchar	255	0	0	<input type="checkbox"/>	
last_editor_id	varchar	255	0	0	<input type="checkbox"/>	
last_editor_display_name	varchar	255	0	0	<input type="checkbox"/>	
last_edit_date	varchar	255	0	0	<input type="checkbox"/>	
last_activity_date	varchar	255	0	0	<input type="checkbox"/>	
title	varchar	255	0	0	<input type="checkbox"/>	
tags	varchar	255	0	0	<input type="checkbox"/>	
answer_count	varchar	255	0	0	<input type="checkbox"/>	
comment_count	int4	0	0	0	<input type="checkbox"/>	
favorite_count	varchar	255	0	0	<input type="checkbox"/>	
closed_date	varchar	255	0	0	<input type="checkbox"/>	
community_owned_date	varchar	255	0	0	<input type="checkbox"/>	

Notice the **view_count**, **owner_display_name**, and **_count** fields? Also the **last_editor_id** field?

The count fields are calculated and don't belong in a table, theoretically speaking. The **owner_display_name** and **last_editor_id** fields should be foreign keys that link to an **authors** or **users** table of some kind – and they do with **last_editor_id** and **owner_id**. Querying this massive table using the required joins, however, would be way too slow for what they need.

So, they denormalized it. Many businesses do – it just makes things faster and simpler.

Let's take a look at the final schema we came up with:



While it is theoretically correct, there is a problem with being historically correct. For instance, if you come to my Taco Truck and buy some Carne Asada, I'll have a record of it stored happily in my orders table.

When I run my sales queries at the end of the month, your sale will be in there, adding \$12.00 to the total. In July of this year, I have \$6800 in total sales! Wahoo!

Sales have gone well and being a good capitalist I decide I'm going to charge \$15.00 for Carne Asada from now on. I'm proud of myself and so I run July's sales reports one more time so I can print them out – I want to see that money rolling in!

Hmmm. The numbers are off for some reason. It used to say \$6800 for July, but now it says \$7300! What happened?

We've made a rather critical mistake with our design, here. One that you see constantly. The deal is that **order_items** is what's known as a "slowly changing historical table". The data in this table is not transactional, it's a matter of record.

So, what do you do if you want to avoid changing the past? We'll discuss that in the next section.

OLAP and OLTP

99% of the databases you and I work in are considered "OLTP": Online Transaction Processing. This type of system is based on performance – many reads, writes and deletes. For most applications this is appropriate.

At some point, however, you're going to want to analyze your data, which is where "OLAP" comes in: *Online Analytical Processing*. These systems are low-transaction systems that change little, if at all, over time apart from nightly/weekly loads. These systems power data warehouses and support data mining.

The structure of each system varies quite a lot. OLTP systems are relational in nature and are structured using the rules of normalization discussed in the last chapter.

OLAP systems are heavily denormalized and are structured with dimensional analysis in mind. Building these systems can take hours and usually happens on a nightly basis, depending on the need.

Let's start where OLTP ends and OLAP begins...

Extraction, Transformation, and Loading (ETL)

My accountant, who's also my best friend, has the same thing to say to me every year when preparing my taxes: "trash in, trash out". That's his warning to me as I prepare my account statements to bring to him.

This is a form of ETL that people do every year (especially in the US): pull all their financial data from their banks, savings, investments etc. and compile it in a single place – maybe Excel. They go through it at that point, making sure it all adds up. Each bank statement reconciles and there are no errors.



You do the same with analytical systems. The first step is to extract the information you want from your OLTP system (and/or other sources) and comb through it for any errors. Maybe you don't want null sales totals, or anything tagged "test".

You then transform as required. Reconciling customer information with your CRM system so you can add history data, account numbers, location information, etc.

Finally, you load the data into your system, which is usually another database that has a special layout. The system I'm most familiar with (and spent years supporting) is Microsoft's SQL Server Analytical Services (SSAS) so I

would usually extract the data from one SQL Server database to another.

They also had a built-in transformer that worked with VBScript, of all things! I used it sometimes but often it would fail. We later moved to a system called Cognos that was a gigantic pile of XML pain.

Today, you can perform quite complicated ETL tasks efficiently and simply by using a set of simple scripts. These can be as simple as shell scripts or, more commonly, quite complex using a programming language like Python or Ruby. Python's speed and popularity make it a very common choice for ETL.

Data Marts and Warehouses

You'll often hear these terms used interchangeably, but they're two very different things. A data warehouse is like a filing cabinet in your office or at home where you keep all your financial information: statements, tax documents, receipts, etc. Hopefully you keep this organized so it's easy to sift through and put in a form that your accountant can understand – such as an Excel spreadsheet.

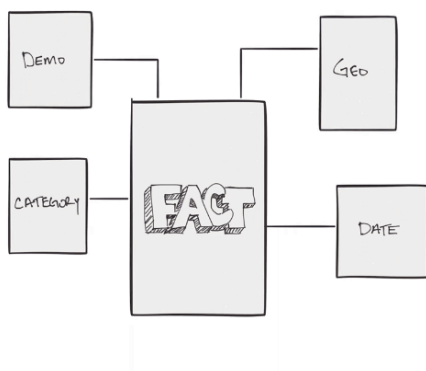
There are other things we can do with this data; maybe we want to know how much we spent on groceries so we can budget properly next year. We might want to calculate how much our life savings could be if we had any ... stuff like that. For now, however, we need to get some data to our accountant because it's tax season here in the US, so we load the data into Excel with a targeted use case: Accounting.

This is a data mart. A place (typically focused/targeted) that can answer questions. We could use the Data Warehouse for this (sending the accountant our shoebox full of receipts and statements) but that would take her a long time and she wouldn't be happy.

Data Mart Schemas

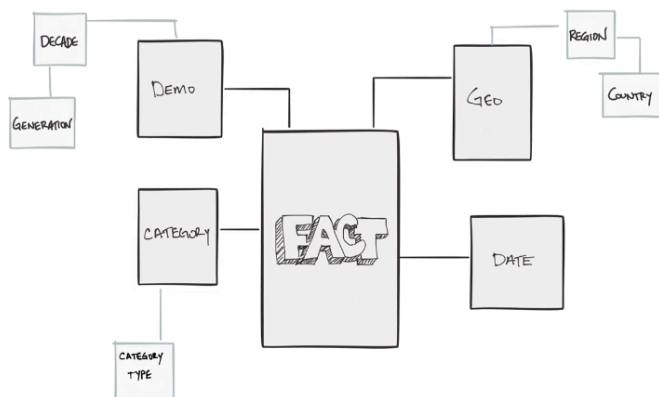
The Excel spreadsheet is an apt way to think about how data is stored in a data mart: flattened. You might have a few joins in there, but the fewer the better because processing the data mart, which is typically millions and millions of records, will slow down with more joins.

What does this look like, however? The one you'll see most often is the star schema:



In the center is a table called “the fact table” which represents a single fact you want to report on. For my accountant this would be a singular transaction of some kind: deposit, debit, adjustment, etc.

A *snowflake* schema is the same, but the dimension tables themselves have more dimensions.



The fact table has keys which link off to dimensional look up tables, which you can use to roll up the data for specific queries. They're called dimensions because they present a different way of rolling up the data. For sales (or anything relating to people) these are typically:

- Categories of some kind
- Time (week, month, quarter, year)
- Geography (city, state, country)
- Demographic (gender, age, race)

Selecting dimensions is much harder than it seems. They need to be fundamentally different from each other, or you'll be rolling up on data that has crossover meaning.

You see this sometimes with schemas that confuse demographic data with geographic data – typically with the region that a person is from. I had an hour-long discussion with a client once about the meaning of “Southerner” in their data.

It might not seem like a big deal but making sure the data can be cross-checked is absolutely critical.

With a data mart it's possible (and common) to query on multiple dimensions at once. If we had left "Southerner" as a bit of demographic information, we would have had conflicting questions and answers:

- Show all sales for both men and women located in the Southern United States
- Show all sales for both men and women who are "Southerners"

I have friends in Hawaii who call themselves "Southerners". I have Hawaiian friends who live in Louisiana. What are we learning with these questions? *Analytics is difficult*. The point is: pick your dimensions with care and make sure you involve the people who are using the reports you'll generate.

The "Southerner" problem (as it became known) is intangible, and it takes some experience with data to be able to spot reporting issues like that.

Others are far easier to spot – such as "double-labeling", which happens all the time and is infuriating.

As a programmer I hope you have a blog where you share your ideas. If you do, it's likely you have a way of tagging your posts with small, contextual keywords (tags).

Let's do a counting query to find out how many comments your blog has for the tag **opinion** vs the tag **humor** (if you have such things... if not let's pretend). It's a simple enough query because, as it turns out, you only have 3 posts with 5 comments apiece:

- "Data Analysis is Silly" tagged opinion; 5 comments
- "Hadoop Honeybadger" tagged opinion and humor; 5 comments

- “A DBA Walks Into a Bar...” tagged humor; 5 comments

So, you run these queries:

```
select count(1) as comment_count from posts
inner join comments on post_id = comments.id
inner join posts_tags on posts_tags.post_id = posts.id
inner join tags on posts_tags.tag_id = tags.id
where tags.tag = 'humor'

--comment_count
---
--10

select count(1) from posts
inner join comments on post_id = comments.id
inner join posts_tags on posts_tags.post_id = posts.id
inner join tags on posts_tags.tag_id = tags.id
where tags.tag = 'opinion'

--comment_count
---
--10
```

Simple enough. But then you remember reading *The Imposter's Handbook* which mentioned cross-checking your rollup queries for accuracy, so you do:

```
select count(1) from comments;

--comment_count
---
--15
```


Uh oh. 10 humor posts + 10 opinion posts does not equal 15!

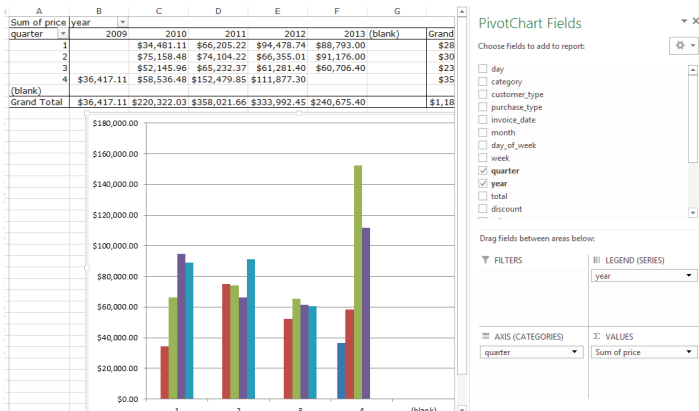
Now you might be thinking “of course it doesn’t” and that cross-checking like this is not accurate! My answer to that is “tell it to the product specialist who wants a sales rollup on various product tags”.

Right now, across the world, sales reports are in this “semi error” state. You cannot do rollup queries that involve many to many categorizations and expect to keep your job. Even if you add warnings! The numbers will suggest a reality that’s not there.

By the way: cross-checking like this is all part of ETL. Bad data should never make it into your data warehouse/data mart.

Analyzing a Data Mart

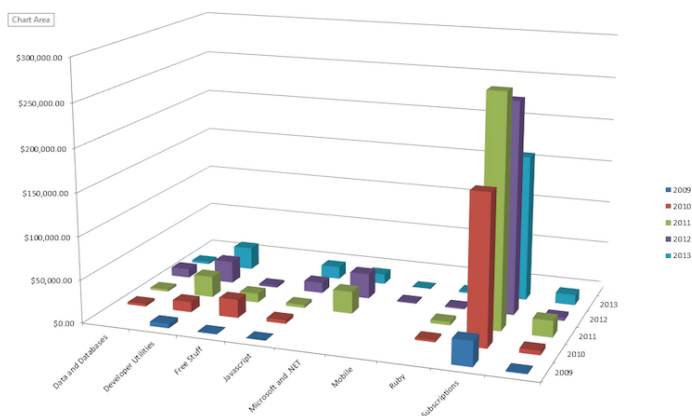
OK, you’ve gone through your data and have decided it’s clean of weirdnesses (good job!) and imported it into your data mart. How do we analyze it? The simplest way is with a Pivot Table and/or a Pivot Chart. It’s likely you’ve seen these in action – here’s some sample data in Excel:



The idea with a pivot table is that you can move dimensions around, rolling your facts up in various interesting ways.

The axes of the graph is a perfect example of why a dimension is called a dimension: the x dimension is often time and the y dimension is often the sum of sales.

What if you wanted to visualize sales by category over time? You just need to add another dimension to the graph – and thank goodness most people in the world can understand things in three dimensions:



Your boss likes this report a lot! It's interesting to give your data a "surface" as we're doing here because, in a way, you can feel what's going on. Now your boss wants to see this data with some demographic information – for instance the buying patterns between men and women.

That requires a fourth dimension. How in the world would you do that! Well, without getting into a physics discussion – you can treat time as a fourth dimension – which can work really well. Unfortunately for me, this book only works in two dimensions (with the illusion of a third),

so I can't show you a moving time-graph ... but close your eyes and see if you can imagine a three-dimensional graph slowly changing over time...

The neat thing is that time is that one of your dimensions so you can lift that to the fourth axis and watch sales by category and gender change slowly.

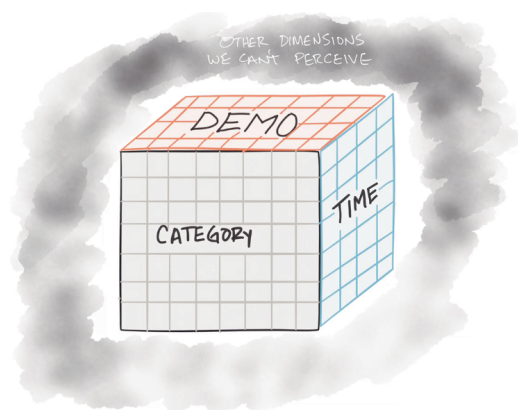
If your boss asks for more axes on this report you need to ask for a raise.

Using an OLAP Cube

Pivot tables work well over data structured in a flat way. If you have more than a few thousand rows, however, things can get mighty slow.

This is where a structure called an OLAP Cube comes in. You tell it about your fact table, the dimensions you're using and their hierarchy, and then begin processing.

An OLAP cube is simply a bunch of pre-calculated data. It's called a "cube" because the data is described typically in three dimensions, and as I mention above people can't really conceive more than four dimensions anyway. Time is usually one dimension, some type of categorization is another, and customer demographic is usually the third. Any more than that and things just get weird.



When you view data along a dimension, you're viewing a slice of the cube and you usually do this with a pivot table of some kind. Excel, for example, will hook up to an OLAP cube.

Pre-calculating data like this makes OLAP cubes very fast at preparing ad-hoc reports over millions of rows of historical data, but that comes at the cost of preprocessing the data. For this reason, data marts that act as the source of an OLAP cube should be structured in a very specific way ... and this will sound counterintuitive.

Your fact table should not have a primary key or indexes of any kind, for that matter. Inserting data into a table with an index means the database needs to update the index whenever data is inserted, which takes time.

Ideally, you've already vetted and cleaned your data during ETL and you trust it – so no keys or indexes. A fact table can grow to billions of records – can you imagine the index sizes on that!

Joins are slow, so denormalize your dimensional look up tables for speed of processing. Building an OLAP cube with millions of facts can take hours depending on the number of dimensions you're rolling up on.

Date rollups are the easiest thing to contain. For instance, your boss might think she wants a weekly sales report – but that's adding 52 additional slices to the OLAP structure – and every other dimension will need to be pre-calculated based on those 52 weeks times however many years.

This will move a three-hour processing run to an overnight run easily. So push back, if possible, or consider building an additional cube.

Distributed Database Systems

It's 2008 (or thereabout) and you're realizing that processors aren't really getting any faster. Buying more RAM used to solve all kinds of problems but lately you're finding that 12G is really all you need; the processor has become the bottle-neck for your database.

In 2010 you had two processors. In 2012 you have four – all for the same price. Today if you pay enough money, you can have 32 ... THIRTY TWO CPU cores on that lovely machine in the sky.

Can your database take advantage of each of those cores? THAT is the question for this chapter.

Multiple CPUs means that many things can be processed at once. This means the software has to be able to *do things in parallel* without freaking out. Parallel processing is not a simple topic – especially concerning data.

Imagine 3 things happening in parallel, each on a different core:

- user 3002 changed their password
- user 3002 updated their profile picture
- user 3002 canceled their account

What happens first? Is there a priority here ... and if so,

what is it based on? What happens if core #1 goes offline for 30 milliseconds because of network trouble?

Very Smart People have focused specifically on these problems over the last 10 or so years and come up with some interesting ways of doing things. **Parallel processing is where things are going** because that's where the hardware is taking us.

A Shift In Thinking

When computer science people tried to figure out data storage back in the 70s and 80s (aka: databases), they did so with two primary constraints in mind:

- Storage capacity: hard drives were not cheap so they had to focus on ways of storing data that would be extremely efficient with hard drive space. This led to column names like "UUN1" and hard-core adherence to normalization.
- Memory and processing speed: computers were simply slower, so storage needed to be optimized for read efficiency as well as overall size.

This is what most developers (including myself) "grew up" with. You used a relational engine to store data and you created your tables, keys, etc. in a very particular way.

NoSQL systems have been around since the 60s, but it was only in the late 90s that the development community really started to pay attention. Then, right around 2010, big software (Amazon, Facebook, Google, etc.) began to see the advantages of using NoSQL systems.

There was one advantage, however, that stood out above the rest: distribution. Simply put: **it's easier (technically and economically) to build distributed databases with a**

NoSQL system than it is to run a few, gigantic servers with oceans of RAM and disk space. Smaller servers are cheaper, and you spread your risk, mitigating data loss and disaster recovery.

You can scale horizontally with relational systems such as PostgreSQL and SQL Server – the problem, however, is that these systems need to remain ACID Compliant, which is a problem in distributed systems. They don't like to work in a parallelized way.

ACID-compliance means that you have certain guarantees when writing data in a transaction. In summary form, each transaction will be:

- **Atomic.** This means that a single transaction happens, or it doesn't. There is no concept of a "partial" transaction
- **Consistent.** The entire database will be aware of a change in the data whenever a transaction is completed. In other words: the state of the database will change completely, not partially.
- **Isolated.** One transaction will not affect another if they happen at the same time. This has the basic appearance of transactions being queued in a single process.
- **Durable.** When a transaction concludes it concludes. Nothing can change the data back unless another transaction changes the data back to the way it was. In essence: there is no undo.

Distributed systems are much different from this and rely on a different rule set entirely.

CAP Theorem

In 1998 Eric Brewer theorized that distributed processing of any kind can only provide two of the following three guarantees at any given time:

- **Consistency.** The same meaning as with ACID above; the state of the database will change with each transaction.
- **Availability.** The distributed system will respond in some way to a request.
- **Partition tolerance.** A distributed system relies on a network of some sort to function. If part of that network goes offline (thus “partitioning” the system), the system will continue to operate.

So far, this has proven to be true. Sort of. In 2012 Brewer wrote a followup which suggested “picking two of three” can be misleading:

...the “2 of 3” view is misleading on several fronts. First, because partitions are rare, there is little reason to forfeit C or A when the system is not partitioned. Second, the choice between C and A can occur many times within the same system at very fine granularity; not only can subsystems make different choices, but the choice can change according to the operation or even the specific data or user involved. Finally, all three properties are more continuous than binary. Availability is obviously continuous from 0 to 100 percent, but there are also many levels of consistency, and even partitions have nuances, including disagreement within the system about whether a partition exists.

Modern distributed database systems are addressing exactly this. RethinkDB is a prime example (full disclosure: it's one of my favorite distributed databases and I love it. Further disclosure: during the writing of this book the company behind RethinkDB closed its doors. The project, however, is open source and will continue).

You can choose the level of consistency you want on a per table or per query basis. Meaning that you choose whether you want an ack (acknowledgment of write to the entire system) or you can just trust the system to do it when it gets around to it.

In addition, you can architect your database on a table-by-table basis to enhance which of the three you want.

This can be really confusing, so let's dive into each of these ideas (as well as the jargon for each) using RethinkDB as an example system (because it's what I know).

You've heard of eventual consistency, it's a buzzword that makes many ACID-loving, relational DB people freak out. I was one of them.

The idea is a simple one: a write operation (think of it as an insert into query) is handed to the system and you receive an ack immediately, meaning that the system has received it and will write it to disk when it can.

At this point (before the write hits the disk), the database is not consistent. You can visualize this by thinking of a database with three nodes. One node receives the write request, which means the other two nodes are not consistent.

This inconsistency is risky. If the power goes out for some reason, the write will be lost, which could be a bad thing depending on the data.

The benefit, however, rests squarely on the benefits of distributed systems in the first place: parallel processing. The

system is available to handle more operations, so many things can happen at the same time, making the system extremely fast.

If the node handling the write goes offline (due to a netsplit, or network partition) it doesn't matter (assuming it still has power) because the write is queued, and will remain queued, until the rest of the system is brought back online and consistency is achieved.

These systems are called "AP systems" (generally) and are built to handle gigantic loads with guaranteed uptime. Facebook's Cassandra and Riak from Basho are prime examples of these systems.

An Alternative to ACID: BASE

Systems that focus on availability and partition tolerance comply with BASE:

- Basically Available
- Soft state
- Eventually consistent

If you've grown up with ACID systems, as I have, this idea might sound nightmarish. ACID is all about data paranoia: *protect it at all costs*. Make sure it's written to disk in isolation.

BASE, on the other hand, is the opposite of paranoid – which kind of makes sense when you consider the idea of strength in numbers: the more machines, the wider the risk is spread.

ACID systems, on the other hand, are typically "big machine" systems. You scale these systems up (bigger hardware) as opposed to out (more hardware).

The problem for ACID, however, is there is only so big

you can get. BASE systems can scale massively simply by adding more nodes. I'm not saying this is easy by any stretch, running these systems is very complicated – but it's doable and is being done in very large companies. We'll discuss this more in the chapter on Big Data.

You're in a meeting where the CTO has just announced that your fictional company, Red:4 Aerospace, is moving its orbiter telemetry data over to a distributed system. She's heard about CAP and needs a risk analysis – and she's looking at you.

What do we tell her in our analysis?

The first thing is that we'll gain a ton of processing power, which is good. Customers might have to wait a few extra seconds (depending on load), but if things get too slow we can just add a few more nodes!

This has the dual advantage of mitigating a netsplit. If we're strategic about our nodes and data centers we should be able to survive just about any outage.

But... what about the data? *There is the possibility of data loss*, always, when you make the AP tradeoff. Losing data can mean losing business (or worse) and to people (like me) who live and breathe the idea of Good Data, this is a hard subject to muse on clearly.

This is where we begin staring out the window as we remember various Hacker News threads on the unreliability of MongoDB which, in many developers' minds, means all distributed NoSQL systems. We remember the many blog posts we've read that ultimately resolve to "we lost data because we did NoSQL wrong"...

You've turned in your report on AP systems and the CTO now wants to know what other options are. Most (if not all) of the distributed database systems out there support partition tolerance well – it's just a matter of choosing availability or data consistency.

Do you want your system to stay up? Or the data to be correct? RethinkDB and MongoDB lean towards the latter – they are CP systems. By default, RethinkDB will only acknowledge a transaction when the data is persisted to disk.

Both MongoDB and RethinkDB are configurable so that you can tweak consistency settings the way you want and need. You can make some tables more AP if you want as the data allows.

The more you get into the nuances of CAP and how modern NoSQL databases handle it, the more confusing things get. Rapidly. As with every topic in this book I could fill chapters and chapters with detail. Instead, I'll leave the CAP discussion here and suggest you read more about how RethinkDB and MongoDB are put together.

It's time to turn our attention to the mechanisms for tweaking availability and consistency.

Distributed databases specialize in handling very large volumes of data. They do this by “spreading the load” between individual database nodes.

I've been focused on NoSQL systems like Cassandra, MongoDB and RethinkDB but relational systems can be clustered into a distributed system as well. Namely my favorite database engine: PostgreSQL. This is how Instagram does it and they've written extensively about maintaining a PostgreSQL distributed cluster.

The rest of this section is about distributed systems in general, unless otherwise specified.

Your CTO is quite happy that the distributed system chosen by the company can handle both AP and CP, and she wants you to come up with strategies for data coming from the orbital probe that's due to arrive in orbit around Jupiter 8 months from now.

Let's shape our distributed system using sharding and replication.

We will have telemetry data coming in at a very high rate, and we need to process this data continually to be sure our orbital calculations are correct and that our probe is where it's supposed to be.

A fast database can hold most *current data* in RAM. By current data I mean the stuff that we care about. With a demo database, this might come to a few megabytes.

The data generated by most consumer-focused websites remains in the < 1Gb realm, which means that sharding/replication will have little effect.

Another good friend of mine, Rob Sullivan is a PostgreSQL DBA who fields some very interesting questions from developers he runs into at conferences and cafes. Recently he was asked how he would suggest sharding a database system that just hit 5Gb total data.

His answer (paraphrased):

... they were trying to do all of this on the cheapest Heroku database possible, and having issues because they didn't want to pay for the appropriate tier... talking themselves into a complete rearchitecture because of a price list...

There is, and will always be, confusion about when and mostly if you should shard your database.

The outcome: when you run out of RAM and it's cheaper to add a machine vs. scaling up to a bigger VM or server.

Let's take a closer look at this.

Here's a price breakdown for Digital Ocean as of summer, 2016:

Choose a size

\$5/mo <small>\$0.007/hour</small> 512 MB / 1 CPU 20 GB SSD Disk 1000 GB Transfer	\$10/mo <small>\$0.015/hour</small> 1 GB / 1 CPU 30 GB SSD Disk 2 TB Transfer	\$20/mo <small>\$0.030/hour</small> 2 GB / 2 CPUs 40 GB SSD Disk 3 TB Transfer	\$40/mo <small>\$0.060/hour</small> 4 GB / 2 CPUs 60 GB SSD Disk 4 TB Transfer	\$80/mo <small>\$0.119/hour</small> 8 GB / 4 CPUs 80 GB SSD Disk 5 TB Transfer	\$160/mo <small>\$0.238/hour</small> 16 GB / 8 CPUs 160 GB SSD Disk 6 TB Transfer
\$320/mo <small>\$0.476/hour</small> 32 GB / 12 CPUs 320 GB SSD Disk 7 TB Transfer	\$480/mo <small>\$0.714/hour</small> 48 GB / 16 CPUs 480 GB SSD Disk 8 TB Transfer	\$640/mo <small>\$0.952/hour</small> 64 GB / 20 CPUs 640 GB SSD Disk 9 TB Transfer			

The price of each machine goes up according to the RAM. Double the RAM, double the price. So here's the question: *would bumping to the top instance they have (\$640/month) be the same as 4 of the \$160/month?*

In short: **no**. A single machine is simply easier to maintain, especially when it comes to databases. You're much better off just upgrading until you can't – let's see why.

Sharding

The top-of-the-line DigitalOcean VM has 64G of RAM, most of which you can probably use for your database cache. You also have 20 CPU cores at your disposal: this is a fast machine.

Unfortunately, the telemetry data is projected to have a current data size of about 250Gb – this is data we'll need to actively write and query extremely quickly.

We don't have access to a machine with this much RAM at our provider, so we'll need to distribute the load across multiple machines. In other words: shard.

There are two ways we can do this with modern databases:

- **Let the machine decide how.** Modern databases can use the primary keys to divide the data into

equal chunks. In our case, we might decide to go with 6 total shards – so our database system will divide the primary keys into groups of 6 based on some algorithm

- **We decide how.** Our telemetry data might lend itself to a natural segregation in the same way a CRM system might divide customers by region or country – our telemetry data could be divided by solar distance, attitude, approach vector, etc. If the majority of our calculations only need a subset of this data, sharding it logically might make more sense.

The less we mess with a sharding situation, the better – so I would suggest letting the machine decide unless you're utterly positive your sharding strategy won't change.

Once you've decided the strategy, it's a matter of implementing it. Some systems make this very easy for you (RethinkDB, for instance) and others require a bit of work (PostgreSQL requires an extension and some configuration, MongoDB is a bit more manual as well).

If everything works well, you should be able to up A – your throughput and processing capacity – dramatically, at the price of C, consistency.

Other data, however, needs some additional guarantees in a distributed system.

Replication

We're receiving and processing telemetry data constantly, and sometimes it might cause us to alter the current mission plan just a little.

Given that we're using a distributed system, we need to be sure that the mission plan the probe is following is up-to-

the-minute and guaranteed to be as accurate as possible – even if there is a massive power outage at our data center.

So, we replicate the data across the nodes in our cluster.

We have data centers in the US, Europe, Asia and the Middle East – all of which can communicate with the probe throughout the daily rotation of the Earth. This data doesn't change all that often – perhaps a few hundred times per day – so we can implement replication at the database level.

Whenever data changes in our replicated tables, we can guarantee that:

- The data will, at some point, propagate if a netsplit occurs. In other words: if we write to a node in the US and our US datacenter goes down, the write will happen eventually when the US datacenter comes back online.
- The system will compensate and rebalance for the loss of a node, ensuring the data will be as consistent (and available) as possible.

Big Data

Most applications generate low to moderate amounts of data, depending on what needs to be tracked. For instance: with Tekpub (my former company), I sold video tutorials for a 5 year period. The total size of my database (a SQL dump file) was just over 4Mb.

Developers often over estimate how much data their application will generate. My friend Karl Seguin has a great quote on this:

I do feel that some developers have lost touch with how little space data can take. The Complete Works of William Shakespeare takes roughly 5.5MB of storage.

A megabyte (1 million bytes) seems so small, doesn't it?

A gigabyte (a billion bytes) seems kind of skimpy for RAM and we all want a few terabytes on our hard drives. What do these numbers really mean outside of computers? Consider this:

- A million seconds is almost 12 days
- A billion seconds is just over 31 years
- A trillion seconds is 317 centuries Many people simply do not grasp just how much bigger a terabyte is vs. a gigabyte. Let's do this again with inches:
 - A million inches is almost 16 miles
 - A billion inches is almost 16,000 miles, or a little over half way around the earth
 - A trillion inches is 16 million miles, or 631 trips around the planet When considering the data generated by your application, it's important to keep these scales in the back of your mind.

In 2006, Ancestry.com added every census record for the United States, from 1790 to 1930.

The project added 540 million names, increasing the company's genealogical database to 600 terabytes of data.

If you had an extra \$32,000 lying around after your seed round of funding, you could buy 15 x 40Tb drives and store all of this information in your closet today. In 2006 this amount of data was quite impressive and would have cost a fortune. Today ... not so much. Don't get me wrong: \$32,000 is a lot of money but it's pocket change for big companies needing to store tons of data.

In 2013 Information Week wrote an article about Ancestry.com and how it stores its data. This, friends, is a massive growth in data:

A little over a year ago [2012], Ancestry was managing about 4 petabytes of data, including more than 40,000 record collections with birth, census, death,

immigration, and military documents, as well as photos, DNA test results, and other info. Today the collection has quintupled to more than 200,000 records, and Ancestry's data stockpile has soared from 4 petabytes to 10 petabytes.

A petabyte is 1000 terabytes – or 1000 trillion bytes of data. If we translate that into seconds it would be almost 32 million years.

Computer Weekly wrote a fascinating article on visualizing the petabyte, with some amazing quotes from industry experts:

... Michael Chui, principal at McKinsey says that the US Library of Congress “had collected 235 terabytes of data by April 2011 and a petabyte is more than four times that.”

Wes Biggs, chief technology officer at Adfonic, ventures the following more grounded measures... One petabyte is enough to store the DNA of the entire population of the US – and then clone them, twice.

Data analysts at Deloitte Analytics also put on their thinking caps to come up with the following... Estimates of the number of cells in a human body vary, but most put the number at approaching 100 trillion, so if one bit is equivalent to a cell, then you'd get enough cells in a petabyte for 90 people – the rugby teams of the Six Nations.

A petabyte is **huge**. You might be wondering why I'm

throwing these statistics at you? It has to do with the title of this chapter.

What Is Big Data?

Social media is driving the idea of Big Data:

Big data is a term for data sets that are so large or complex that traditional data processing applications are inadequate. Challenges include analysis, capture, data curation, search, sharing, storage, transfer, visualization, querying, updating and information privacy. The term often refers simply to the use of predictive analytics, user behavior analytics, or certain other advanced data analytics methods that extract value from data, and seldom to a particular size of data set.

It's a buzzword, sure, but there is meaning behind it. Companies like Google, Facebook and Twitter are generating gigantic amounts of data daily. Back in 2008 Google was processing over 20 petabytes of data per day:

Google currently processes over 20 petabytes of data per day through an average of 100,000 MapReduce jobs spread across its massive computing clusters. The average MapReduce job ran across approximately 400 machines in September 2007, crunching approximately 11,000 machine years in a single month.

While researching this chapter I stumbled on an inter-

esting post from FollowTheData.com, which outlined how much data was processed by certain organizations daily back in 2014:

- The US National Security Administration (NSA) collects 29 petabytes per day
- Google collects 100 petabytes per day
- Facebook collects 600 petabytes per day
- Twitter collects 100 petabytes per day
- For data storage, the same article states:
- Google stores 15,000 petabytes of data, or 15 exabytes
- The NSA stores 10,000 petabytes
- Facebook stores 300 petabytes

These numbers are rough estimates, of course. No one knows about Google's storage capabilities outside of Google, but Randall Munroe (of xkcd fame) decided to try to deduce how much data Google could store for one of his *What If?* articles using metrics like data center size, money spent, and power usage:

Google almost certainly has more data storage capacity than any other organization on Earth... Google is very secretive about its operations, so it's hard to say for sure. There are only a handful of organizations who might plausibly have more storage capacity or a larger server infrastructure.

A fascinating read. Please take a second and have a look – but be warned! You will likely get lost in all the *What If?* posts.

Simply put relational systems are just not up to the task, for the most part. The reason for this is a simple one: processing this data needs to be done in parallel, with multiple machines churning over the vast amounts of data. This is the most reliable way to scale a system like Google's that generates gigantic quantities of data every day: just add another data center.

How do you process this kind of information in parallel, however? This is where systems like Hadoop come in:

Hadoop is an open-source software framework for storing data and running applications on clusters of commodity hardware. It provides massive storage for any kind of data, enormous processing power and the ability to handle virtually limitless concurrent tasks or jobs.

Hadoop was born from efforts at Yahoo!, and then turned into an open-source project that any organization can download and install.

Hadoop partitions your data using its dedicated file system, HDFS, which is based on Java. When you query data, you use Map/Reduce.

In The Real World

I worked at a business analytics company for about 4 years, working with companies who wanted us to sift through their warranty claims information, looking for patterns. This involved natural language processing (NLP), where we split claim information into sentence structures and then ran various algorithms over it.

It was fun, but 90% or more of my work was trying to figure out which data were good. Emails, phone calls, sifting through and correcting millions upon millions of records ... it's a lot of work.

Even then, I wouldn't call that Big Data. That was just basic analysis over a large set of data. As Sean Parker's character said in *The Social Network*:

A million dollars isn't cool, you know what's cool? ... A billion dollars.

These days a billion records of anything doesn't even mean much. Terabytes and ... yeah you're getting there. You know what's cool? A *petabyte* is cool.

SHELL SCRIPTING

FLEXING BASH AND UNIX TO INCREASE YOUR
PRODUCTIVITY

Unix and Unix-like systems (Linux, BSD, Solaris, RedHat, etc.) have been around forever. You simply can't expect to grow much in your career if you don't have a basic competency with Unix and its commands. While this kind of thing isn't (typically) a part of a CS curriculum, it is something you pick up along the way in university.

If you don't believe me, skip right over this chapter. It'll be here when you come back, after you've realized just how true this is.

This is an exciting thing! Crawling under the hood of your computer can increase your efficiency dramatically. Shell scripts, Makefiles, server setup routines, quick little commands to update your system, configuring your web/database server remotely over SSH ... these are skills you must know.

So let's wander through the shell. I won't go into Unix history as I'm just not qualified to do so. I'll also sidestep the basics of the Unix commands – that'll be up to you.

Instead, let's get right to the thing that will help you the most in your job: basic shell scripting skills.

You can find the code used in parts of this section up at my GitHub repo. In addition, you can buy 34 video walkthroughs of the code you see in this chapter and others from here.

What Is a Shell?

A computer needs a way to receive data, and we're going to do that through the command line using a thing called a shell. The first computer ever conceived used punch cards to receive data, when I was in high school, I used a combination of a keyboard as well as a cassette tape player to boot my computer!

Today we have visual interfaces that look quite juicy and convey information in a friendly way. We use mice to issue commands (most of the time) and, occasionally, our fingers or a stylus.

During the 1960s through 1980s, computer users entered their commands as text from a keyboard. This practice has continued today and is what you're about to do, using the command line interface.

All these things are shells. A shell is simply a generalized way in which you give commands to a computer and receive the output. A visual shell uses a graphical interface, or a GUI, and is what I'm using now to type this sentence on my Mac, using a visual editor.

A text-based shell has no visuals except for things you can do with ASCII symbols. To work with a text-based shell (like Bash, for instance), you use a command line interface, or CLI.

There are several shells that you can work with, so far

we've discussed two: Powershell and Bash. You can install other ones, if you like, including:

- Z shell (or zsh). I like this one a lot and it's what I use every day together with Oh-My-Zsh from Robby Russell
- Fish. They win for the best tag line: "Finally, a command line shell for the 90s"
- Tcsh (or "tc shell"). This is a common one you see on many Unix machines

Why The Name "shell"?

At this point you might be wondering why these things are called "shells". It has to do with the way Unix is constructed. There is a kernel that does all the processing, which is protected by several "protection rings". Each ring provides certain services, with the most sensitive being closer to the kernel and the least being on the very edge, or "shell" of the system. I won't go into Unix design at this point (mostly because I'm not qualified to); but I find that an interesting way to think about Unix.

If you look around, you'll find many shells that look interesting. Bash works well for most things, but if you're looking for something a bit more friendly than I might recommend having a look at Z Shell. I've been using it for years and love it. One main reason is that it has helpful completions, spelling corrections, and you can program the prompt to be colorful and pretty.

The biggest reason, however, is the Oh-My-Zsh project, mentioned above. You get a sane way to organize scripts, aliases and other things. Here's their project description:

A delightful community-driven (with 1,000+ contributors) framework for managing your zsh configuration. Includes 200+ optional plugins (rails, git, OSX, hub, capistrano, brew, ant, php, python, etc.), over 140 themes to spice up your morning, and an auto-update tool so that makes it easy to keep up with the latest updates from the community. <http://ohmyz.sh/>

It's been very useful for me.

Keeping Shell Stuff Organized

This is kind of a big deal. As you learn to work with the shell more and more, you find good organization becomes important. Oh-My-Zsh can help with most things, but not everything.

For example: settings. If you ever work with Vim, Git, Atom, AWS, etc. – you know they have various startup settings in an rc file somewhere. These files (typically ending in “rc”) ¹need to live somewhere. This is where strong organizational skills will help a lot.

Dot Files

Most Unix-adept developers treat their scripts and settings with the same respect as any other code: organizing it carefully and versioning it with git. For example, one of my very favorite people is Gary Bernhardt and he keeps his dot files on GitHub.

You'll hear the term “dot files” a lot – and you'll see them a lot. It's a convention to begin a file name with a period (or “dot”) to hide it from the finder and from the standard listing

command, `ls`. These files are shell scripts that are read in by various programs and they contain settings of all kinds.

One that a lot of people obsess on (including yours truly) is `.vimrc`. In this file you will find settings for Vim. It's a shell script that's executed every time Vim starts.

Think about developers you follow on Twitter and see if they have a dot files repository on GitHub. I already mentioned Gary Bernhardt - here's another one of my favorite people: Ryan Bates. Have a look at how they organize these files and what's in there. You could lose hours on this!

In case you haven't figured it yet, "rc files" are simply shell scripts that are executed by a program when it starts. They're simply text files that are, typically, well-commented and allow you to change how a program behaves.

OK, so now we understand how programs use shell scripts to configure themselves. How can you use them to help with what you do every day?

Why Script a Shell?

Think about the project you're working on right now and the tasks you need to perform on a routine basis. Here are some that I do when building web sites with Node, Ruby, or Elixir:

- Navigate to a project
- Open up the project in an editor of some kind
- Work with a source control system, something like Git perhaps
- Work with a database, something like PostgreSQL, MongoDB, etc.
- Write a blog post, perhaps
- Lint/concatenate/minify/compile your code files (CSS, JavaScript, whatever)

Sure, there are plenty of tools that can do these tasks graphically for you. Code editors have dialogs for opening certain directories, database GUI tools can help you write queries, and there are plenty of tools out there for graphically working with Git. These tools look nice, but they're horribly slow when compared to their command line (CLI) counterparts.

Every task you and I do daily can be done faster with a CLI tool. You can type much faster than you can visualize/click. We could argue that point, I suppose. I have some good friends who work in Visual Studio (Microsoft's .NET IDE) and with the purchase of a few plugins they can write code rather quickly and have gained a decent level of efficiency.

It still doesn't come close to what you can do using shell scripts.

Let's say your boss comes in and asks you to make sure you have a database backup setup on a nightly basis that zips and loads the file to your Amazon S3 bucket. How would you do this with your favorite database GUI tools? This is a 20-line shell script.

You need to lint, concatenate and minify your JS files in a very particular way, using the rules your development lead has set for the team. In addition, you need to create a warning when the build size exceeds 100K. This is a 15-line Makefile.

You have a new marketing manager who has decreed that your company site needs to load faster – your current YSlow Grade is a D. It's decided that the 1200 JPEG files your site serves need to be optimized and reduced in size to no more than 600 pixels wide. This can be done in a 30-line shell script.

I know what you're thinking: *I can do all of this from the shell using Ruby/Python/JavaScript – why do I care about your shell*

scripts? It's a good question, and a fair point. My response to that would be ... how many packages and supporting files are you going to need? How long will it take, as they say, to "shave that Yak"?

This is the great thing about shell scripts: once you know them, you know them. It's one of those skills that you can use to do ... just about anything.

A Simple Shell Script

Your company website has quite a few images; some of them rather large. Much larger than they should be. A new marketing director was just hired and found out the site is ridiculously slow to load, and has decided that these images are to blame. In short: *you have an image problem*.

Your boss has tasked you with auditing the images and then resizing them. What fun! Isn't this why you became a programmer? The very first thing she's asked for is a list of all the images in our site's directory. That will be our first task.

In the downloads for this section, you'll find a directory called "images". You can use that directory to work on.

Let's crack open our terminal. On a Mac, this is (most likely) going to be Terminal.app, which you can find in /Applications/Utility. Or you can get your keyboard skills on by typing CMD-Space to bring up Spotlight, then type "Terminal".

It will open in your home directory, or **\$HOME** in Unix land. To navigate around you can use **cd** to change directories – just use the name of the directory you want to go to. If you want to go back one, you can use **cd ..**; if you want go all the way back to **\$HOME** you can just type **cd** followed by **<Enter>**.

Let's assume you downloaded the image files to your

Desktop. For simplicity, let's create a directory in our **\$HOME** called "imposter", and then another inside that one called "demos". In your terminal, type **mkdir -p imposter/demos**.

This command will create a directory set in your **\$HOME**. The **-p** flag tells **mkdir** to create the entire structure if it's not already there.

Nice work, now let's move our demo files in there, and then change into that directory:

```
mv ~/Desktop/task-images ~/imposter/demos
cd imposter/demos/task-images
```

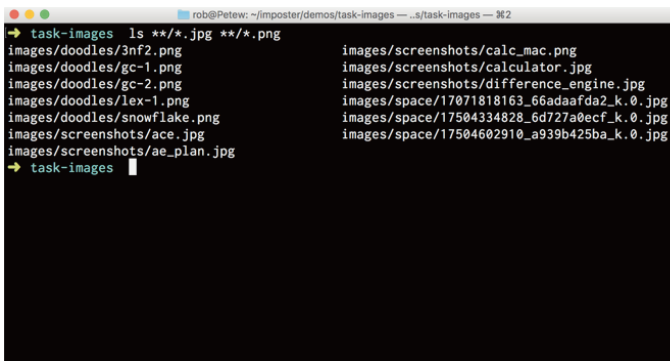
The command **mv** will move files and directories around on your machine and **cd** will change directory, which I'm sure you could reason out.

Is all this typing getting you down? Bash and many other shells support command completion using **TAB**. Try it! It really helps when navigating around your machine.

Now that we're here, let's list out the images. You can list files with the **ls** command, but you can also restrict it with what's known as a glob. You can think of this as a series of wildcards:

```
ls **/*.jpg **/*.png
```

This line right here says "list out the jpg and png files, any name, any directory". Your output should look something like this:



```

rob@Petew: ~/imposter/demos/task-images -- s/task-images -- 22
task-images ls **/*.jpg **/*.png
images/doodles/3nf2.png          images/screenshots/calc_mac.png
images/doodles/gc-1.png          images/screenshots/calculator.jpg
images/doodles/gc-2.png          images/screenshots/difference_engine.jpg
images/doodles/lex-1.png         images/space/17071818163_66adaafda2_k.0.jpg
images/doodles/snowflake.png     images/space/17504334828_6d727a0ecf_k.0.jpg
images/screenshots/ace.jpg       images/space/17504602910_a939b425ba_k.0.jpg
task-images

```

If this isn't what you're seeing, make sure you're in the correct directory. Also, be sure you entered the glob correctly as well.

OK, we're almost done. What we need to do now is to create a list that we can show our boss. To do that, we'll redirect the output of the command into a text file:

```
ls **/*.jpg **/*.png > images.txt
```

And we're done! If you want to see this file, you can use the command `open images.txt`, and you'll see them in your default text editor.

That wasn't so bad, was it? That one line saved us quite a bit of work, don't you think? How would you have done this using visual tools?

I just threw a lot at you, but I'm sure it wasn't that difficult. There are two things I want to highlight, however.

I was using the term **\$HOME** a lot. This is a special place on a Unix machine – it's where you get to do whatever you want. Visually speaking, you can think of **\$HOME** as the place the Finder opens up to when you first open it. It's usually a place like `/Users/rob` (in my case). You don't ever

work on the root of the machine – that's only for special users which we'll discuss later.

Look at your **\$HOME**. You can do this using the command **echo \$HOME**. The **echo** command simply outputs a value to the screen, in this case it will be whatever the **\$HOME** variable is set to. That's right – **\$HOME** is a variable, and a special one at that. It's called an "environmental variable" and there are many them. You can tell you're working with a variable in Unix because they have a **\$** prepended to them (this is parameter expansion, which I'll get into below). Other variables include **\$PATH** and **\$USER**.

We'll be working with variables of our own making later on.

The next thing I mentioned (but kind of glossed over) was that I redirected the output to a text file. I did this using the **>** operator. This is a crucial thing to understand when working with the shell: there is a standard input and standard output. The standard input is the keyboard, the standard output is the terminal.

In the same way you can refer to **\$HOME**, you can refer to standard output as **STDOUT** and standard input as **STDIN**. This might seem a bit academic at this point, but if you think about working with a computer, in general, you give it information and it gives you something back. It does this with **STDOUT** and **STDIN**.

You wouldn't want to have to specify where you want the output sent every time you executed a command, would you? This is where **STDOUT** comes in. If you did want the output of a program to go somewhere, it's easy to specify. Which is what we did using **>**.

When you're working in a shell you're working in a REPL (Read, Eval, Print Loop). If you don't know what that is – it's a way of working directly with a language. If you have Node installed on your machine you can type in "node" and you'll

be in the Node REPL. From here you can enter all kinds of JavaScript code.

If you have Ruby installed on your machine you can enter “`irb`” in the terminal and you’ll be in the Ruby REPL. The Unix command line is the same thing. Bash (or Z shell or whatever shell you’re using) will expand and then execute the commands you give it, executing them directly. We’ll get into command expansion more later on; for now let’s keep rolling.

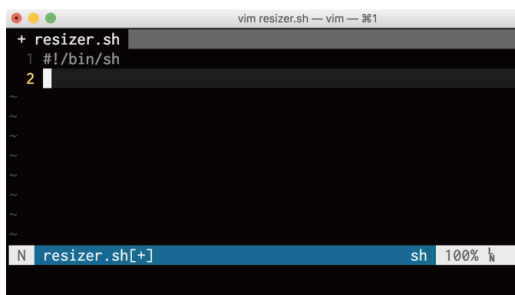
Create a new file called “`resizer.sh`” and open it up in Vim. If you’re not a Vim fan, use whatever editor you like – but I would challenge you to just give it a try, at least for this walk-through.

You’ll want to be sure you’re in the same directory as before – the one with the “`images`” subdirectory. Then, open up Vim, passing it the file name you want:

```
vim resizer.sh
```

This will create a buffer in Vim, not an actual file. That won’t be created until we save the file.

The first thing we need to enter is a *shebang*, which is a great word don’t you think? It’s also called a *hashbang* by some. To do this in Vim, enter “`i`” to go into “insert mode” (so you can type some text) and then enter this at the top:



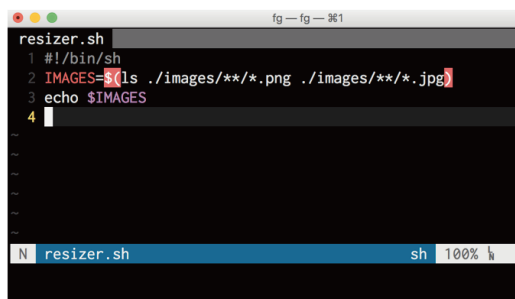
```
vim resizer.sh -- vim -- %1
+ resizer.sh
1 #!/bin/sh
2 |
N resizer.sh[+] sh 100%
```

When you're done typing, hit the ESCAPE key (<ESC>). This will return you to "normal mode".

Line #1 above is our shebang, it tells the shell what interpreter we want to use to run this script in the form of an absolute path. In this case, the interpreter is the shell itself, which uses the "sh" program to read in commands from the keyboard, a file, or **STDIN**.

The next thing we'll do is assign our image files to a variable. To make sure we've done this right, I'll output the result to the screen in the same way I might use **console.log** in JavaScript or **puts** with Ruby.

Your cursor should still be on line 1. If it is, enter "o" to create a new line and enter insert mode. You should be on line #2, where you can enter the following:



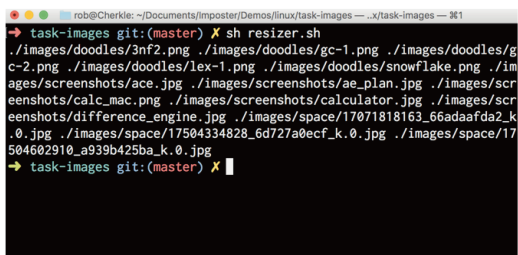
```
fg -- fg -- %1
resizer.sh
1 #!/bin/sh
2 IMAGES=$(ls ./images/**/*.png ./images/**/*.jpg)
3 echo $IMAGES
4 |
N resizer.sh sh 100%
```

As with last time, make sure you hit <ESC> to go back

to normal mode. The next thing we need to do is save the file, and you can do that by entering “:w”, which means “write this buffer to disk”.

Let’s run it to be sure everything works, and then we’ll get to the explanation. Enter **CTRL-Z** to suspend the Vim session, flipping back over to the terminal. If you can’t get this to work for some reason, just open up a second terminal window and navigate to the same directory you’ve been working in – make sure you’re in the shell, not Vim. Our goal is to have the shell execute what we’ve just written.

Enter “**sh resizer.sh**” into the terminal and you should see an amazing splash of text:



```

rob@Cherlie: ~/Documents/Imposter/Demos/linux/task-images — ..x/task-images — 961
→ task-images git:(master) ✗ sh resizer.sh
./images/doodles/3nf2.png ./images/doodles/gc-1.png ./images/doodles/g
c-2.png ./images/doodles/lex-1.png ./images/doodles/snowflake.png ./im
ages/screenshots/ace.jpg ./images/screenshots/ae_plan.jpg ./images/scr
eenshots/calc_mac.png ./images/screenshots/calculator.jpg ./images/sr
eenshots/difference_engine.jpg ./images/space/17071818163_66adaafda2_k
.0.jpg ./images/space/17504334828_6d727a0ecf_k.0.jpg ./images/space/17
504602910_a939b425ba_k.0.jpg
→ task-images git:(master) ✗

```

It worked! Or did it? We used the **sh** command, which feeds a file to the shell, the contents of which are expanded and executed. That’s kind of like executing it, but not really. It’s a bit like using **eval** in Ruby or JavaScript to run a string of code, rather than executing it directly.

To properly execute a shell script, you just invoke it. Try entering “./resizer.sh” directly. This command simply gives the exact location of the shell script file, with the leading “./” indicating “this directory”. Doing this should lead to some problems:

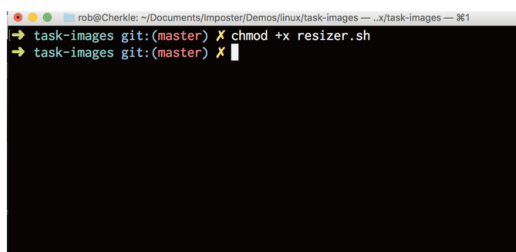
A terminal window with a black background and white text. The window title is 'rob@Cherkie: ~/Documents/imposter/Demos/linux/task-images -- ./x/task-images -- 第1'. The prompt is 'task-images git:(master) X'. The user enters './resizer.sh'. The response is 'zsh: permission denied: ./resizer.sh'. The prompt returns to 'task-images git:(master) X' with a cursor.

```
rob@Cherkie: ~/Documents/imposter/Demos/linux/task-images -- ./x/task-images -- 第1
task-images git:(master) X ./resizer.sh
zsh: permission denied: ./resizer.sh
task-images git:(master) X
```

By default, you can't execute a file directly in the shell unless you specifically say it's OK to do so. This is a security feature of Unix, as you might imagine.

To grant execution permissions you need to tell the operating system, and you do that by using the **chmod** command (change file mode). By the way, if you ever want to know more about any of the commands you see here, you can use **man** in the terminal itself. This shows the “manual” for each command. Try it now – enter “man chmod” and it will tell you all about it.

For our needs, I need to **chmod +x** our **resizer**, which means “add execute privileges to this file:

A terminal window with a black background and white text. The window title is 'rob@Cherkie: ~/Documents/imposter/Demos/linux/task-images -- ./x/task-images -- 第1'. The prompt is 'task-images git:(master) X'. The user enters 'chmod +x resizer.sh'. The prompt returns to 'task-images git:(master) X' with a cursor.

```
rob@Cherkie: ~/Documents/imposter/Demos/linux/task-images -- ./x/task-images -- 第1
task-images git:(master) X chmod +x resizer.sh
task-images git:(master) X
```

One thing to get used to with Unix: commands will typically not return any kind of result if they are successful. Silence, in this case, means all went well. Now we should be able to execute our script:

```

rob@Cherlie: ~/Documents/Imposter/Demos/linux/task-images — .x/task-images — 361
➔ task-images git:(master) ✗ chmod +x resizer.sh
➔ task-images git:(master) ✗ ./resizer.sh
./images/doodles/3nf2.png ./images/doodles/gc-1.png ./images/doodles/g
c-2.png ./images/doodles/lex-1.png ./images/doodles/snowflake.png ./im
ages/screenshots/ace.jpg ./images/screenshots/ae_plan.jpg ./images/scr
eenshots/calc_mac.png ./images/screenshots/calculator.jpg ./images/scr
eenshots/difference_engine.jpg ./images/space/17071818163_66adaafda2_k
.0.jpg ./images/space/17504334828_6d727a0ecf_k.0.jpg ./images/space/17
504602910_a939b425ba_k.0.jpg
➔ task-images git:(master) ✗

```

Nice work! Now let's dive into the code some. To get back over to Vim, enter “fg” in the terminal to bring up the suspended app (fg means “foreground”). If you get a warning about the file being modified, just enter “I” to load it anyway. You should see this now:

```

fg — fg — 361
resizer.sh
1 #!/bin/sh
2 IMAGES=$(ls ./images/**/*.png ./images/**/*.jpg)
3 echo $IMAGES
4
N resizer.sh sh 100%

```

We understand the first line, which is our shebang, but the second line looks a tad cryptic. Here, we're setting a variable called **IMAGES** to the result of our image listing ... but what's that syntax?

When you surround a command with a **\$()** it's called a *subshell*. As you can probably reason, I need to set the **IMAGES** variable to the result of the list command, which means I need to invoke it in place. I can do that by wrapping it in a subshell. This subshell will be expanded, and the results returned to the **IMAGES** variable.

We'll do more with subshells in a later section; for now,

you can think of it as invoking a command in place and using its results directly.

The next line uses the `echo` command to output the value of the **IMAGES** variable to the screen. To use a variable (which should always be upper-cased), you must expand it as well using the `$`. This is called *parameter expansion* and might seem a little weird until you have a play with it.

Open a new terminal window and type in **THING=1**. This will set the variable **THING** to the value 1. Now let's use `echo` one more time to have a look at this value. Try entering `echo THING`.

What happened? The `echo` command doesn't know if you're giving a literal value or a variable – it's up to you to expand the value before `echo` gets ahold of it. You do this in the same way you expand a subshell: using `$`.

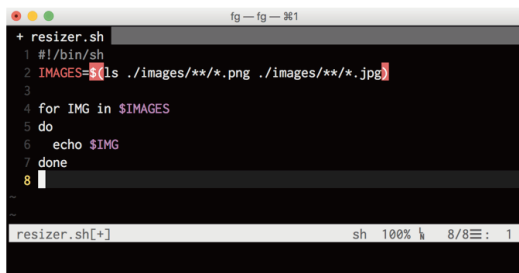
There are different ways to run subshells and expansions, and we'll get into that in a later section.

We have our list of images and, at some point, we'll need to operate on those images individually. This means we'll need some kind of loop – a **for** loop specifically.

We can do this with our shell. You should be in normal mode with Vim; if you're not, just hit `<ESC>` until you are. You can move around the screen using the `h`, `j`, `k` and `l` keys. Give it a try, see what happens.

It helps me to think of the “`j`” as an anchor, pulling down and the “`k`” as a rock climber, clinging to the face of a vertical wall. Once you're on line #3, enter “`dd`”, which will delete the line.

Now, enter “`i`” to get back to insert mode. We need to type some more code:



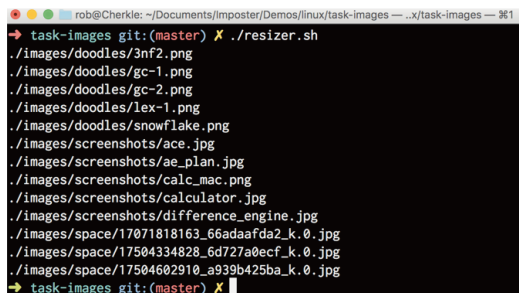
```

+ resizer.sh
1 #!/bin/sh
2 IMAGES=$(ls ./images/**/*.png ./images/**/*.jpg)
3
4 for IMG in $IMAGES
5 do
6   echo $IMG
7 done
8
resizer.sh[+] sh 100% 8/8 1

```

This is our `for` loop. We declare a variable inline (**IMG**) and then create a **do** block, ending it with **done**. Inside this block we'll report the value of the **IMG** variable. Hit `<ESC>` when you're done entering this code, then type `“:w”` to save it. Now `CTRL-Z` to suspend and flip back to the terminal.

Execute again, invoking the file directly (or hit the up arrow until you see it):



```

rob@Cherkie: ~/Documents/Imposter/Demos/linux/task-images -- x/task-images -- 1
task-images git:(master) X ./resizer.sh
./images/doodles/3nf2.png
./images/doodles/gc-1.png
./images/doodles/gc-2.png
./images/doodles/lex-1.png
./images/doodles/snowflake.png
./images/screenshots/ace.jpg
./images/screenshots/ae_plan.jpg
./images/screenshots/calc_mac.png
./images/screenshots/calculator.jpg
./images/screenshots/difference_engine.jpg
./images/space/17071818163_66adaafda2_k.0.jpg
./images/space/17504334828_6d727a0ecf_k.0.jpg
./images/space/17504602910_a939b425ba_k.0.jpg
task-images git:(master) X

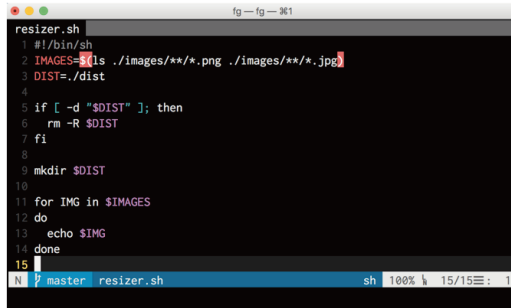
```

Nice! OK, let's keep rolling and pick up some speed.

We don't want to blow away our original files, so we'll create a destination directory, where all the modified files will be placed. We need to check if this directory exists before we do anything. If it does, we'll delete it, otherwise we'll just create it.

Use `“fg”` again to get back into Vim and then navigate to

line #2, where the **IMAGES** variable is declared. Now enter “o” to open a new line below it. Enter the following:



```

resizer.sh
1 #!/bin/sh
2 IMAGES=$(ls ./images/**/*.png ./images/**/*.jpg)
3 DIST=./dist
4
5 if [ -d "$DIST" ]; then
6   rm -R $DIST
7 fi
8
9 mkdir $DIST
10
11 for IMG in $IMAGES
12 do
13   echo $IMG
14 done
15
master resizer.sh sh 100% 15/15 1

```

The first thing is the most important: use variables for everything. It’s bad form to hardcode values in a shell script! Here, I’m simply specifying where the output directory is going to be.

On line #5 I’m testing to see if this directory exists. Notice the spacing here? It’s important! Make sure there’s a single space between the brackets and the conditional statement. Also: notice the semicolon? That’s optional – it signifies a code line termination. If I put them on a new line the semicolon isn’t needed. I use it here because my eyes are used to reading **if** statements in this way.

Next: notice that I wrapped **\$DIST** in quotes? This is a subtle point and not one I expect you to remember entirely. If you write many scripts, however, you’ll have a very interesting problem to overcome: *how do I control this expansion thing?*

We’re using the command line, which is text-based, so Bash will take you literally when you type in anything. We need a way to work with text in this environment.

For instance, let’s say I create a file with a silly file name:

```
touch super-*.txt
ls super-*. *
ls: super-*. *: No such file or directory
```

Now this file name and, indeed, this entire example is ridiculous ... sort of. I've seen some amazingly weird file names! Anyway: this command set will cause an error. We're confusing our shell because it can't tell the difference between our literal text **super-*** and our wildcard placeholder **.***. How do we get around this problem?

To “unconfuse” the shell we can use quoting:

```
ls 'super-*. *'
super-*.txt
```

I'm using a single quote here, which is referred to as “strong quoting”, which means absolutely nothing within the string will have any special meaning to Bash. For our needs this is actually a bit too much (or I should say too literal). We want to have some expansion in there!

For instance, we might want to set the **\$DIST** directory to be something like **\$HOME/fixed** or something. We want **\$HOME** (and other variables) to be expanded in this context.

We can do this by using “weak quoting”, or double-quotes, which will allow us to access variables using **\$** (we can do other things too, like use a **\(\sim \)** for our home directory and **** to escape things).

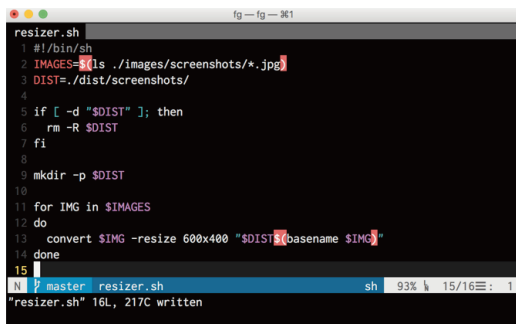
It's good practice to use quoting when referencing variables as we're doing here. If any of our directories or files contain characters that will confuse our shell, their expansion will be ignored.

OK, we're almost done – let's rock this out!

To run the actual image conversion, I'll use ImageMagick, a popular open-source image manipulation tool. You can install it on your Mac using Homebrew ("brew install image-magick") or with MacPorts.

Once ImageMagick is installed, I simply need to use the `resize` command with some dimensions and an output.

We only need to resize our screenshots, so I'll reset the `$DIST` variable to only include them (I'll change this later). Finally, I just call `convert` within our loop, and off we go:



```

resizer.sh
1 #!/bin/sh
2 IMAGES=$(ls ./images/screenshots/*.jpg)
3 DIST=./dist/screenshots/
4
5 if [ -d "$DIST" ]; then
6   rm -R $DIST
7 fi
8
9 mkdir -p $DIST
10
11 for IMG in $IMAGES
12 do
13   convert $IMG -resize 600x400 "$DIST$(basename $IMG)"
14 done
15
16 master resizer.sh
"resizer.sh" 16L, 217C written

```

If we execute this script as before, we'll have a lovely new set of resized images waiting for us in the `./dist/screenshots` directory. Not bad for 15 lines of code!

But we can do better.

Being the good coder that you are, you're probably wondering why I hardcoded everything for the "screenshots" directory? Good for you!

What we have here is a very workable shell script, but it could be better. Just like any code that you write, think about modularity and reuse. In our case we have a lovely bit of functionality that I'll probably want to use later. The good news is that I can do this by turning it into a function:

```

vim resizer.sh — vim — 361
resizer.sh
1 #!/bin/sh
2 function resize_jpegs(){
3   IMAGES=$(ls ./images/screenshots/*.jpg)
4   DIST=./dist/screenshots/
5
6   if [ -d "$DIST" ]; then
7     rm -R $DIST
8   fi
9
10  mkdir -p $DIST
11
12  for IMG in $IMAGES
13  do
14    convert $IMG -resize 600x400 "$DIST$(basename $IMG)"
15  done
16 }
17
18 resize_jpegs
19
20 master resizer.sh sh 100% 19/19
"resizer.sh" 19L, 266C written

```

On line #2 I just declare the function, which looks a bit like JavaScript doesn't it? On line #16 I close it off with a brace and then I can call it directly on line #18.

This is neat, but it's not modular! For that I'll need to send in some arguments. You work with arguments positionally in a shell script, using parameter expansion like we did before with variables.

In this case I'll change "screenshots" to reference the first parameter passed to our function (\$1):

```

fg — fg — 361
resizer.sh
2 function resize_jpegs(){
3   IMAGES=$(ls ./images/$1/*.jpg)
4   DIST=./dist/$1/
5
6   if [ -d "$DIST" ]; then
7     rm -R $DIST
8   fi
9
10  mkdir -p $DIST
11
12  for IMG in $IMAGES
13  do
14    convert $IMG -resize 600x400 "$DIST$(basename $IMG)"
15  done
16 }
17
18 resize_jpegs screenshots
19
20 master resizer.sh sh 100% 19/19

```

Nice! To pass a value into the function you just tack it on to the function call, which you can see on line #18. I've

replaced the hardcoded value with **\$1**, and we're good to go.

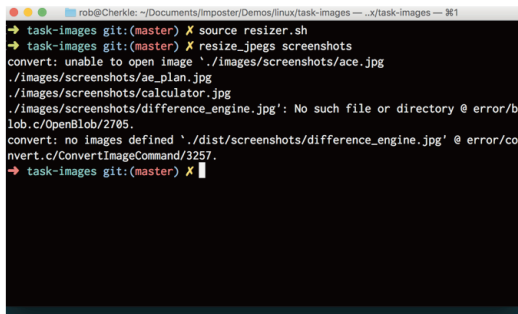
Right now, our script executes every time we call it, which is fine, but we might want this command available to us from anywhere. We can do this if we just tweak a few things.

First, let's get rid of line #18. This will prevent our function from executing each time. Then, instead of invoking our script file, we can just source it:

```
source resizer.sh
```

When you source a file, as we've done here, you load its contents into the current shell. This has the same effect as execution, essentially, but it does a bit more.

For us, we can now call our **resize_jpegs** function from anywhere:



```
rob@Cherlie: ~/Documents/Imposter/Demos/linux/task-images — x/task-images — 391
→ task-images git:(master) X source resizer.sh
→ task-images git:(master) X resize_jpegs screenshots
convert: unable to open image './images/screenshots/ace.jpg'
./images/screenshots/ae_plan.jpg
./images/screenshots/calculator.jpg
./images/screenshots/difference_engine.jpg': No such file or directory @ error/b
lob.c/OpenBlob/2705.
convert: no images defined './dist/screenshots/difference_engine.jpg' @ error/co
nvert.c/ConvertImageCommand/3257.
→ task-images git:(master) X
```

Uh oh. Looks like I was wrong about that. What happened?

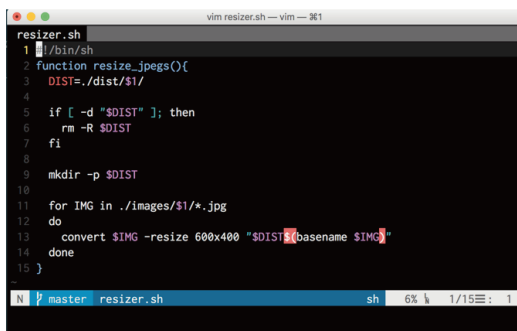
Remember above when I said that executing a shell script is sort of like loading it directly into the shell? That sort of just came back to bite me.

When you execute a shell script, as we did above, it's

given its own process. Our **\$IMAGES** variable executes properly in a subshell and returns a list of files, as it should. We can then loop over that and all is well.

When we source a script file, we load it into the current terminal session and the code we write in our shell script acts just as if we typed it in there ourselves. What this means is that the expansion we're using to load the **IMAGES** variable is viewed as string value. If you read the error output above, you'll see that ImageMagick is trying to convert a file identified by a huge string value!

We don't want that. The good news is that we can fix this easily by just looping over the glob directly:



```

vim resizer.sh -- vim -- 961
resizer.sh
1 !/bin/sh
2 function resize_jpegs(){
3     DIST=./dist/$1/
4
5     if [ -d "$DIST" ]; then
6         rm -R $DIST
7     fi
8
9     mkdir -p $DIST
10
11     for IMG in ./images/$1/*.jpg
12     do
13         convert $IMG -resize 600x400 "$DIST$(basename $IMG)"
14     done
15 }
16
17 N master: resizer.sh sh 6% 1/15 1

```

Confusing, isn't it? You'll run into these problems, and the only way to solve them is to think about which process is executing them: your current shell or a spawned shell?

If you're still confused, hang tight – we're going to do more in the next section. For now you have a usable script! Can you see improvements to be made? What about the sizing? Or the input directory?

Have at it. Playing with shell scripts can be fun, and if you get stuck or need to try something new, there are plentiful resources online.

MAKE

TURNING ONE FILE INTO ANOTHER WITH AN OLD
FRIEND

Make is a build utility that works with a file called a “Makefile” and basic shell scripts. It can be used to orchestrate the output of any project that requires a build phase. It’s part of Linux and it’s easy to use.

It’s important to understand that the utility, make, is not a compiler as many people believe. It’s a build tool just like MSBuild or Ant.

Make is an extraordinarily simple tool which, combined with the power of the shell, can greatly reduce the complexity of your application’s build needs. Even if you’re a Gulp/Grunt/Whatever fan, you should understand the power of make, as well as its shortcomings.

The Basics

Make will turn one (or more) files into another file. That's the whole purpose of the tool. If you run make and your source hasn't changed, make won't build your output.

Make runs on shell commands, orchestrated using the concept of "targets". Let's have a look.

First, create a directory where we can work and then create a Makefile:

```
mkdir make_demo && cd make_demo  
touch Makefile
```

Like many build utilities, a single file with a particular name drives the process. With Grunt, it's a Gruntfile; with Rake it's a Rakefile. Ever wonder where that convention came from? Yep: *it's Make*.

Anatomy of a Makefile

All of our build commands will go into the Makefile. Let's create the basic skeleton:

```
all:  
  
clean:  
  
install:
```

What you see here are targets: the things that Make will try to build for you. By convention they are named based on the file they are building – in some cases, like ours here, they are directives and don't build anything.

Every Makefile should have these directives:

- **all**: builds everything
- **clean**: cleans up all the existing build artifacts; usually deleting the files built

It's typical, as well, to have these directives as well:

- **install**: installs whatever built files are generated
- **.PHONY**: lists out the directives that don't create a file

Have you ever downloaded software from source onto a Linux box and had to run **make** && **make install**? This is why. The downloaded source (usually C or C++) is compiled and then installed wherever binaries go in the system, which can be /opt or somewhere else such as /usr/local or /usr/bin.

With our example we won't be installing anything so we don't need the install target. In its place we'll add another directive that tells Make which of the targets don't create a file. For this we'll use **.PHONY**, for "phony" targets:

```
all:

clean:

.PHONY: all clean
```

Understanding Targets

Make builds output files from input files. It was originally designed for C programs, which utilize both code and header files which are built into object files. These object files are

then compiled to binary. This is a multistep build that requires some orchestration. That's what Make is all about.

Sometimes, however, you'll want a build step that might transform some input – it might not create a file. Letting Make know about these special (phony) targets will increase performance greatly. We'll get more into this in the next chapter.

This is a silly demo, but it's critical you see the way things work before we dive into the good stuff. Let's create a file in the root of our project directory that we want to transform and distribute. We'll leave the file empty for now – let's also create a Makefile while we're at it:

```
echo "//some code" > app.js && touch Makefile
```

The simplest first step would be to copy our app.js code file to a /dist directory. So let's do it:

```
all:
  mkdir -p dist
  cp app.js dist/app.js

clean:
  rm -rf dist

.PHONY: all clean
```

The indents in a Makefile must use tabs. If you don't use tabs you'll get an error about an invalid separator. If you copy/paste the code here, be sure you indent with a tab that doesn't get translated to spaces.

OK, save the Makefile and run make:

```
rob@Cherkle: ~/Documents/imposter/Demos/linux/make -- .os/linux/make -- %1
→ make git:(master) X make
mkdir -p dist
cp app.js dist/app.js
→ make git:(master) X
```

Yes! Make outputs each command as it's executed – so here it's reporting that it ran `mkdir` and `cp` successfully. This can be good ... it can also be annoying.

Let's fix the chatter and provide a clean target while we're at it:

```
all:
    @ mkdir -p dist
    @ cp app.js dist/app.js

clean:
    @ rm -rf dist

.PHONY: all clean
```

By prepending an `@` sigil to a command line in our Makefile, we're silencing the output. Also - our clean target will delete the entire build directory. We can test that by running **make clean**.

Try it out!

Orchestrating The Build

Our code files are a bit messy - just stored right in the project root. So, let's clean things up with some organization:

```
mkdir src mv app.js src
```

Great. Let's assume (for now) that all our project code goes into the `/src` directory.

Our current Makefile is not really doing any build orchestrations of any kind – it's just copying a single JavaScript file to the `/dist` directory. Let's change that by adding a build timestamp to the output, so we know when the last build was run.

Let's clean things up a bit so we have a build target that produces a file and another that produces the destination:

```
all: dist app.js

dist:
    @ mkdir -p dist

app.js:
    @ cp src/app.js dist/app.js

clean:
    @ rm -rf dist

.PHONY: all clean
```

A lot just went on there – and with it we get to learn some more jargon.

First, I created two new targets called `app.js` and `dist`. Targets in a Makefile are just the labels; what comes after the target is the recipe. So, for the `dist` target, `mkdir -p dist` is the recipe.

The `all` target has prerequisites that appear on the first line, but it has no recipe. Prerequisites are targets that must

be built before creating the current target. So, for **all** to work, **dist** and **app.js** need to have run first.

If you put a target, recipe and prerequisites together, you have a *rule*. Which is precisely what a Makefile is: *a set of rules for building your software*.

This is the power of Make. Orchestrating what happens when, and in what order. You could say that this is all that Make does, which isn't surprising if you understand the Linux philosophy of "do one thing well".

Using Variables

We're dealing with shell scripts here, and just like any programming effort, repeating ourselves is typically frowned upon. Let's take the hard-coded stuff out first so we can change as we need to.

By convention, variables should be declared at the top of your Makefile:

```
JS_FILES=src/app.js
DIST=dist

all: dist app.js

dist:
    @ mkdir -p $(DIST)

app.js:
    @ cp $(JS_FILES) $(DIST)/app.js

clean:
    @ rm -rf $(DIST)

.PHONY: all clean
```

Now that's starting to look like a proper Makefile! Variables in any shell script should typically be upper-cased.

Notice how I must use **\$(DIST)** to reference the variable? Do you remember what that is? It's a *command-replacement subshell*. Make runs all the commands in a subshell, outside its process.

At first glance it might not look like we've done much here – but if we change our source files, which we will, it's a simple change to the **JS_FILES** variable.

OK, one last thing. We're still repeating ourselves in a couple of places – specifically with the target and the destination file names of **app.js**. If we're leaning on convention, we shouldn't have to specify things twice.

To get around this, we can use the **\$@** shorthand – which means “this target name”:

```
JS_FILES=src/app.js
DIST=dist

all: dist app.js

dist:
    @ mkdir -p $@

app.js:
    @ cp $(JS_FILES) $(DIST)/app.js

clean:
    @ rm -rf $(DIST)

.PHONY: all clean
```

We're getting there. Now, let's create our timestamp. For this I'll use a variable straight away, and put it right at the top:

```
JS_FILES=src/app.js
DIST=dist
TODAY=$(shell date +%Y-%B-%d)
TIMESTAMP="//Created at $(TODAY) \n\n"
#...
```

Here I'm using the shell command, which, if you recall from previous sections, executes shell commands for you. We need to get the literal value of the date and the date's formatting instruction, and store it in the **TODAY** variable.

We then create our timestamp. Using variables is a great way to keep your code clean and understandable, and that's critical in Makefiles as the syntax can quickly become overwhelming.

The next task is to read the source file and timestamp it:

```
#...
app.js:
    @ echo $(TIMESTAMP) > $(DIST)/$@
    @ cat $(JS_FILES) >> $(DIST)/$@
#...
```

This is a bit roundabout, but it works. As I keep mentioning, there's almost always a better way when it comes to shell scripting – and we should most definitely extend that idea to Makefiles. This is nice and clear to me, however, and hopefully it makes sense to you as well.

I'm redirecting the output of **echo** (which is the **TIMESTAMP** variable) into **dist/app.js**, which is our output file. On the next line I'm appending **STDOUT** using **>>** to the same file – this time with the contents of the JavaScript files in our **src** directory.

Here's another go at this:

```

JS_FILES=src/app.js
DIST=dist
TODAY=$(shell date +%Y-%B-%d)
TIMESTAMP="//Created at $(TODAY) \n\n"

all: dist app.js

dist:
    @ mkdir -p $@

app.js:
    @ echo $(TIMESTAMP) > $(DIST)/$@
    @ cat $(JS_FILES) >> $(DIST)/$@

clean:
    @ rm -rf $(DIST)

.PHONY: all clean

```

This is a solid Makefile, but it's not quite there yet. Notice that I have a **dist** target and a **DIST** variable? This is redundant! You can use variables as target names, so let's do that and arrive at our final Makefile:


```

JS_FILES=src/app.js
DIST=dist
TODAY=$(shell date +%Y-%B-%d)
TIMESTAMP="//Created at ${TODAY} \n\n"

all: dist app.js

dist:
    @ mkdir -p $@

app.js:
    @ echo ${TIMESTAMP} > ${DIST}/${@}
    @ cat ${JS_FILES} >> ${DIST}/${@}

clean:
    @ rm -rf ${DIST}

.PHONY: all clean

```

Let's give it a run using `make clean && make`. You should see a `dist/app.js` file with this in it:

```

//Created at 2016-October-19

//some code

```

Nice work! Once again, this is a bit of goofy demo, but your brain should be racing a little now... thinking about all the ways you might put Make to use with your project.

In the next section we'll do even more work with JavaScript files, kicking Grunt right out of our project.

Using Make to Build Your Web App

Every language/platform seems to have its own build and automation toolset. Microsoft has MSBuild, Java has Ant, Maven, Gradle, and perhaps a few others. JavaScript has Grunt, Gulp, Jake and a few more. Ruby has Rake. Why is this?

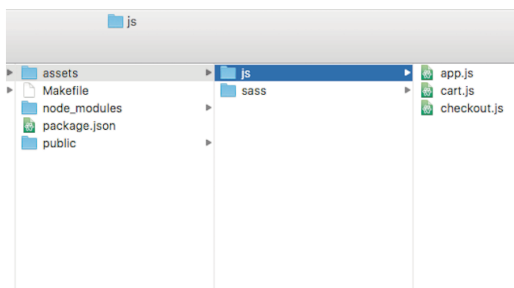
The short answer is that Make can be a little opaque and, predictably, language users like to build things using their own language and believe it will impart some type of benefit over the decades-old Makefile.

This subject, like others in this book, is a tad volatile and I want to recognize that right up front. An easy way to start an argument between two developers is to bring up the subject of build automation! A good friend of mine, Rob Ashton, thought it would be fun to submit a pull request to the MSBuild GitHub Repo, suggesting that it be replaced with Make. If you've ever had to wrestle with MSBuild before (which I have, numerous times), then this might make you laugh. I thought it was funny, but there are many who did not.

No matter which build tool you use and love, knowing Make will give you *a perspective that you need*. It's built right into Unix-based systems and you can use it to automate just about anything. Whenever you crack open that Gruntfile, Gulpfile, or Rakefile, ask yourself whether the layers of complication you're adding to your project by using these tools are really warranted. The answer, typically, is no.

The problem, typically, is that most people don't know how Make works. Let's change that now, using a fairly typical use case: compiling assets for a web project.

I have a typical web application that uses SASS and a bit of JavaScript:



As you can see, it's a simple app. I have a set of JavaScript files and a single SASS file that I need to build and then output to a directory somewhere.

This is my goal for this task: *concatenate, compress and build my SASS files and then concatenate and uglify my JavaScript files.*

Working With SASS

Some people dig SASS, others LESS. Other people (like myself) tend to knuckle under and just write CSS directly. Hopefully what you're about to do will translate to how you develop things.

Also: when working with SASS and LESS it's common to import all of your files into a main.scss file, which you then build with the interpreter. That's what I'll be doing today.

I need to install a few Node modules to help me out:

- **node-sass:** the module I need to compile and build my SASS file
- **uglifyjs:** removes whitespace, concatenates and then minifies my JavaScript files

That's it. To install these we use NPM:

```
npm install uglifyjs node-sass --save-dev
```

The next step is to create the Makefile in the project root and then give it a default structure:

```
touch Makefile
```

Open the Makefile and then add the default structure – the targets we know we’re going to be working with:

```
all:

clean:

.PHONY: all
```

Hey we’re getting good at this!

We don’t want any hardcoded values in our rules, so let’s define as much as we can at the top:

```
SASS=node_modules/.bin/node-sass
SASS_FILES=assets/sass/main.scss
JS_FILES=assets/js/*.js
UGLIFY=node_modules/.bin/uglifyjs
DIST_CSS=public/css
DIST_JS=public/js

all:

clean:

.PHONY: all
```

I’m defining the binaries, the source files, and the destinations.

Binaries

I've installed the **node-sass** and **uglifyjs** binaries locally as development dependencies. This will increase the time it takes to run `npm install`, but it also guarantees that the binaries will be present. It's really annoying to have to install global dependencies (my opinion).

In case this is all new to you or if you've never used Node: you can install packages from NPM and have them run locally or globally. You've probably seen a `node_modules` directory at some point in your career – this is the local package installation location. If I was to install `node-sass` locally I could run it by using `node ./node_modules/.bin/node-sass`. Or I could make life easy on myself and install it globally in the global NPM cache, which would allow me to run the binary anywhere on my machine.

Given that I want to smash all the JavaScript files into a single `app.js` file, I can create that target. Same with `app.css`. The destination for these files will be the `public` directory, which should exist already in my project. However we can't guarantee that so let's make sure that we have a **dist** target as well:

```

SASS=node_modules/.bin/node-sass
SASS_FILES=assets/sass/main.scss
JS_FILES=assets/js/*.js assets/js/**/*.js
UGLIFY=node_modules/.bin/uglifyjs
DIST_CSS=public/css
DIST_JS=public/js

all: dist app.css app.js

dist:
    mkdir $(DIST_CSS) $(DIST_JS)

app.css:

app.js:

clean:
    @rm -rf $(DIST_CSS) $(DIST_JS)

.PHONY: all

```

Simple enough. Creating the directories, we need with the **dist** rule, removing them with **clean**. Now all we need to do is to fill in the commands to create the files¹.

The first step is simple: we only need to compile the `main.scss` file using `node-sass`, which comes with a binary in the `node_modules/.bin` directory.

This is what the command would look like normally:

```
node_modules/.bin/node-sass --output-style compressed assets/sass/main.scss > public/css/
```

We just need to replace this command with our variables and we're good to go:

```
app.css:
@ $(SASS) --output-style compressed $(SASS_FILES) > $(DIST_CSS)/$@
```

Bam. That couldn't be easier!

OK, I lied: *it could be easier*. You'll find that most of your time is spent wrangling the commands together – reading the docs, trying to understand how to use the binaries properly.

When I first put this file together for a project I was working on, I had to read through the source code of some binary files to understand all the options and how they worked. I then ran those commands from the command line to make sure things happened the way I wanted.

The JavaScript files are a bit of a different story. I need to concatenate them together, and then uglify/compress them. Concatenating is simple – we can use `cat` directly to pull the code from each file – which we can then pipe directly to `uglifyjs`:

```
app.js:
@ cat $(JS_FILES) | $(UGLIFY) > $(DIST_JS)/$@
```

We haven't discussed the `|` notation you see here. That's called the pipe operator and is a Unix operator that redirects the output of one command to the input of the following command. In the same way I redirected **STDOUT** to a text file using `>` in a previous section, the `|` operator redirects **STDOUT** and **STDIN** for two given functions.

In this example, I'm using `cat` to concatenate all the JavaScript files specified by my `JS_FILES` variable. I'm then piping that text into the `uglifyjs` binary, and then redirecting the output of the `uglifyjs` call to my destination.

I wish I could tell you it was harder than this ... but it's

not. This is the power of Make. Run `make` and look at the built output in the public directories. If you've already run it, be sure to run `make clean && make` before you run `make` again to avoid errors.

If this seems a bit cryptic or if your reaction to this code is something like “yeah great for you – you know shell scripts!” you might be interested to know that I'm a complete n00b to this stuff. I learned it better over the last year but creating Makefiles like this was absolutely something that was beyond me just 8 months ago.

It just takes a little time, some Googling, and some practice and you will get it too.

Ideally your JavaScript files are not dependent on load order, but in the Real World this is typically not the case. There are several ways to get around the problem and they involve knowledge of some basic commands.

Personally, if I need files loaded in a particular order (as with my SQL files – tables need to be built before views and so on) – I just name them in a particular way:

- 01-init.sql
- 02-tables.sql
- 03-views.sql
- 04-funtions.sql

This works fine for SQL files, but it might not work for your JavaScript build. If you need files loaded in a particular order you can pipe the result to `sed`, as we did with the Jekyll task, and then transform it from there.

You can also separate the build steps, so you build a directory first, then another, outputting interim files to `assets/tmp` directory, which you then concatenate later.

In this section, we've touched on some very fundamental ways we can use Make to automate and simplify our lives.

But we have only scratched the surface. Make has a long, long legacy, and is the spiritual great-granddaddy of newer tools like Grunt, Gulp, and other automation tools we take for granted. I can't overstate the value that even a basic fluency with this tool will bring to your understanding of how source files are assembled into working software.

FINAL THOUGHTS

IN WHICH WE SAY GOODBYE... FOR NOW...

This book began as a series of tasks in Wunderlist: *things I need to know someday*. I didn't intend to write a book. I thought maybe I would write some blog posts, perhaps do a video or two on a few of the topics I found.

The problem I had then (and still have now) is this: **I didn't know what I didn't know**. It's getting worse, too. The more I learn – the more I feel like an imposter.

There is comfort in not-knowing something. More than that: there is magic. Even more than that I think there is also a **bit of dumb courage**. You just don't know if it will hurt – so you try it out.

I'm reminded of being 15 and jumping off the roof of my house with an umbrella. Along with wondering if the umbrella would slow me down, *I also wondered how badly it would hurt*. Turns out the umbrella did nothing and jumping off the roof of my mom's house did, in fact, hurt. I didn't break anything or seriously injure myself, which is a good thing ... but I only tried the umbrella jump once. The next time I did it without the umbrella ...

I picked the cover of this book because that's what it felt like when I learned to program on my own: *jumping off something rather high, hoping the landing wouldn't break me somehow*. I had a job and a budding career as a geologist – but it was boring. A good friend of mine was a trainer – the very first one certified by Microsoft to teach Active Server Pages. So he taught me (thanks Dave!) and I **jumped**.

I've hit the ground hard a few times, but the joy of jumping into new things is my drug. Which is kind of a problem because new things don't stay new for very long, and soon you start to feel the magic of it all start to fade.

It's the curse of learning: remove the mystery, remove the magic. Move on to the next mystery. Pretty soon you wonder what the point is.

However.

The process of writing this book has been *transformative*. I'm going to push the whole jumping metaphor a little more – and I'm also going to reflect on the cover image a bit more too. You may not have noticed, but on the back of the “jumper” on the cover where a parachute should be is ... this book. If you look closer, you can see the same cover. I felt it appropriate as recursion seems to ... recur throughout this book.



More than that, though, is the idea of a parachute. This

book has become my parachute, in a sense – the things I’ve learned over the last year are allowing me to jump off higher rooftops – but this time I have a bit of knowledge to soften the fall.

Climb higher, jump farther. The mystery/magic/dumb courage seems to be increasing as I learn these things.

One Final Thought

I’ve gone deeper into these topics than I ever thought I would. I feel like I have a solid grasp on complexity theory, Big-O, graphing algorithms and Bernoulli’s Weak Law of Large Numbers and ... just writing this sentence is giving me a rush! This shit is BREATHTAKING!



Sorry. I promised myself I wouldn’t swear in this book – but if any of the paragraphs written deserve some caps and a four-letter word ... well it’s that one.

There. Is. So. Much. Magic.

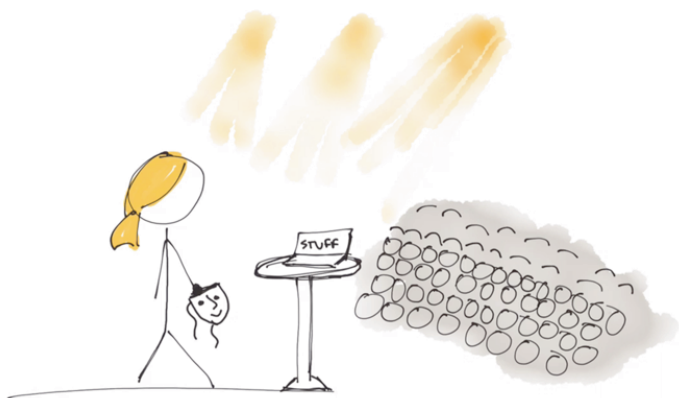
Go find it. Don’t stop here. It’s so easy to fall into snark and uncertainty; to feel like *you just don’t know what other people*

are talking about. It can really be isolating and lead one to look upon others with scorn for trying new things. As I write this sentence, Facebook just pushed a new package manager for JavaScript. As expected, quite a few people ridiculed it and laughed without even trying it!

Let's not be those people. We're the explorers who keep trying (and hopefully failing). We push the edges to find out what's possible. Enjoy this!

I urge you to explore and dig deeper, ask more questions, take a friend out and bore them to tears with all you know! Draw a picture of Traveling Salesman for your kids and ask how they would solve it. Make a Markov Chain out of popcorn for the holidays!

With that: *I leave you.* Thank you for going on this journey with me. Now let's go make a difference.



NOTES

Preface

1. I don't know of anyone who has ever laid down a solidly factual reason for tabs vs. spaces. This is fine, it's a preference almost every time. Understanding this is key, especially if your boss is the one with the opinion. Their code, their style, end of story.

Introduction

1. Von Neumann's Self-Reproducing Automata, Arthur W. Brooks. June 1969.
2. At the risk of sending you down a rabbit hole right off the bat, here's some light reading on this wonderful mathematical paradox <https://math.stackexchange.com/questions/162/why-is-the-set-of-all-sets-a-paradox-in-laymans-terms>
3. We'll be playing around with this algorithm later in the book.
4. If you've been through a whiteboard interview, you know this feeling all too well. This is, in fact, what the interviewer wants to know! What, exactly, are you capable of computing!
5. <https://www.mathsisfun.com/definitions/natural-number.html>
6. Given recent events at the onset of the pandemic, I'm sure you're well aware of the limitations of toilet paper supply.
7. You can watch a proof of this from an MIT lecture by Erik Demaine. It's a fascinating watch! https://youtu.be/moPtWq_cVH8?t=708
8. You've likely heard about $P=NP$? If not, you have some fun coming your way in the chapter on Complexity Theory. Skynet, by the way, is the sentient computer system from The Terminator movies. Wonderful story about machines that gain self-awareness. Which is exactly where we're headed if we're to solve anything truly worth solving.

3. Computing Machines

1. We'll read more about Turing throughout this book.
2. The term *infinite* is used a lot when discussing Turing Machines – don't get hung up on it. If you need to, translate it as “just enough”. In other words, your computer can store massive amounts of information – but

someday it will fill up. At that point you can go get another hard drive – and someday you’ll fill that up. But then you’ll get another drive ... the space your machine has on disk has nothing to do with its computing power – same with the size of RAM. These things do have a lot to do with how long you’ll have to wait, however.

10. Object-Oriented Design Principles

1. I know the word “global” is a bad one when it comes to programming and variables. Don’t get stuck on this! We simply need to ensure that we can access the container when we need it, and often tools such as Ninject have ways of ensuring that you can access the container without using global variables.
2. Sort of. I’ve used Rails and Django a few times, which have their own ways of doing things. I’ve also used ORMs for smaller projects which also have their own opinions. When I’m free to choose what I want, this is what I do.

11. Test Driven Design

1. YAGNI is covered in the Software Design Principles chapter
2. DRY is covered in the Software Design Principles chapter
3. A “mock” is typically an object, or a “what” that we set explicit expectations on. Like when we mocked the subscription repository above to return two subscriptions. Stubs are dummy objects, typically, that contain the answers we’re looking for. For instance, you might stub a Customer to see if the status changed during a test. These terms are often confused.

12. Behavior Driven Design

1. The XUnit way of doing BDD is a little verbose and there are other ways. I typically don’t like a ton of testing dependencies so I’ll stick with the most basic tooling I can. In Node, for instance, you can write “specs” easily by telling Mocha (a Node testing framework) what style you’ll be doing. This amounts to syntactic sugar, but it is very helpful when reading results.

15. Shell Scripting

1. The more you work with command line tools, the more you'll come across "rc files". You don't have to name startup scripts with an "rc" ending, but if you do, people will know what it's supposed to do by convention.

Like so many things in Unix land, the origin of the term "rc" is a bit cloudy. Google it if you like, but the actual meaning of it doesn't matter. Just know that `.thingrc` will be the startup script for the thingcommand/binary/whatever. Some people like to think it means "runtime configuration" – that sounds like a good explanation to me.

16. Make

1. Loading up your source files like this is simple when using a glob pattern, but you might have different needs. For instance: you might have an `app.js` file in `assets/js` that you want loaded first, and then all the other files loaded after that. You can do this easily by specifying `assets/app.js` as the first file in the list, and then the glob `assets/js/**/*.js` after that.

