ULTIMATE GO NOTEBOOK

WRITTEN BY WILLIAM KENNEDY
WITH HOANH AN

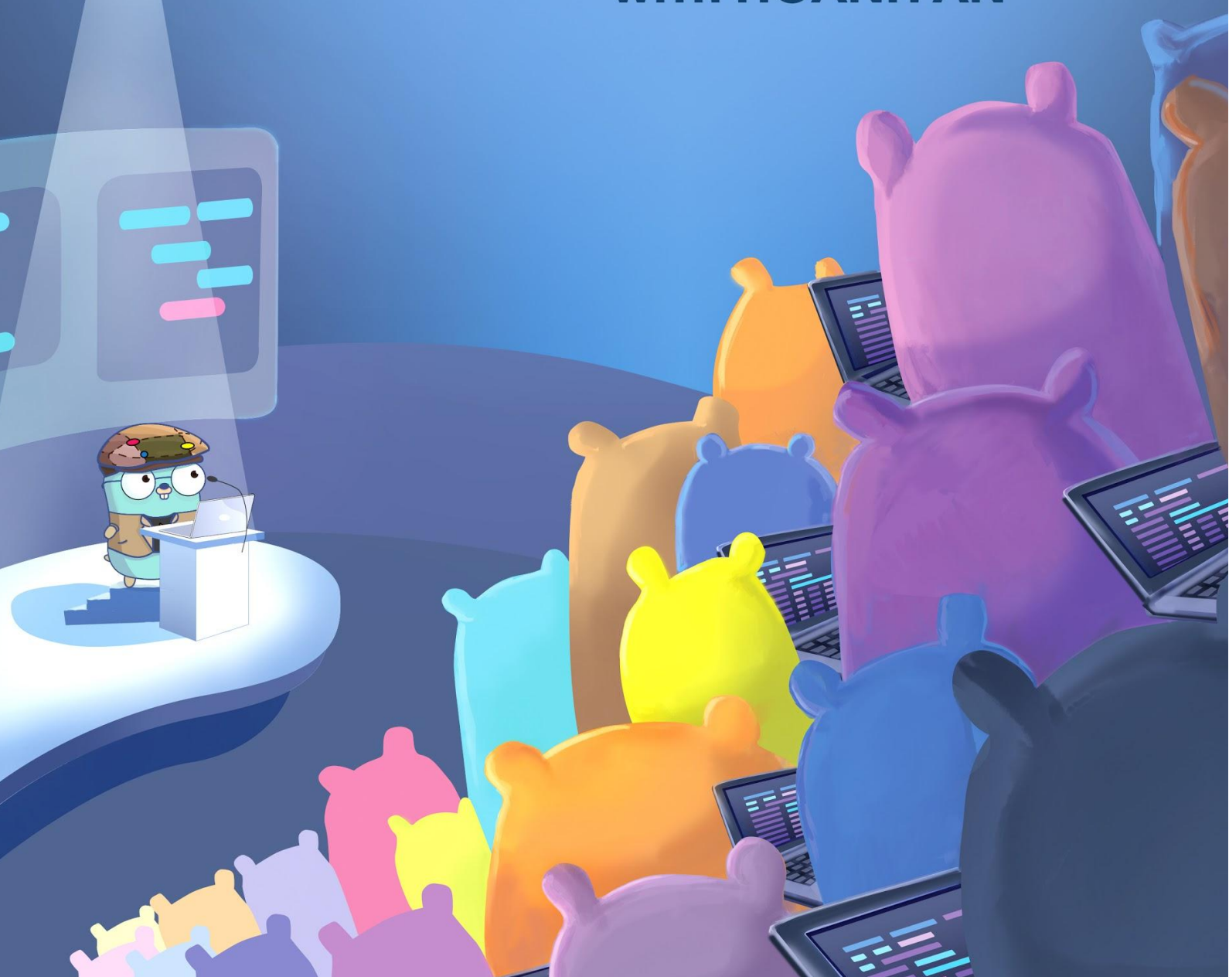


Table of Contents - Version 0.3

Versioning

March 2021 - Version 0.3 : Verdana Font, 1.25 line spacing, chapter numbers
March 2021 - Version 0.2 : Initial generics, more channel patterns, design philosophy
March 2021 - Version 0.1 : Initial draft release

Todo

- Illustrations
- More Generic Code
- Consider removing blog links for content
- Add chapter intro content

Welcome

Back in August 2019, Hoanh An started a project in Github called the Ultimate Go Study Guide. It was a collection of notes he took after taking the Ultimate Go class. Surprisingly, it got a lot of attention from the community and eventually had more stars and activity than the actual repo for the class. This shows the power of open sourcing material.

Then Hoanh decided to publish a book from his notes and repo. When I saw what Hoanh had written and the excitement his followers had, I reached out to him. We decided I would review and refactor his original work and we would publish a book together. This is that book and it represents the notes I would like any student to make while taking the class.

If you have taken the class before, these notes will sound and feel familiar. I think it's a great reference to all the things I say in class. If you've never taken the class before, I think the notes are still a good source of learning Go. Maybe it will give you the push you need to take the class.

Now in April 2021, it's taken us six months to get the initial release of the book in your hands. The plan is to continue to add more and more content to the book over time. Things like modules, generics, and packages from the standard library are slated for future releases.

I want to thank everyone in the Go community for their support and help over the years in creating this material. When I started in March 2013, I didn't know anything about Go and had no idea where Go would take me. Learning is a journey that takes time and effort. If this material can help jump start your learning about Go, then the time and effort was worth every minute.

Thanks,
-- Bill Kennedy

Chapter 1 - Introduction

It’s important that you prepare your mind for the material you are about to read. The introduction will provide some thoughts and ideas, using quotes to stimulate your initial understanding of the language.

Somewhere Along The Line

- We became impressed with programs that contain large amounts of code.
- We strived to create large abstractions in our code base.
- We forgot that the hardware is the platform.
- We lost the understanding that every decision comes with a cost.

These Days Are Gone

- We can throw more hardware at the problem.
- We can throw more developers at the problem.

Open Your Mind

- Technology changes quickly but people's minds change slowly.
- Easy to adopt new technology but hard to adopt new ways of thinking.

Interesting Questions - What do they mean to you?

- Is it a good program?
- Is it an efficient program?
- Is it correct?
- Was it done on time?
- What did it cost?

Aspire To

- Be a champion for quality, efficiency and simplicity.
- Have a point of view.
- Value introspection and self-review.

1.1 Reading Code

Go is about being a language that focuses on code being readable.

Quotes

“If most computer people lack understanding and knowledge, then what they will select will also be lacking.” - Alan Kay

"The software business is one of the few places we teach people to write before we teach them to read." - Tom Love (inventor of Objective C)

"Code is read many more times than it is written." - Dave Cheney

"Programming is, among other things, a kind of writing. One way to learn writing is to write, but in all other forms of writing, one also reads. We read examples both good

and bad to facilitate learning. But how many programmers learn to write programs by reading programs?" - Gerald M. Weinberg

"Skill develops when we produce, not consume." - Katrina Owen

1.2 Legacy Software

Do you care about the legacy you are leaving behind?

Quotes
"There are two kinds of software projects: those that fail, and those that turn into legacy horrors." - Peter Weinberger (inventor of AWK)

"Legacy software is an unappreciated but serious problem. Legacy code may be the downfall of our civilization." - Chuck Moore (inventor of Forth)

"Few programmers of any experience would contradict the assertion that most programs are modified in their lifetime. Why then do we rarely find a program that contains any evidence of having been written with an eye to subsequent modification." - Gerald M. Weinberg

"We think awful code is written by awful devs. But in reality, it's written by reasonable devs in awful circumstances." - Sarah Mei

"There are many reasons why programs are built the way they are, although we may fail to recognize the multiplicity of reasons because we usually look at code from the outside rather than by reading it. When we do read code, we find that some of it gets written because of machine limitations, some because of language limitations, some because of programmer limitations, some because of historical accidents, and some because of specifications—both essential and inessential." - Gerald M. Weinberg

1.3 Mental Models

You must constantly make sure your mental model of your projects are clear. When you can't remember where a piece of logic is or you can't remember how something works, you are losing your mental model of the code. This is a clear indication that refactoring is a must. Focus time on structuring code that provides the best mental model possible and code review for this as well.

How much code in that box do you think you can maintain a mental model of in your head? I believe asking a single developer to maintain a mental model of more than one ream of paper in that box (~10k lines of code) is asking a lot. If you do the math, then it takes a team of 100 people to work on a code base that hits a million lines of code. That is 100 people that need to be coordinated, grouped, tracked and in a constant feedback loop of communication.

Quotes
"Let's imagine a project that's going to end up with a million lines of code or more. The probability of those projects being successful in the United States these days is very low - well under 50%. That's debatable." - Tom Love (inventor of Objective C)

"100k lines of code fit inside a box of paper." - Tom Love (inventor of Objective C)

"One of our many problems with thinking is "cognitive load": the number of things we can pay attention to at once. The cliche is 7±2, but for many things it is even less. We make progress by making those few things be more powerful." - Alan Kay

"The hardest bugs are those where your mental model of the situation is just wrong, so you can't see the problem at all." - Brian Kernighan

"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?" - Brian Kernighan

"Debuggers don't remove bugs. They only show them in slow motion." - Unknown

"Fixing bugs is just a side effect. Debuggers are for exploration." - @Deech (Twitter)

Reading
The Magical Number Seven, Plus or Minus Two - Wikipedia
Psychology of Code Readability - Egon Elbre

1.4 Productivity vs Performance

Productivity and performance both matter, but in the past you couldn't have both. You needed to choose one over the other. We naturally gravitated to productivity, with the idea or hope that the hardware would resolve our performance problems for free. This movement towards productivity has resulted in the design of programming languages that produce sluggish software that is outpacing the hardware's ability to make them faster.

By following Go's idioms and a few guidelines, we can write code that can be reasoned about by anyone who looks at it. We can write software that simplifies, minimizes and reduces the amount of code we need to solve the problems we are working on. We don't have to choose productivity over performance or performance over productivity anymore. We can have both.

Quotes
"The hope is that the progress in hardware will cure all software ills. However, a critical observer may observe that software manages to outgrow hardware in size and

sluggishness. Other observers had noted this for some time before, indeed the trend was becoming obvious as early as 1987." - Niklaus Wirth

"The most amazing achievement of the computer software industry is its continuing cancellation of the steady and staggering gains made by the computer hardware industry." - Henry Petroski (2015)

"The hardware folks will not put more cores into their hardware if the software isn't going to use them, so, it is this balancing act of each other staring at each other, and we are hoping that Go is going to break through on the software side." - Rick Hudson (2015)

"C is the best balance I've ever seen between power and expressiveness. You can do almost anything you want to do by programming fairly straightforwardly and you will have a very good mental model of what's going to happen on the machine; you can predict reasonably well how quickly it's going to run, you understand what's going on." - Brian Kernighan (2000)

"The trend in programming language design has been to create languages that enhance software reliability and programmer productivity. What we should do is develop languages alongside sound software engineering practices so the task of developing reliable programs is distributed throughout the software lifecycle, especially into the early phases of system design." - Al Aho (2009)

1.5 Correctness vs Performance

You want to write code that is optimized for correctness. Don't make coding decisions based on what you think might perform better. You must benchmark or profile to know if code is not fast enough. Then and only then should you optimize for performance. This can't be done until you have something working.

Improvement comes from writing code and thinking about the code you write. Then refactoring the code to make it better. This requires the help of other people to also read the code you are writing. Prototype ideas first to validate them. Try different approaches or ask others to attempt a solution. Then compare what you have learned.

Too many developers are not prototyping their ideas first before writing production code. It is through prototyping that you can validate your thoughts, ideas and designs. This is the time when you can break down walls and figure out how things work. Prototype in the concrete and consider contracts after you have a working prototype.

Refactoring must become part of the development cycle. Refactoring is the process of improving the code from the things that you learn on a daily basis. Without time to refactor, code will become impossible to manage and maintain over time. This creates the legacy issues we are seeing today.

Quotes

"Make it correct, make it clear, make it concise, make it fast. In that order." - Wes Dyer

"Good engineering is less about finding the "perfect" solution and more about understanding the tradeoffs and being able to explain them." - JBD

"Choosing the right limitations for a certain problem domain is often much more powerful than allowing anything." - Jason Moiron

"The correctness of the implementation is the most important concern, but there is no royal road to correctness. It involves diverse tasks such as thinking of invariants, testing and code reviews. Optimization should be done, but not prematurely." - Al Aho (inventor of AWK)

"The basic ideas of good style, which are fundamental to write clearly and simply, are just as important now as they were 35 years ago. Simple, straightforward code is just plain easier to work with and less likely to have problems. As programs get bigger and more complicated, it's even more important to have clean, simple code." - Brian Kernighan

"Problems can usually be solved with simple, mundane solutions. That means there's no glamorous work. You don't get to show off your amazing skills. You just build something that gets the job done and then move on. This approach may not earn you oohs and aahs, but it lets you get on with it." - Jason Fried

Reading

Prototype your design! - Robert Griesemer

1.6 Rules

What should we understand about rules?

- Rules have costs.
- Rules must pull their weight - Don’t be clever (high level).
- Value the standard, don’t idolize it.
- Be consistent!
- Semantics convey ownership.

Quotes

"An architecture isn't a set of pieces, it's a set of rules about what you can expect of them." - Michael Feathers

Reading

The Philosophy of Google's C++ Code - Titus Winters

1.7 Senior vs Junior Developers

What is the difference between a Senior and Junior developer?

Quotes
"You are personally responsible for the software you write." - Stephen Bourne (Bourne shell)

"And the difference between juniors+seniors to those who are in-between, is the confidence to ask "dumb" questions." - Natalie Pistunovich

"Mistakes are an inevitable consequence of doing something new and, as such, should be seen as valuable; without them, we'd have no originality." - Ed Catmull (President of Pixar)

"It takes considerable knowledge just to realize the extent of your own ignorance." - Thomas Sowell

"If you don't make mistakes, you're not working on hard enough problems." - Frank Wilczek

"Don't cling to a mistake because you spent so much time making it." - Aubrey de Grey

1.8 Code Reviews

You can't look at a piece of code, function, or algorithm and determine if it smells good or bad without a design philosophy. These four major categories are the basis for code reviews and should be prioritized in this order: Integrity, Readability, Simplicity and then Performance. You must consciously and with great reason be able to explain the category you are choosing.

1.9 Integrity

We need to become very serious about reliability.

There are two driving forces behind integrity:

- Integrity is about every allocation, read and write of memory being accurate, consistent and efficient. The type system is critical to making sure we have this micro level of integrity.
- Integrity is about every data transformation being accurate, consistent and efficient. Writing less code and error handling is critical to making sure we have this macro level of integrity.

Write Less Code

There have been studies that have researched the number of bugs you can expect to have in your software. The industry average is around 15 to 50 bugs per 1000 lines of code. One simple way to reduce the number of bugs, and increase the integrity of your software, is to write less code.

Bjarne Stroustrup stated that writing more code than you need results in Ugly, Large and Slow code:

- Ugly: Leaves places for bugs to hide.
- Large: Ensures incomplete tests.
- Slow: Encourages the use of shortcuts and dirty tricks.

Error Handling
When error handling is treated as an exception and not part of the main code, you can expect the majority of your critical failures to be due to error handling.
48 critical failures were found in a study looking at a couple hundred bugs in Cassandra, HBase, HDFS, MapReduce, and Redis.

- 92% : Failures from bad error handling
    - 35% : Incorrect handling
        - 25% : Simply ignoring an error
        - 8% : Catching the wrong exception
        - 2% : Incomplete TODOs
    - 57% System specific
        - 23% : Easily detectable
        - 34% : Complex bugs
- 8% : Failures from latent human errors

Quotes
"Failure is expected, failure is not an odd case. Design systems that help you identify failure. Design systems that can recover from failure." - JBD

"Product excellence is the difference between something that only works under certain conditions, and something that only breaks under certain conditions". - Kelsey Hightower

"Instability is a drag on innovation." - Yehudah Katz

Reading
Software Development for Infrastructure - Bjarne Stroustrup
Normalization of Deviance in Software - danluu.com
Lessons learned from reading postmortems - danluu.com
Technical Debt Quadrant - Martin Fowler
Design Philosophy On Integrity - William Kennedy
Ratio of bugs per line of code - Dan Mayer

Masterminds of Programming - Federico Biancuzzi and Shane Warden
Developing Software The Right Way, with Intent and Carefulness - David Gee
What bugs live in the Cloud - usenix.org

1.10 Readability

We must structure our systems to be more comprehensible.

This is about writing simple code that is easy to read and understand without the need of mental exhaustion. Just as important, it's about not hiding the cost/impact of the code per line, function, package and the overall ecosystem it runs in.

Code Must Never Lie
We have all been here if you have been programming long enough. At this point it doesn't matter how fast the code might be if no one can understand or maintain it moving forward.

Average Developer
You must be aware of who you are on your team. When hiring new people, you must be aware of where they fall. The code must be written for the average developer to comprehend. If you are below average, you have the responsibility to come up to speed. If you are the expert, you have the responsibility to reduce being clever.

Real Machine
In Go, the underlying machine is the real machine rather than a single abstract machine. The model of computation is that of the computer. Here is the key, Go gives you direct access to the machine while still providing abstraction mechanisms to allow higher-level ideas to be expressed.

Quotes
"This is a cardinal sin amongst programmers. If code looks like it's doing one thing when it's actually doing something else, someone down the road will read that code and misunderstand it, and use it or alter it in a way that causes bugs. That someone might be you, even if it was your code in the first place." - Nate Finch

"Can you explain it to the median user (developer)? as opposed to will the smartest user (developer) figure it out?" - Peter Weinberger (inventor of AWK)

"Making things easy to do is a false economy. Focus on making things easy to understand and the rest will follow." - Peter Bourgon

Reading
Code Must Never Lie

1.11 Simplicity

We must understand that simplicity is hard to design and complicated to build.

This is about hiding complexity. A lot of care and design must go into simplicity because it can cause more problems than it solves. It can create issues with readability and it can cause issues with performance.

Complexity Sells Better
Focus on encapsulation and validate that you're not generalizing or even being too concise. You might think you are helping the programmer or the code but simple, valid things are still easy to use, understand, debug and maintain.

Encapsulation
Encapsulation is what we have been trying to figure out as an industry for 40 years. Go is taking a slightly new approach with the package. Bringing encapsulation up a level and providing richer support at the language level.

Quotes
"Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better." - Edsger W. Dijkstra

"Everything should be made as simple as possible, but not simpler." - Albert Einstein

"You wake up and say, I will be productive, not simple, today." - Dave Cheney

Paraphrasing: "Encapsulation and the separation of concerns are drivers for designing software. This is largely based on how other industries handle complexity. There seems to be a human pattern of using encapsulation to wrestle complexity to the ground." - Brad Cox (inventor of Objective C)

"The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise." - Edsger W. Dijkstra

"A good API is not just easy to use but also hard to misuse." - JBD

"Computing is all about abstractions. Those below yours are just details. Those above yours are limiting complicated crazy town." - Joe Beda

Reading
Simplicity is Complicated - Rob Pike
What did Alan Kay mean by, "Lisp is the greatest single programming language ever designed"? - Alan Kay

1.12 Performance

We must compute less to get the results we need.

This is about not wasting effort and achieving execution efficiency. Writing code that is mechanically sympathetic with the runtime, operating system and hardware. Achieving performance by writing less and more efficient code but staying within the idioms and framework of the language.

Rules of Performance

- Never guess about performance.
- Measurements must be relevant.
- Profile before you decide something is performance critical.
- Test to know you are correct.

Broad Engineering

Performance is important but it can't be your priority unless the code is not running fast enough. You only know this once you have a working program and you have validated it. We place those who we think know how to write performant code on a pedestal. We need to put those who write code that is optimized for correctness and performs fast enough on those pedestals.

Quotes

"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%." — Donald E. Knuth

"I don't trust anything until it runs... In fact, I don't trust anything until it runs twice." - Andrew Gelman (one of the greatest living statisticians at Columbia University).

"When we're computer programmers we're concentrating on the intricate little fascinating details of programming and we don't take a broad engineering point of view about trying to optimize the total system. You try to optimize the bits and bytes." - Tom Kurtz (inventor of BASIC)

1.13 Micro-Optimizations

Micro-Optimizations are about squeezing every ounce of performance as possible. When code is written with this as the priority, it is very difficult to write code that is readable, simple or idiomatic. You are writing clever code that may require the unsafe package or you may need to drop into assembly.

1.14 Data-Oriented Design

"Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming." - Rob Pike

Design Philosophy

- If you don't understand the data, you don't understand the problem.
- All problems are unique and specific to the data you are working with.
- Data transformations are at the heart of solving problems. Each function, method and work-flow must focus on implementing the specific data transformations required to solve the problems.
- If your data is changing, your problems are changing. When your problems are changing, the data transformations need to change with it.
- Uncertainty about the data is not a license to guess but a directive to STOP and learn more.
- Solving problems you don't have, creates more problems you now do.
- If performance matters, you must have mechanical sympathy for how the hardware and operating system work.
- Minimize, simplify and REDUCE the amount of code required to solve each problem. Do less work by not wasting effort.
- Code that can be reasoned about and does not hide execution costs can be better understood, debugged and performance tuned.
- Coupling data together and writing code that produces predictable access patterns to the data will be the most performant.
- Changing data layouts can yield more significant performance improvements than changing just the algorithms.
- Efficiency is obtained through algorithms but performance is obtained through data structures and layouts.

Reading

Data-Oriented Design and C++ - Mike Acton
Efficiency with Algorithms, Performance with Data Structures - Chandler Carruth

1.15 Interface And Composition Design

Design Philosophy

- Interfaces give programs structure.
- Interfaces encourage design by composition.
- Interfaces enable and enforce clean divisions between components.
    - The standardization of interfaces can set clear and consistent expectations.
- Decoupling means reducing the dependencies between components and the types they use.

    - This leads to correctness, quality and performance.
- Interfaces allow you to group concrete types by what they do.
    - Don't group types by a common DNA but by a common behavior.
    - Everyone can work together when we focus on what we do and not who we are.
- Interfaces help your code decouple itself from change.
    - You must do your best to understand what could change and use interfaces to decouple.
    - Interfaces with more than one method have more than one reason to change.
    - Uncertainty about change is not a license to guess but a directive to STOP and learn more.
- You must distinguish between code that:
    - defends against fraud vs protects against accidents

Validation

Use an interface when:

- users of the API need to provide an implementation detail.
- API's have multiple implementations they need to maintain internally.
- parts of the API that can change have been identified and require decoupling.

Don't use an interface:

- for the sake of using an interface.
- to generalize an algorithm.
- when users can declare their own interfaces.
- if it's not clear how the interface makes the code better.

Reading

Methods, interfaces and Embedding - William Kennedy
Composition with Go - William Kennedy
Reducing type hierarchies - William Kennedy
Application Focused API Design - William Kennedy
Avoid interface pollution - William Kennedy
Interface Values Are Valueless - William Kennedy
Interface Semantics - William Kennedy

1.16 Concurrent Software Design

Concurrency means "out of order" execution. Taking a set of instructions that would otherwise be executed in sequence and finding a way to execute them out of order and still produce the same result. For the problem in front of you, it has to be obvious that out of order execution would add value. When I say it adds value, I mean that it adds enough of a performance gain for the complexity cost. Depending on your problem, out of order execution may not be possible or even make sense.

It's also important to understand that concurrency is not the same as parallelism. Parallelism means executing two or more instructions at the same time. This is a different concept from concurrency. Parallelism is only possible when you have at least 2 operating system (OS) and hardware threads available to you and you have at least 2 Goroutines, each executing instructions independently on each OS/hardware thread.

Both you and the runtime have a responsibility in managing the concurrency of the application. You are responsible for managing these three things when writing concurrent software:

Design Philosophy

- The application must startup and shutdown with integrity.
    - Know how and when every goroutine you create terminates.
    - All goroutines you create should terminate before main returns.
    - Applications should be capable of shutting down on demand, even under load, in a controlled way.
        - You want to stop accepting new requests and finish the requests you have (load shedding).
- Identify and monitor critical points of back pressure that can exist inside your application.
    - Channels, mutexes and atomic functions can create back pressure when goroutines are required to wait.
    - A little back pressure is good, it means there is a good balance of concerns.
    - A lot of back pressure is bad, it means things are imbalanced.
    - Back pressure that is imbalanced will cause:
        - Failures inside the software and across the entire platform.
        - Your application to collapse, implode or freeze.
    - Measuring back pressure is a way to measure the health of the application.
- Rate limit to prevent overwhelming back pressure inside your application.
    - Every system has a breaking point, you must know what it is for your application.
    - Applications should reject new requests as early as possible once they are overloaded.
        - Don't take in more work than you can reasonably work on at a time.
        - Push back when you are at critical mass. Create your own external back pressure.
    - Use an external system for rate limiting when it is reasonable and practical.
- Use timeouts to release the back pressure inside your application.
    - No request or task is allowed to take forever.
    - Identify how long users are willing to wait.

    - Higher-level calls should tell lower-level calls how long they have to run.
    - At the top level, the user should decide how long they are willing to wait.
    - Use the Context package.
        - Functions that users wait for should take a Context.
            - These functions should select on <-ctx.Done() when they would otherwise block indefinitely.
        - Set a timeout on a Context only when you have good reason to expect that a function's execution has a real time limit.
        - Allow the upstream caller to decide when the Context should be canceled.
        - Cancel a Context whenever the user abandons or explicitly aborts a call.
- Architect applications to:
    - Identify problems when they are happening.
    - Stop the bleeding.
    - Return the system back to a normal state.

Reading
Scheduling In Go : Part I - OS Scheduler
Scheduling In Go : Part II - Go Scheduler
Scheduling In Go : Part III - Concurrency

1.17 Channel Design

Channels allow goroutines to communicate with each other through the use of signaling semantics. Channels accomplish this signaling through the use of sending/receiving data or by identifying state changes on individual channels. Don't architect software with the idea of channels being a queue, focus on signaling and the semantics that simplify the orchestration required.

Depending on the problem you are solving, you may require different channel semantics. Depending on the semantics you need, different architectural choices must be taken.

Language Mechanics
- Use channels to orchestrate and coordinate goroutines.
    - Focus on the signaling semantics and not the sharing of data.
    - Signaling with data or without data.
    - Question their use for synchronizing access to shared state.
        - There are cases where channels can be simpler for this but initially question.
- Unbuffered channels:
    - Receive happens before the Send.
    - Benefit: 100% guarantee the signal being sent has been received.
    - Cost: Unknown latency on when the signal will be received.

- Buffered channels:
    - Send happens before the Receive.
    - Benefit: Reduce blocking latency between signaling.
    - Cost: No guarantee when the signal being sent has been received.
        - The larger the buffer, the less guarantee.
        - Buffer of 1 can give you one delayed send of guarantee.
- Closing channels:
    - Close happens before the Receive. (like Buffered)
    - Signaling without data.
    - Perfect for signaling cancellations and deadlines.
- NIL channels:
    - Send and Receive block.
    - Turn off signaling
    - Perfect for rate limiting or short-term stoppages.

Design Philosophy

- If any given Send on a channel CAN cause the sending goroutine to block:
    - Be careful with Buffered channels larger than 1.
        - Buffers larger than 1 must have reason/measurements.
    - Must know what happens when the sending goroutine blocks.
- If any given Send on a channel WON'T cause the sending goroutine to block:
    - You have the exact number of buffers for each send.
        - Fan Out pattern
    - You have the buffer measured for max capacity.
        - Drop pattern
- Less is more with buffers.
    - Don’t think about performance when thinking about buffers.
    - Buffers can help to reduce blocking latency between signaling.
        - Reducing blocking latency towards zero does not necessarily mean better throughput.
        - If a buffer of one is giving you good enough throughput then keep it.
        - Question buffers that are larger than one and measure for size.
        - Find the smallest buffer possible that provides good enough throughput.

Chapter 2 - Language Mechanics

2.1 Built-in Types

Types provide integrity and readability by asking 2 questions:

- What is the amount of memory to allocate? (e.g. 1, 2, 4, 8 bytes)
- What does that memory represent? (e.g. int, uint, bool,..)

Types can be specific to a precision such as int32 or int64:

- uint8 represents an unsigned integer with 1 byte of allocation
- int32 represents a signed integer with 4 bytes of allocation

When I declare a type using a non-precision based type (unit, int) the size of the values constructed for these types are based on the architecture being used to build the program:

- 32 bit arch: int represents a signed int at 4 bytes of memory allocation
- 64 bit arch: int represents a signed int at 8 bytes of memory allocation

2.2 Word Size

The word size represents the amount of memory allocation required to store integers and pointers for a given architecture. For example:

- 32 bit arch: word size is 4 bytes of memory allocation
- 64 bit arch: word size is 8 bytes of memory allocation

This is important because Go has internal data structures (slices, interfaces) that store integers and pointers. The size of these data structures will be based on the architecture being used to build the program.

In Go, the amount of memory allocated for a value of type int, a pointer, or a word data, will always be the same.

2.3 Zero Value Concept

Every single value I construct in Go is initialized at least to its zero value state unless I specify the initialization value at construction. The zero value is the setting of every bit in every byte to zero.

This is done for data integrity and it's not free. It takes time to push electrons through the machine to reset those bits, but I should always take integrity over performance.

Listing 2.1

```
Type                Zero Value
Boolean             false
Integer             0
```

```
Floating            0
Complex             0i
String              "" (empty)
Pointer             nil
```

2.4 Declare and Initialize

The keyword var can be used to construct values for all types to their zero value state.

Listing 2.2

```
var a int
var b string
var c float64
var d bool

fmt.Printf("var a int \t %T [%v]\n", a, a)
fmt.Printf("var b string \t %T [%v]\n", b, b)
fmt.Printf("var c float64 \t %T [%v]\n", c, c)
fmt.Printf("var d bool \t %T [%v]\n\n", d, d)

Output:
var a int           int [0]
var b string        string []
var c float64       float64 [0]
var d bool          bool [false]
```

Strings use the UTF8 character set, but are really just a collection of bytes.

A string is a two-word internal data structure in Go:

- The first word represents a pointer to a backing array of bytes
- The second word represents the length or the number of bytes in the backing array
- If the string is set to its zero value state, then the first word is nil and the second word is 0.

Using the short variable declaration operator, I can declare, construct, and initialize a value all at the same time.

Listing 2.3

```
aa := 10       // int [10]
bb := "hello"  // string [hello]
cc := 3.14159  // float64 [3.14159]
dd := true     // bool [true]

fmt.Printf("aa := 10 \t %T [%v]\n", aa, aa)
fmt.Printf("bb := \"hello\" \t %T [%v]\n", bb, bb)
fmt.Printf("cc := 3.14159 \t %T [%v]\n", cc, cc)
fmt.Printf("dd := true \t %T [%v]\n\n", dd, dd)
```

2.5 Conversion vs Casting

Go doesn't have casting, but conversion. Instead of telling the compiler to map a set of bytes to a different representation, the bytes need to be copied to a new memory location for the new representation.

Listing 2.4

```
aaa := int32(10)
fmt.Printf("aaa := int32(10) %T [%v]\n", aaa, aaa)

Output:
aaa := int32(10) int32 [10]
```

2.6 Struct and Construction Mechanics

The declaration represents a concrete user defined type with a composite of different fields.

Listing 2.5

```
type example struct {
    flag    bool
    counter int16
    pi      float32
}
```

Declare a variable of type example and initialize it to its zero value state.

Listing 2.6

```
var e1 example

fmt.Printf("%+v\n", e1)

Output:
{flag:false counter:0 pi:0}
```

Declare a variable of type example not set to its zero value state by using literal construction syntax.

Listing 2.7

```
e2 := example{
    flag:    true,
    counter: 10,
    pi:      3.141592,
}

fmt.Println("Flag", e2.flag)
fmt.Println("Counter", e2.counter)
fmt.Println("Pi", e2.pi)

Output:
```

```
Flag true
Counter 10
Pi 3.141592
```

Declare a variable of an unnamed literal type set to its non-zero value state using literal construction syntax. This is a one-time thing.

Listing 2.7

```
e3 := struct {
    flag    bool
    counter int16
    pi      float32
}{
    flag:    true,
    counter: 10,
    pi:      3.141592,
}

fmt.Println("Flag", e3.flag)
fmt.Println("Counter", e3.counter)
fmt.Println("Pi", e3.pi)

Output:
Flag true
Counter 10
Pi 3.141592
```

The idea of literal construction is just that, to construct something literally and not to its zero value state. Because of this, I should use var for zero value and the short variable declaration operator with the { } syntax for non-zero value construction.

2.7 Padding and Alignment

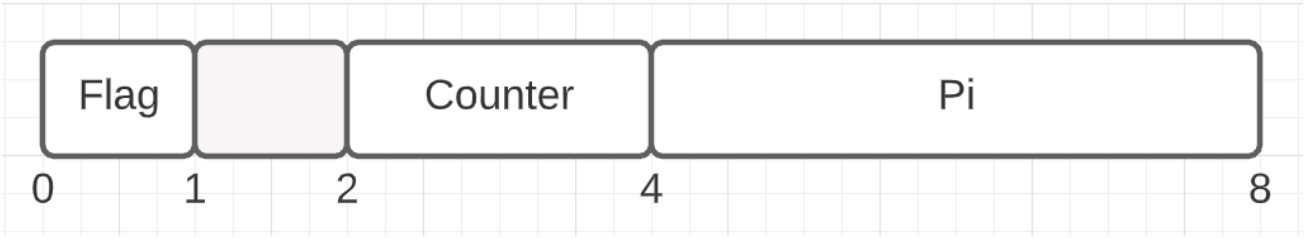
How much memory is allocated for a value of type example?

Listing 2.8

```
type example struct {
    flag    bool
    counter int16
    pi      float32
}
```

A bool is 1 byte, int16 is 2 bytes, float32 is 4 bytes. Add that all together and I get 7 bytes. However, the actual answer is 8 bytes. Why, because there is a padding byte sitting between the flag and counter fields for the reason of alignment.

Figure 2.1



The idea of alignment is to allow the hardware to read memory more efficiently by placing memory on specific alignment boundaries. The compiler takes care of the alignment boundary mechanics so I don't have to.

Depending on the size of a particular field, Go determines the alignment I need.

Listing 2.9

```
type example2 struct {
    flag    bool
    counter int16
    flag2   bool
    pi      float32
}
```

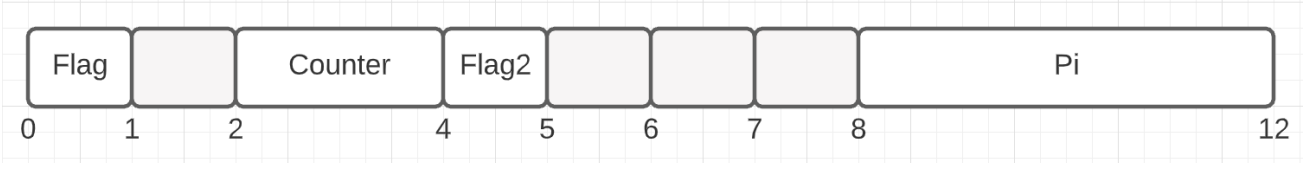
In this example, I've added a new field called flag2 between the counter and pi fields. This causes more padding inside the struct.

Listing 2.10

```
type example2 struct {
    flag    bool     // 0xc000100020 <- Starting Address
            byte     // 0xc000100021 <- 1 byte padding
    counter int16    // 0xc000100022 <- 2 byte alignment
    flag2   bool     // 0xc000100024 <- 1 byte alignment
            byte     // 0xc000100025 <- 1 byte padding
            byte     // 0xc000100026 <- 1 byte padding
            byte     // 0xc000100027 <- 1 byte padding
    pi      float32  // 0xc000100028 <- 4 byte alignment
}
```

This is how the alignment and padding play out if I pretend a value of type example2 starts at address 0xc000100020. The flag field represents the starting address and is only 1 byte in size. Since the counter field requires 2 bytes of allocation, it must be placed in memory on a 2-byte alignment, meaning it needs to fall on an address that is a multiple of 2. This means the counter field must start at address 0xc000100022. This creates a 1-byte gap between the flag and counter fields.

Figure 2.2



The flag2 field is a bool and can fall at the next address 0xc000100024. The final field is pi and requires 4 bytes of allocation so it needs to fall on a 4-byte alignment. The next address for a 4 byte value is at 0xc000100028. That means 3 more padding bytes are needed to maintain a proper alignment. This results in a value of type example2 requiring 12 bytes of total memory allocation.

The largest field in a struct represents the alignment boundary for the entire struct. In this case, the largest field is 4 bytes so the starting address for this struct value must be a multiple of 4. I can see the address 0xc000100020 is a multiple of 4.

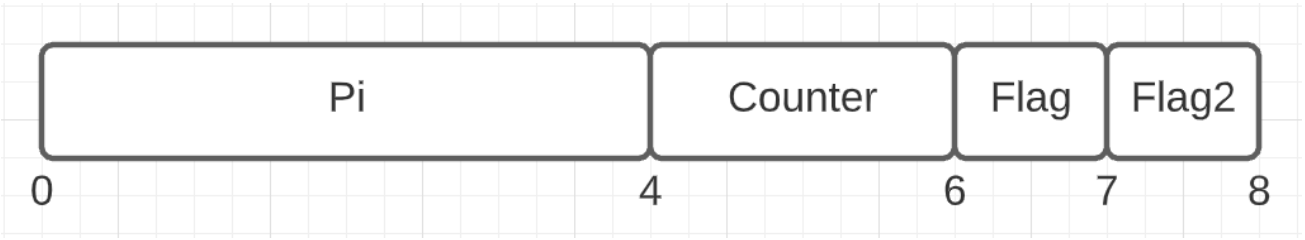
If I need to minimize the amount of padding bytes, I must lay out the fields from highest allocation to smallest allocation. This will push any necessary padding bytes down to the bottom of the struct and reduce the total number of padding bytes necessary.

Listing 2.10

```
type example struct {
    pi      float32  // 0xc000100020 <- Starting Address
    counter int16    // 0xc000100024 <- 2 byte alignment
    flag    bool     // 0xc000100026 <- 1 byte alignment
    flag2   bool     // 0xc000100027 <- 1 byte alignment
}
```

After the reordering of the fields, the struct value only requires 8 bytes of allocation and not 12 bytes. Since all the fields allow the struct value to fall on a 4-byte alignment, no extra padding bytes are necessary.

Figure 2.3



2.8 Assigning Values

If I have two different named types that are identical in structure, I can't assign a value of one to the other.

For example, if the types example1 and example2 are declared using the same exact declaration and we initialize two variables.

Listing 2.11

```
var ex1 example1
var ex2 example2
```

I can't assign these two variables to each other since they are of different named types. The fact that they are identical in structure is irrelevant.

Listing 2.12

```
ex1 = ex2  // Not allowed, compiler error
```

To perform any assignment, I would have to use conversion syntax and since they are identical in structure, the compiler will allow this.

Listing 2.13

```
ex1 = example1(ex2)  // Allowed, NO compiler error
```

However, if one of the variable's (like ex2) was declared as an unnamed type with the same structure, no conversion syntax would be required.

Listing 2.14

```
var ex2 struct {
    flag    bool
    counter int16
    pi      float32
}

ex1 = ex2  // Allowed, NO need for conversion syntax
```

The compiler will allow this assignment without the need for conversion.

2.9 Pointers

Pointers serve the purpose of sharing. Pointers allow me to share values across program boundaries. There are several types of program boundaries. The most common one is between function calls. There is also a boundary between Goroutines which I have notes for later.

When a Go program starts up, the Go runtime creates a Goroutine. Every Goroutine is a separate path of execution that manages the instructions that need to be executed by the machine. I can also think of Goroutines as lightweight application level threads because they are. Every Go program has at least 1 Goroutine called the main Goroutine.

Every Goroutine is given a block of memory, called stack memory. The memory for the stack starts out at 2K bytes. It's very small. Stacks can grow over time. Every time a function is called, a block of stack space is taken to help the Goroutine execute the

instructions associated with that function. Each individual block of memory is called a frame.

The size of a frame for a given function is calculated at compile time. No value can be constructed on the stack unless the compiler knows the size of that value at compile time. If the compiler doesn't know the size of a value at compile time, the value has to be constructed on the heap.

Stacks are self cleaning and zero value helps with the initialization of the stack. Every time I make a function call, and a frame of memory is blocked out, the memory for that frame is initialized, which is how the stack is self cleaning. On a function return, the memory for the frame is left alone since it's unknown if that memory will be needed again. It would be inefficient to initialize memory on returns.

2.10 Pass By Value

All data is moved around the program by value. This means as data is being passed across program boundaries, each function or goroutine is given it's own copy of the data. There are two types of data I work with, the value itself (int, string, user) or the value's address. Addresses are data that need to be copied and stored across program boundaries.

The following code attempts to explain this more.

Listing 2.15

```
// Declare variable of type int with a value of 10.
count := 10

// To get the address of a value, use the & operator.
println("count:\tValue Of[", count, "]\tAddr Of[", &count, "]")

// Pass a copy of the "value of" count (what's in the box)
// to the increment1 function.
increment1(count)

// Print out the "value of" and "address of" count.
// The value of count will not change after the function call.
println("count:\tValue Of[", count, "]\tAddr Of[", &count, "]")

// Pass a copy of the "address of" count (where is the box)
// to the increment2 function. This is still considered a pass by
// value and not a pass by reference because addresses are values.
increment2(&count)

// Print out the "value of" and "address of" count.
// The value of count has changed after the function call.
println("count:\tValue Of[", count, "]\tAddr Of[", &count, "]")

// increment1 declares the function to accept its own copy of
// and integer value.
```

```
func increment1(inc int) {

    // Increment the local copy of the caller's int value.
    inc++
    println("inc1:\tValue Of[", inc, "]\t
                    Addr Of[", &inc, "]")
}

// increment2 declares the function to accept its own copy of
// an address that points to an integer value.
// Pointer variables are literal types and are declared using *.
func increment2(inc *int) {

    // Increment the caller's int value through the pointer.
    *inc++
    println("inc2:\tValue Of[", inc, "]\t
                    Addr Of[", &inc, "]\t
                    Value Points To[", *inc, "]")
}

Output:
count:  Value Of[ 10 ]  Addr Of[ 0xc000050738 ]
inc1:   Value Of[ 11 ]  Addr Of[ 0xc000050730 ]
count:  Value Of[ 10 ]  Addr Of[ 0xc000050738 ]
inc2:   Value Of[ 0xc000050738 ] Addr Of[ 0xc000050748 ] Value Points To[ 11 ]
```

This post provides a four part series that explains the mechanics and design behind pointers, stacks, heaps, escape analysis and value/pointer semantics in Go.

https://www.ardanlabs.com/blog/2017/05/language-mechanics-on-stacks-and-pointers.html

2.11 Escape Analysis

I don't like the term "escape analysis" for the algorithm the compiler uses to determine if a value should be constructed on the stack or heap because it makes it sound like all values are constructed on the stack and then escape (or move) to the heap when necessary. This is NOT the case. The construction of any value only happens once, and the escape analysis algorithm decides where that will be (stack or heap). Only construction on the heap is called an allocation in Go.

Understanding escape analysis is about understanding value ownership. The idea is, when a value is constructed within the scope of a function, then that function owns the value. From there ask the question, does the value being constructed still have to exist when the owning function returns? If the answer is no, the value can be constructed on the stack. If the answer is yes, the value must be constructed on the heap.

Note: Escape analysis does have flaws and there are other reasons why a value might be constructed on the heap, but for now this general rule is a good starting point.

Listing 2.16

```
// user represents a user in the system.
type user struct {
    name  string
    email string
}

func stayOnStack() user {
    u := user{
        name:  "Hoanh An",
        email: "hoanhan101@gmail.com",
    }

    return u
}
```

The stayOnStack function is using value semantics to return a user value back to the caller. In other words, the caller gets their own copy of the user value being constructed.

When stayOnStack returns, the user value it constructs no longer needs to exist, since the caller is getting their own copy. Therefore, the construction of the user value inside of stayOnStack can happen on the stack. No allocation.

Listing 2.17

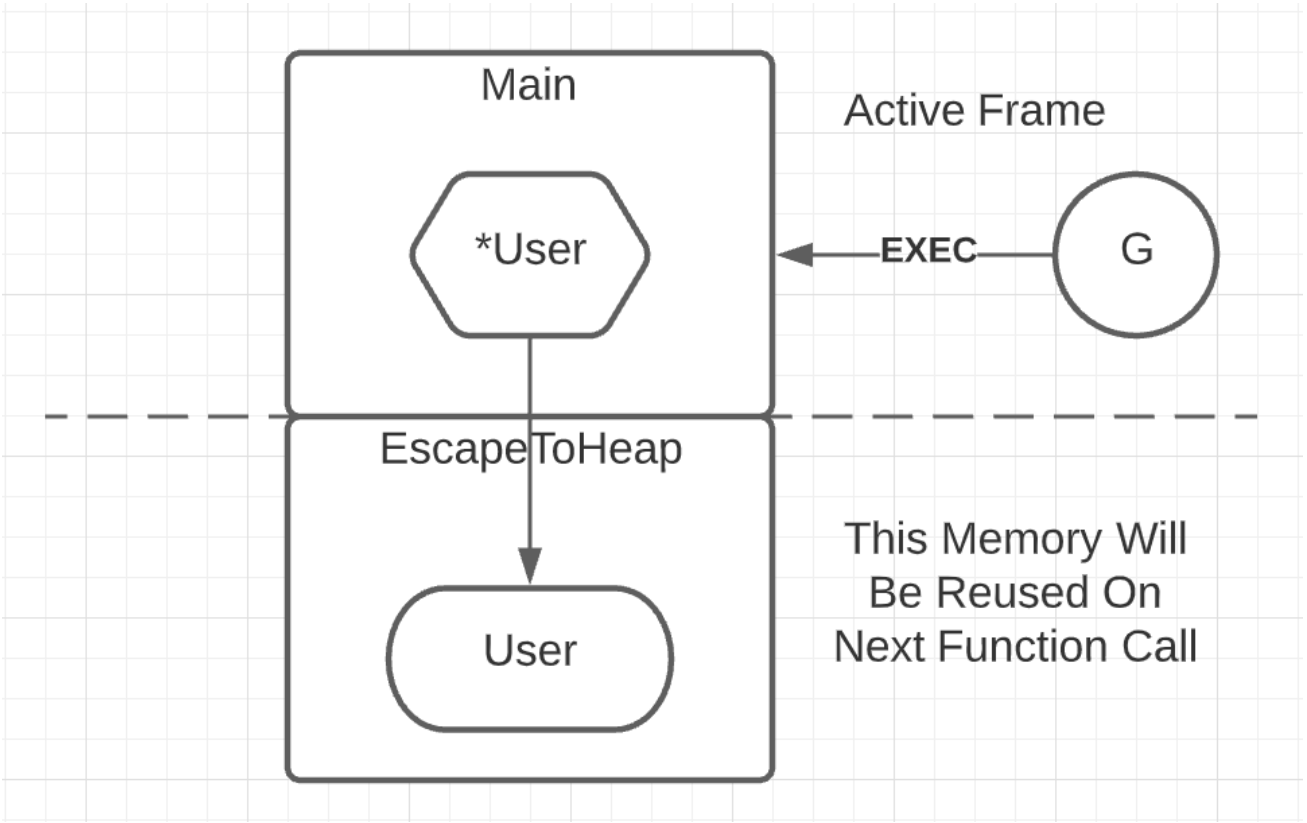
```
// user represents a user in the system.
type user struct {
    name  string
    email string
}

func escapeToHeap() *user {
    u := user{
        name:  "Hoanh An",
        email: "hoanhan101@gmail.com",
    }

    return &u
}
```

The escapeToHeap function is using pointer semantics to return a user value back to the caller. In other words, the caller gets shared access (an address) to the user value being constructed.

When escapeToHeap returns, the user value it constructs does still need to exist, since the caller is getting shared access to the value. Therefore, the construction of the user value inside of escapeToHeap can’t happen on the stack, it must happen on the heap. Yes allocation.

Think about what would happen if the value was constructed on the stack when using pointer semantics on the return.

Figure 2.4



The caller would get a copy of a stack address from the frame below. Integrity would be lost. Once control goes back to the calling function, the memory on the stack where the user value exists is reusable again. The moment the calling function makes another function call, a new frame is sliced and the memory will be overridden, destroying the shared value.

This is why I think about the stack being self cleaning. Zero value initialization helps every stack frame that I need to be cleaned without the use of GC. The stack is self cleaning since a frame is taken and initialized for the execution of each function call. The stack is cleaned during function calls and not on returns because the compiler doesn't know if that memory on the stack will ever be needed again.

Escape analysis decides if a value is constructed on the stack (the default) or the heap (the escape). With the stayOnStack function, I'm passing a copy of the value back to the caller, so it's safe to keep the value on the stack. With the escaoeToHeap function, I'm passing a copy of the values address back to the caller (sharing up the stack) so it's not safe to keep the value on the stack.

To learn more read this post:

https://www.ardanlabs.com/blog/2017/05/language-mechanics-on-escape-analysis.html

2.12 Stack Growth

The size of each frame for every function is calculated at compile time. This means, if the compiler doesn't know the size of a value at compile time, the value must be constructed on the heap. An example of this is using the built-in function make to construct a slice whose size is based on a variable.

Listing 2.18

```
b := make([]byte, size) // Backing array allocates on the heap.
```

Go uses a contiguous stack vs using a segmented stack mechanic. In general, these are mechanics that dictate how stacks grow and shrink.

Every function call comes with a little preamble that asks, "Is there enough stack space for this new frame?". If yes, then no problem and the frame is taken and initialized. If not, then a new larger stack must be constructed and the memory on the existing stack must be copied over to the new one, with changes to pointers that reference memory on the stack. The benefits of contiguous memory and linear traversals with modern hardware is the tradeoff for the cost of the copy.

Because of the use of contiguous stacks, no Goroutine can have a pointer to some other Goroutine's stack. There would be too much overhead for the runtime to keep track of every pointer to every stack and readjust those pointers to the new location.

2.13 Garbage Collection

Once a value is constructed on the heap, the Garbage Collector (GC) has to get involved. The most important part of the GC is the pacing algorithm. It determines the frequency/pace that the GC has to run in order to maintain the smallest heap possible in conjunction with the best application throughput.

There are lots of little details related to the GC and they are best understood in this blog post series.

https://www.ardanlabs.com/blog/2018/12/garbage-collection-in-go-part1-semantics.html

2.14 Constants

One of the more unique features of Go is how the language implements constants. The rules for constants in the language specification are unique to Go. They provide the flexibility Go needs to make the code we write readable and intuitive while still maintaining type safety.

Constants can be typed or untyped. When a constant is untyped, it's considered to be of a kind. Constants of a kind can be implicitly converted by the compiler. This all happens at compile time and not at runtime.

Untyped numeric constants have a precision of 256 bits.

Listing 2.19

```
const ui = 12345     // kind: integer
const uf = 3.141592  // kind: floating-point
```

Typed constants still use the constant type system, but their precision is restricted.

Listing 2.20

```
const ti int     = 12345     // type: int
const tf float64 = 3.141592  // type: float64
```

This doesn't work because the number 1000 is too large to store in an uint8.

Listing 2.21

```
const myUint8 uint8 = 1000
```

Constant arithmetic supports the use of different kinds of constants. Kind Promotion is used to handle these different scenarios. All of this happens implicitly.

Listing 2.22

```
var answer = 3 * 0.333  // KindFloat(3) * KindFloat(0.333)
```

The answer variable will be of type float64 and represent 0.999 at a precision of 64 bits.

Listing 2.23

```
const third = 1 / 3.0  // KindFloat(1) / KindFloat(3.0)
```

The third constant will be of kind floating point and represent 1/3 at a precision of 256 bits.

Listing 2.24

```
const zero = 1 / 3  // KindInt(1) / KindInt(3)
```

The zero constant will be of kind integer and set to 0.

Listing 2.25

```
const one int8 = 1
const two = 2 * one  // int8(2) * int8(1)
```

This is an example of constant arithmetic between typed and untyped constants. In this case a constant of a type promotes over a constant of a kind. The two constant will

be of type int8 and set to 2.

Listing 2.26

```
const maxInt = 9223372036854775807
```

This is the max integer value for a 64 bit integer.

Listing 2.27

```
const bigger = 92233720368547758085435223345
```

The bigger constant is a much larger value than a 64 bit integer, but it can be stored in a constant of kind int since constants of kind int are not limited to 64 bits of precision.

Listing 2.28

```
const bigger int64 = 92233720368547758085435223345

Compiler Error:
  constant 92233720368547758085435223345 overflows int64
```

However, if bigger was a constant of type int64, this would not compile.

To learn more read this post:
https://www.ardanlabs.com/blog/2014/04/introduction-to-numeric-constants-in-go.html

2.15 IOTA

IOTA provides nice support for setting numeric constants.

Listing 2.29

```
const (
    A1 = iota  // 0 : Start at 0
    B1 = iota  // 1 : Increment by 1
    C1 = iota  // 2 : Increment by 1
)
fmt.Println(A1, B1, C1)

Output:
0 1 2
```

The iota keyword works within a constant block and starts with the value of 0. Then for each next constant declared, iota increments by 1.

Listing 2.30

```
const (
    A2 = iota  // 0 : Start at 0
    B2         // 1 : Increment by 1
```

```
    C2          // 2 : Increment by 1
)
fmt.Println(A2, B2, C2)

Output:
0 1 2
```

I don’t need to repeat the use of the iota keyword. It is assumed once applied.

Listing 2.31

```
const (
    A3 = iota + 1  // 1 : Start at 0 + 1
    B3             // 2 : Increment by 1
    C3             // 3 : Increment by 1
)
fmt.Println(A3, B3, C3)

Output:
1 2 3
```

If I didn’t want to start with 0 for the first constant, perform some math and the incrementation will take place automatically.

Listing 2.32

```
const (
    Ldate= 1 << iota  //  1 : Shift 1 to the left 0.  0000 0001
    Ltime             //  2 : Shift 1 to the left 1.  0000 0010
    Lmicroseconds     //  4 : Shift 1 to the left 2.  0000 0100
    Llongfile         //  8 : Shift 1 to the left 3.  0000 1000
    Lshortfile        // 16 : Shift 1 to the left 4.  0001 0000
    LUTC              // 32 : Shift 1 to the left 5.  0010 0000
)

fmt.Println(Ldate, Ltime, Lmicroseconds, Llongfile, Lshortfile, LUTC)

Output:
1 2 4 8 16 32
```

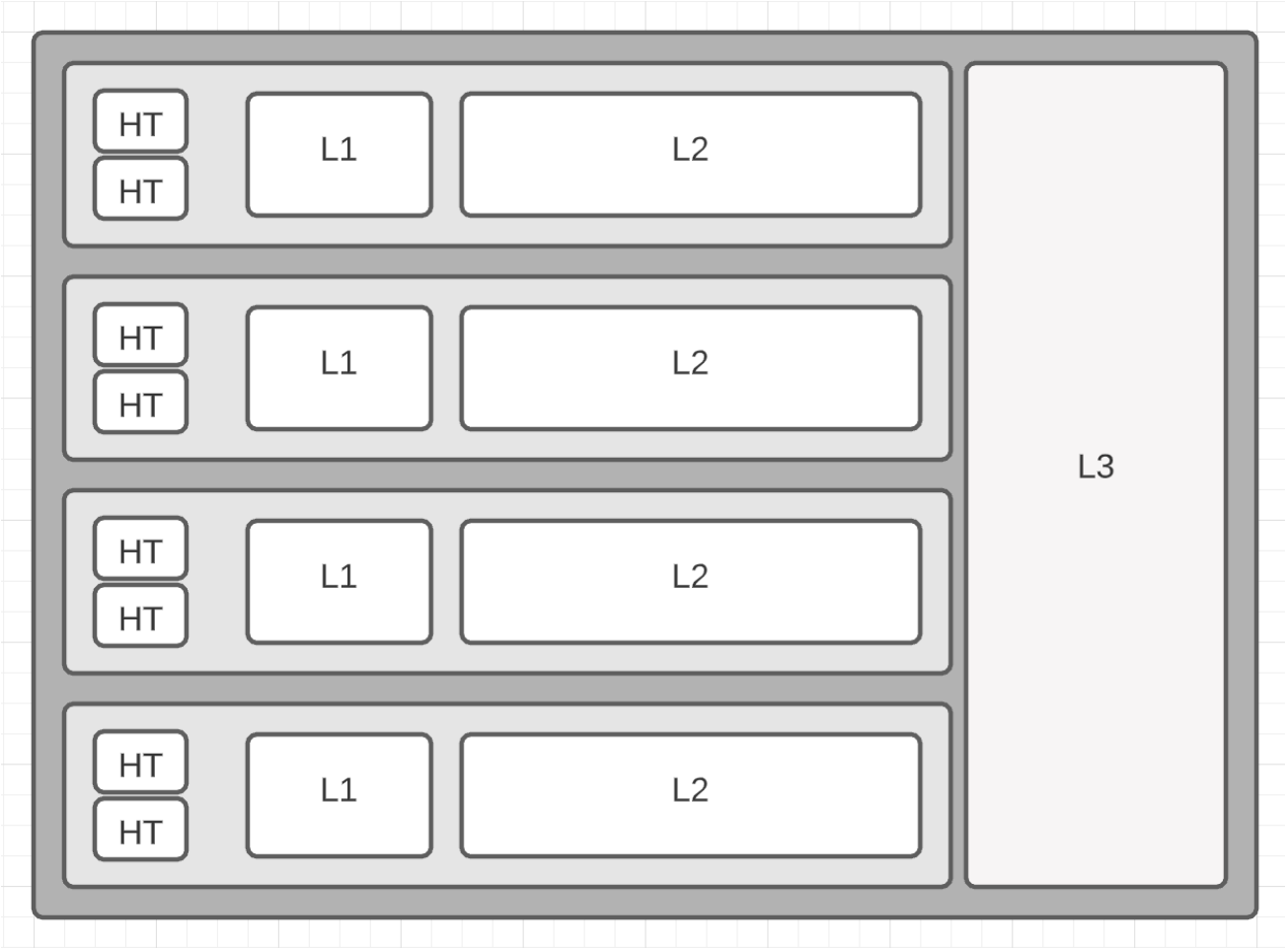
I can use this feature like the Log package does for these flags. In this case, bit operations are being used to create flag values.

Chapter 3 - Data Structures

3.1 CPU Caches

Each core inside the processor has its own local cache of memory (L1 and L2) and a common cache of memory (L3) used to store/access data and instructions. The hardware threads in each core can only access the local L1 and L2 caches. Data from L3 or main memory need to be copied into the L1 or L2 cache for access.

Figure 3.1



The latency cost of accessing data that exists in the different caches changes from least to most: L1 -> L2 -> L3 -> main memory. As Scott Meyers said, "If performance matters then the total amount of memory I have is the total amount of cache. Main memory is so slow to access, practically speaking, it might as well not even be there."

Performance today is about how efficiently data flows through the hardware. If every piece of data the hardware needs (at any given time) exists only in main memory, my programs would run slower as compared to the data already being present in the L1 or L2 caches.

Listing 3.1

```
3GHz(3 clock cycles/ns) * 4 instructions per cycle = 12 instructions per ns!

1 ns ............. 1 ns ............. 12 instructions  (one)
1 µs .......... 1000 ns .......... 12,000 instructions  (thousand)
1 ms ..... 1,000,000 ns ...... 12,000,000 instructions  (million)
1 s .. 1,000,000,000 ns .. 12,000,000,000 instructions  (billion)

Industry Defined Latencies
L1 cache reference ......................... 0.5 ns ...................  6 ins
L2 cache reference ........................... 7 ns .................. 84 ins
Main memory reference ...................... 100 ns ................. 1200 ins
```

How do I write code where I can expect the data the hardware needs (at any given time) to already be present in the L1 or L2 caches? I need to write code that is mechanically sympathetic with the processor's prefetcher. The prefetcher attempts to predict what data is needed before instructions request the data so it's already present in either the L1 or L2 cache.

There are different granularities of memory access depending on where the access is happening. My code can read/write a byte as the smallest unit of memory access. However, from the caching systems point of view, the granularity is 64 bytes. This 64 byte block of memory is called a cache line.

The Prefetcher works best when the instructions being executed create predictable access patterns to memory. One way to create a predictable access pattern to memory is to construct a contiguous block of memory and then iterate over that memory in a linear traversal. That creates a predictable stride across the memory.

The array is the most important data structure to the hardware because it supports predictable access patterns. However, the slice is the most important data structure in Go. Slices in Go use an array underneath.

Once I construct an array, whatever its size, every element is equally distant from the next or previous. As I iterate over an array, I begin to walk cache line by connected cache line in a predictable stride. The Prefetcher will pick up on this predictable data access pattern and begin to efficiently get the data into the processor, thus reducing data access latency costs.

Imagine I have a big square matrix of memory and a linked list of nodes that match the number of elements in the matrix. If I perform a traversal across the linked list, and then traverse the matrix in both directions (Column and Row), how will the performance of the different traversals compare?

Listing 3.2

```
func RowTraverse() int {
    var ctr int
```

```
    for row := 0; row < rows; row++
        for col := 0; col < cols; col++ {
            if matrix[row][col] == 0xFF {
                ctr++
            }
        }
    }
    return ctr
}
```

Row traverse will have the best performance because it walks through memory, cache line by connected cache line, which creates a predictable access pattern. Cache lines can be prefetched and copied into the L1 or L2 cache before the data is needed.

Listing 3.3

```
func ColumnTraverse() int {
    var ctr int
    for col := 0; col < cols; col++ {
        for row := 0; row < rows; row++ {
            if matrix[row][col] == 0xFF {
                ctr++
            }
        }
    }
    return ctr
}
```

Column Traverse is the worst by an order of magnitude because this access pattern crosses over OS page boundaries on each memory access. This causes no predictability for cache line prefetching and becomes essentially random access memory.

Listing 3.4

```
func LinkedListTraverse() int {
    var ctr int
    d := list
    for d != nil {
        if d.v == 0xFF {
            ctr++
        }
        d = d.p
    }
    return ctr
}
```

The linked list is twice as slow as the row traversal mainly because there are cache line misses but fewer TLB misses (explained next). A bulk of the nodes connected in the list exist inside the same OS pages.

Listing 3.5

```
BenchmarkLinkListTraverse-16    128      28738407 ns/op
BenchmarkColumnTraverse-16       30     126878630 ns/op
```

```
BenchmarkRowTraverse-16         310      11060883 ns/op
```

3.2 Translation Lookaside Buffer (TLB)

Each running program is given a full memory map of virtual memory by the OS and that running program thinks they have all of the physical memory on the machine. However, physical memory needs to be shared with all the running programs. The operating system shares physical memory by breaking the physical memory into pages and mapping pages to virtual memory for any given running program. Each OS can decide the size of a page, but 4k, 8k, 16k are reasonable and common sizes.

The TLB is a small cache inside the processor that helps to reduce latency on translating a virtual address to a physical address within the scope of an OS page and offset inside the page. A miss against the TLB cache can cause large latencies because now the hardware has to wait for the OS to scan its paging table to locate the right page for the virtual address in question. If the program is running on a virtual machine (like in the cloud) then the virtual machine paging table needs to be scanned first.

Remember when I said:

The linked list is twice as slow as the row traversal mainly because there are cache line misses but fewer TLB misses (explained next). A bulk of the nodes connected in the list exist inside the same OS pages.

The LinkedList is orders of magnitude faster than the column traversal because of TLB access. Even though there are cache line misses with the linked list traversal, since a majority of the memory for a group of nodes will land inside the same page, TLB latencies are not affecting performance. This is why for programs that use a large amount of memory, like DNA based applications, I may want to use a distribution of linux that is configured with page sizes in the order of a meg or two of memory.

All that said, data-oriented design matters. Writing an efficient algorithm has to take into account how the data is accessed. Remember, performance today is about how efficiently I can get data into the processor.

3.3 Declaring and Initializing Values

Declare an array of five strings initialized to its zero value state.

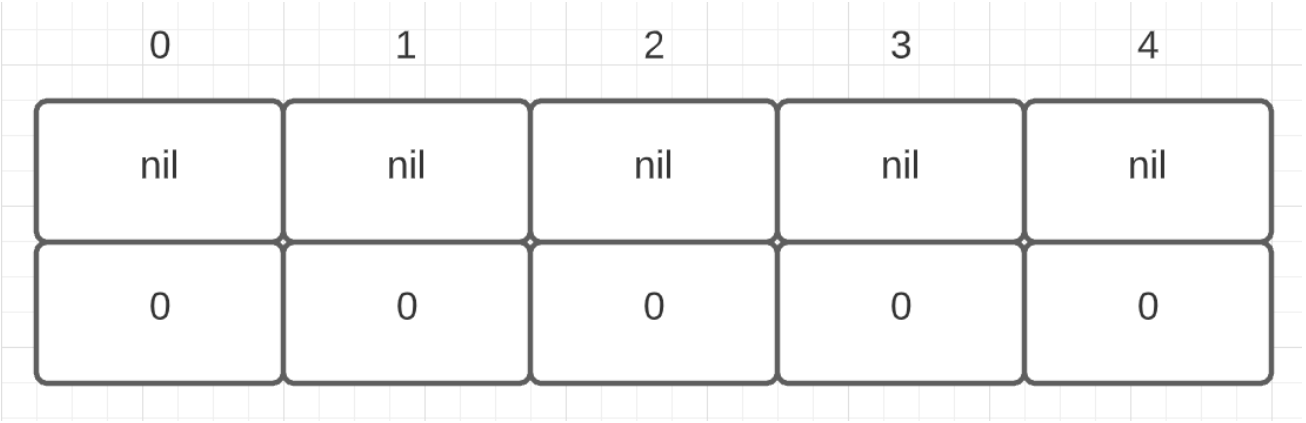
Listing 3.6

```
var strings [5]string
```

A string is a 2 word data structure representing a pointer to a backing array of bytes and the total number of bytes in the backing array. Since this array is set to its zero

value state, every element is set to its zero value state. This means that each string has the first word set to nil and the second word set to 0.

Figure 3.2



3.4 String Assignments

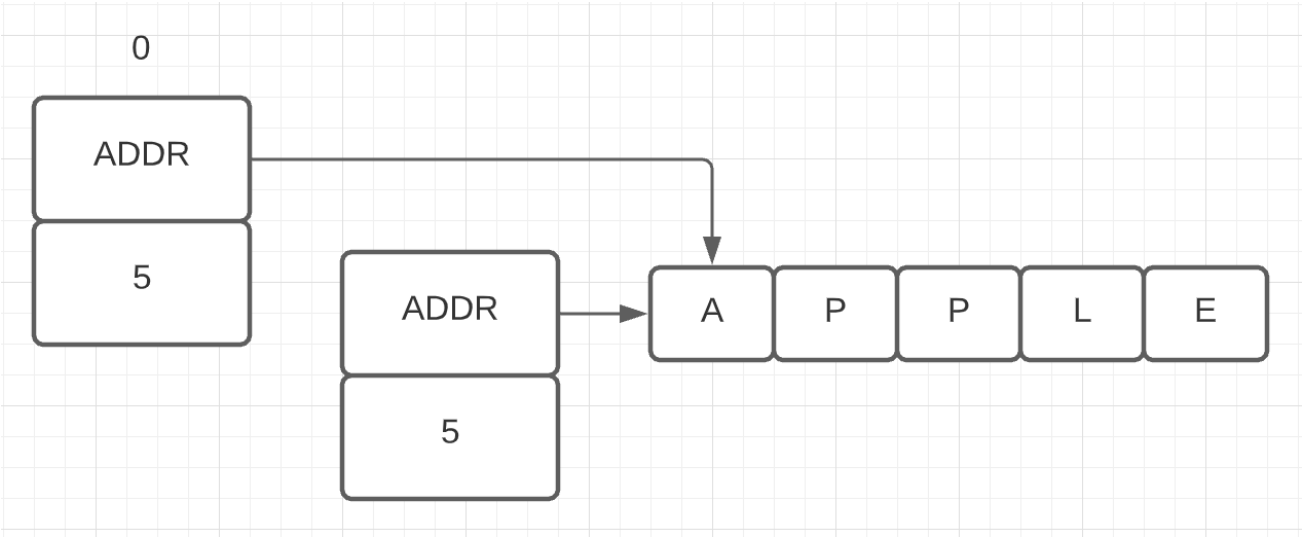
What happens when a string is assigned to another string?

Listing 3.7

```
strings[0] = "Apple"
```

When a string is assigned to another string, the two word value is copied, resulting in two different string values both sharing the same backing array.

Figure 3.3



The cost of copying a string is the same regardless of the size of a string, a two word copy.

3.5 Iterating Over Collections

Go provides two different semantics for iterating over a collection. I can iterate using value semantics or pointer semantics.

Listing 3.8

```
// Value Semantic Iteration
for i, fruit := range strings {
    println(i, fruit)
}


// Pointer Semantic Iteration
for i := range strings {
    println(i, strings[i])
}
```

When using value semantic iteration, two things happen. First, the value I'm iterating over is copied and I iterate over that copy. So if the value is an array, the array is copied and if the value is a slice, the slice value is copied. Second, I get a copy of each element being iterated on. When using pointer semantic iteration, I iterate over the original collection and I access each element associated with the collection directly.

3.6 Value Semantic Iteration

Given the following code and output.

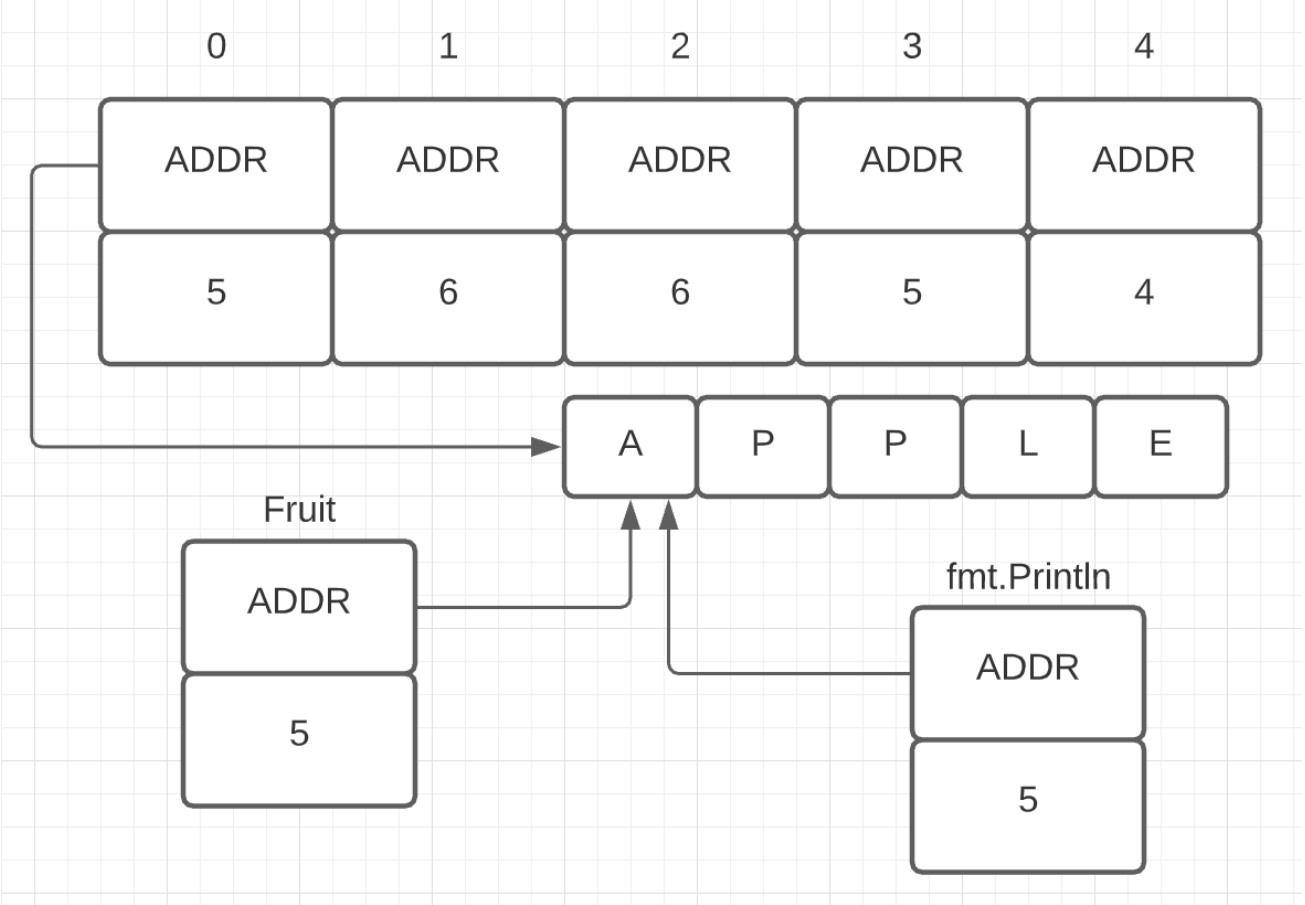
Listing 3.9

```
strings := [5]string{"Apple", "Orange", "Banana", "Grape", "Plum"}
for i, fruit := range strings {
    println(i, fruit)
}

Output:
0 Apple
1 Orange
2 Banana
3 Grape
4 Plum
```

The strings variable is an array of 5 strings. The loop iterates over each string in the collection and displays the index position and the string value. Since this is value semantic iteration, the for range is iterating over its own shallow copy of the array and on each iteration the fruit variable is a copy of each string (the two word data structure).

Notice how the fruit variable is passed to the print function using value semantics. The print function is getting its own copy of the string value as well.

Figure 3.4



By the time the string is passed to the print function, there are 4 copies of the string value (array, shallow copy, fruit variable and the print function's copy). All 4 copies are sharing the same backing array of bytes. Making copies of the string value is important because it prevents the string value from ever escaping to the heap. This eliminates non-productive allocation on the heap.

3.7 Pointer Semantic Iteration

Given the following code and output.

Listing 3.10

```
strings := [5]string{"Apple", "Orange", "Banana", "Grape", "Plum"}
for i := range strings {
    println(i, strings[i])
}

Output:
0 Apple
1 Orange
2 Banana
3 Grape
4 Plum
```

Once again, the strings variable is an array of 5 strings. The loop iterates over each string in the collection and displays the index position and the string value. Since this is pointer semantic iteration, the for range is iterating over the strings array directly and on each iteration, the string value for each index position is accessed directly for the print call.

3.8 Data Semantic Guideline For Built-In Types

As a guideline, if the data I'm working with is a numeric, string, or bool, then use value semantics to move the data around my program. This includes declaring fields on a struct type.

Listing 3.11

```
func Foo(x int, y string, z bool) (int, string, bool)

type Foo struct {
    X int
    Y string
    Z bool
}
```

One reason I might take an exception and use pointer semantics is if I need the semantic of NULL (absence of value). Then using pointers of these types is an option, but document this if it's not obvious.

The nice thing about using value semantics for these types is that I'm guaranteed that each function is operating on its own copy. This means reads and writes to this data are isolated to that function. This helps with integrity and identifying bugs related to data corruption.

3.9 Different Type Arrays

Declare an array of 4 and 5 integers initialized to its zero value state. Then try to assign them to each other.

Listing 3.12

```
var five [5]int
four := [4]int{10, 20, 30, 40}

five = four

Compiler Error:
cannot use four (type [4]int) as type [5]int in assignment
```

When I try to assign variable four to variable five, the compiler says that "cannot use four (type [4]int) as type [5]int in assignment". It's important to be clear about what the compiler is saying. It's saying that an array of 4 integers and an array of 5 integers represent data of different types. The size of an array is part of its type information. In

Go, the size of an array has to be known at compile time.

3.10 Contiguous Memory Construction

Declare an array of 5 strings initialized with values. Then use value semantic iteration to display information about each string.

Listing 3.13

```
five := [5]string{"Annie", "Betty", "Charley", "Doug", "Hoanh"}

for i, v := range five {
    fmt.Printf("Value[%s]\tAddress[%p]  IndexAddr[%p]\n",
        v, &v, &five[i])
}

Output:
Value[Annie]     Address[0xc000010250]    IndexAddr[0xc000052180]
Value[Betty]     Address[0xc000010250]    IndexAddr[0xc000052190]
Value[Charley]   Address[0xc000010250]    IndexAddr[0xc0000521a0]
Value[Doug]      Address[0xc000010250]    IndexAddr[0xc0000521b0]
Value[Hoanh]     Address[0xc000010250]    IndexAddr[0xc0000521c0]
```

The output shows each individual string value, the address of the v variable and the address of each element in the array. I can see how the array is a contiguous block of memory and how a string is a two word or 16 byte data structure on my 64 bit architecture. The address for each element is distanced on a 16 byte stride.

The fact that the v variable has the same address on each iteration strengthens the understanding that v is a local variable of type string which contains a copy of each string value during iteration.

3.11 Constructing Slices

The slice is Go's most important data structure and it's represented as a three word data structure.

First Word:	Holds a pointer to a backing array of data of a declared type.
Second Word:	Length of the slice which represents the number of elements that can be accessed from the backing array.
Third Word:	Capacity of the slice which represents the total number of elements that exist in the backing array.

Constructing a slice can be done in several ways.

Listing 3.14

```
// Slice of string set to its zero value state.
var slice []string
slice := make([]string)
```

```
// Slice of string set to its empty state.
slice := []string{}

// Slice of string set with a length and capacity of 5.
slice := make([]string, 5)

// Slice of string set with a length of 5 and capacity of 8.
slice := make([]string, 5, 8)

// Slice of string set with values with a length and capacity of 5.
slice := []string{"A", "B", "C", "D", "E"}
```

I can see the built-in function `make` allows me to pre-allocate both length and capacity for the backing array. If the compiler knows the size at compile time, the backing array could be constructed on the stack.

3.12 Slice Length vs Capacity

The length of a slice represents the number of elements that can be read and written to. The capacity represents the total number of elements that exist in the backing array from that pointer position.

Because of some syntactic sugar, slices look and feel like an array.

```
slice := make([]string, 5)
slice[0] = "Apple"
slice[1] = "Orange"
slice[2] = "Banana"
slice[3] = "Grape"
slice[4] = "Plum"
```

I can tell the difference between slice and array construction since an array has a known size at compile time and slices necessarily do not.

If I try to access an element beyond the slice's length, I will get a runtime error.

```
slice := make([]string, 5)
slice[5] = "Raspberry"

Compiler Error:
Error: panic: runtime error: index out of range slice[5] = "Runtime error"
```

3.13 Data Semantic Guideline For Slices

As a guideline, if the data I'm working with is a slice, then use value semantics to move the data around my program. This includes declaring fields on a type.

```
func Foo(data []byte) []byte

type Foo struct {
```

```
    X []int
    Y []string
    Z []bool
}
```

This goes for all of Go's internal data structures (slices, maps, channels, interfaces, and functions).

One reason to switch to pointer semantics is if I need to share the slice for a decoding or unmarshaling operation. Using pointers for these types of operations are ok, but document this if it's not obvious.

3.14 Contiguous Memory Layout

The idea behind the slice is to have an array, which is the most efficient data structure as it relates to the hardware. However, I still need the ability to be dynamic and efficient with the amount of data I need at runtime and future growth.

```
slice := make([]string, 5, 8)
slice[0] = "Apple"
slice[1] = "Orange"
slice[2] = "Banana"
slice[3] = "Grape"
slice[4] = "Plum"

inspectSlice(slice)

func inspectSlice(slice []string) {
    fmt.Printf("Length[%d] Capacity[%d]\n", len(slice), cap(slice))
    for i := range slice {
        fmt.Printf("[%d] %p %s\n", i, &slice[i], slice[i])
    }
}

Output:
Length[5] Capacity[8]
[0] 0xc00007e000 Apple
[1] 0xc00007e010 Orange
[2] 0xc00007e020 Banana
[3] 0xc00007e030 Grape
[4] 0xc00007e040 Plum
```

The inspectSlice function shows how a slice does have a contiguous backing array with a predictable stride. It also shows how a slice has a length and capacity which may be different. Notice how the print function only iterates over the length of a slice.

3.15 Appending With Slices

The language provides a built-in function called append to add values to an existing slice.

```
var data []string

for record := 1; record <= 102400; record++ {
    data = append(data, fmt.Sprintf("Rec: %d", record))
}
```

The append function works with a slice even when the slice is initialized to its zero value state. The api design of append is what's interesting because it uses value semantic mutation. Append gets its own copy of a slice value, it mutates its own copy, then it returns a copy back to the caller.

Why is the api designed this way? This is because the idiom is to use value semantics to move a slice value around a program. This must still be respected even with a mutation operation. Plus, value semantic mutation is the safest way to perform mutation since the mutation is being performed on the function's own copy of the data in isolation.

Append always maintains a contiguous block of memory for the slice's backing array, even after growth. This is important for the hardware. Every time the append function is called, the function checks if the length and capacity of the slice is the same or not. If it's the same, it means there is no more room in the backing array for the new value. In this case. append creates a new backing array (doubling or growing by 25%) and then copies the values from the old array into the new one. Then the new value can be appended.

If it's not the same, it means that there is an extra element of capacity existing for the append. An element is taken from capacity and added to the length of the slice. This makes an append operation very efficient. When the backing array has 1024 elements of capacity or less, new backing arrays are constructed by doubling the size of the existing array. Once the backing array grows past 1024 elements, growth happens at 25%.

3.16 Slicing Slices

Slices provide the ability to avoid extra copies and heap allocations of the backing array when needing to isolate certain elements of the backing array for different operations.

Slicing a slice uses the following syntax `[a:b]` where a is the starting index position for the first element of the new slice and b represents the ending index which is not included in the new length of the slice.

```
slice1 := []string{"A", "B", "C", "D", "E"}
slice2 := slice1[2:4]
```

The variable slice2 is a new slice value that is now sharing the same backing array that slice1 is using. However, slice2 only allows me to access the elements at index 2 and 3 (C and D) of the original slice's backing array. The length of slice2 is 2 and not 5 like in slice1.

The slicing syntax represents the notation [a:b) index a through b not including the element at index b.

A better way to think about slicing is to focus on the length using this notation [a:a+len] index a through a plus the length. This will reduce errors in calculating new slices.

Using this inspect function.

```
func inspectSlice(slice []string) {
    fmt.Printf("Length[%d] Capacity[%d]\n", len(slice), cap(slice))
    for i, s := range slice {
        fmt.Printf("[%d] %p %s\n",
            i,
            &slice[i],
            s)
        }
    }
}
```

I can see this in action.

```
slice1 := []string{"A", "B", "C", "D", "E"}
slice2 := slice1[2:4]
inspectSlice(slice1)
inspectSlice(slice2)

Output:
Length[5] Capacity[5]
[0] 0xc00007e000 A
[1] 0xc00007e010 B
[2] 0xc00007e020 C
[3] 0xc00007e030 D
[4] 0xc00007e040 E
Length[2] Capacity[3]
[0] 0xc00007e020 C
[1] 0xc00007e030 D
```

Notice how the two different slices are sharing the same backing array. I can see this by comparing addresses.

The nice thing here is there are no allocations. The compiler knows the size of the backing array for slice1 at compile time. Passing a copy of the slice value down into the inspectSlice function keeps everything on the stack.

3.17 Mutations To The Backing Array

When I use slice2 to change the value of the string at index 0, any slice value that is sharing the same backing array (where the address for that index is part of that slice's length) will see the change.

```
slice1 := []string{"A", "B", "C", "D", "E"}
slice2 := slice1[2:4]
slice2[0] = "CHANGED"
inspectSlice(slice1)
inspectSlice(slice2)

Output:
Length[5] Capacity[5]
[0] 0xc00007e000 A
[1] 0xc00007e010 B
[2] 0xc00007e020 CHANGED
[3] 0xc00007e030 D
[4] 0xc00007e040 E
Length[2] Capacity[3]
[0] 0xc00007e020 CHANGED
[1] 0xc00007e030 D
```

I always have to be aware when I am modifying a value at an index position if the backing array is being shared with another slice.

What if I use the built-in function append instead?

```
slice1 := []string{"A", "B", "C", "D", "E"}
slice2 := slice1[2:4]
slice2 = append(slice2, "CHANGED")
inspectSlice(slice1)
inspectSlice(slice2)

Output:
Length[5] Capacity[5]
[0] 0xc00007e000 A
[1] 0xc00007e010 B
[2] 0xc00007e020 C
[3] 0xc00007e030 D
[4] 0xc00007e040 CHANGED
Length[3] Capacity[3]
[0] 0xc00007e020 C
[1] 0xc00007e030 D
[2] 0xc00007e040 CHANGED
```

The append function creates the same side effect, but it's hidden. In this case, bringing in more length from capacity for slice2 has caused the value at address 0xc00007e040 to be changed. Unfortunately, slice1 had this address already as part of its length.

One way to avert the side effect is to use a three index slice when constructing slice2 so the length and capacity is the same at 2.

```
slice1 := []string{"A", "B", "C", "D", "E"}
slice2 := slice1[2:4:4]
inspectSlice(slice1)
inspectSlice(slice2)

Output:
Length[5] Capacity[5]
[0] 0xc00007e000 A
[1] 0xc00007e010 B
[2] 0xc00007e020 C
[3] 0xc00007e030 D
[4] 0xc00007e040 E
Length[2] Capacity[2]
[0] 0xc00007e020 C
[1] 0xc00007e030 D
```

The syntax for a three index slice is [a:b:c] when b and c should be the same since [a-b] sets the length and [a-c] sets the capacity. Now the length and capacity of slice2 is the same.

```
slice1 := []string{"A", "B", "C", "D", "E"}
slice2 := slice1[2:4:4]
slice2 = append(slice2, "CHANGED")
inspectSlice(slice1)
inspectSlice(slice2)

Output:
Length[5] Capacity[5]
[0] 0xc00007e000 A
[1] 0xc00007e010 B
[2] 0xc00007e020 C
[3] 0xc00007e030 D
[4] 0xc00007e040 E
Length[3] Capacity[4]
[0] 0xc000016080 C
[1] 0xc000016090 D
[2] 0xc0000160a0 CHANGED
```

Notice after the call to append, slice2 has a new backing array. This can be seen by comparing the addresses of each slice. In this case, the mutation against slice2 didn't cause a side effect against slice1.

3.18 Copying Slices Manually

There is a built-in function named copy that will allow for the shallow copying of slices. Since a string has a backing array of bytes, it can be used as a source but never a destination.

```
slice1 := []string{"A", "B", "C", "D", "E"}
slice3 := make([]string, len(slice1))
copy(slice3, slice1)
```

```
inspectSlice(slice1)
inspectSlice(slice3)

Output:
Length[5] Capacity[5]
[0] 0xc00005c050 A
[1] 0xc00005c060 B
[2] 0xc00005c070 C
[3] 0xc00005c080 D
[4] 0xc00005c090 E
Length[5] Capacity[5]
[0] 0xc00005c0a0 A
[1] 0xc00005c0b0 B
[2] 0xc00005c0c0 C
[3] 0xc00005c0d0 D
[4] 0xc00005c0e0 E
```

As long as the destination slice has the proper type and length, the built-in function copy can perform a shallow copy.

3.19 Slices Use Pointer Semantic Mutation

It’s important to remember that even though I use value semantics to move a slice around the program, when reading and writing with a slice, I am using pointer semantics. Sharing individual elements of a slice with different parts of my program can cause unwanted side effects.

```
// Construct a slice of 1 user, set a pointer to that user,
// use the pointer to update likes.
users := make([]user, 1)
ptrUsr0 := &users[0]
ptrUsr0.likes++
for i := range users {
    fmt.Printf("User: %d Likes: %d\n", i, users[i].likes)
}

Output:
User: 0 Likes: 1
```

A slice is used to maintain a collection of users. Then a pointer is set to the first user and used to update likes. The output shows that using the pointer is working.

```
// Append a new user to the collection. Use the pointer again
// to update likes.
users = append(users, user{})
ptrUsr0.likes++
for i := range users {
    fmt.Printf("User: %d Likes: %d\n", i, users[i].likes)
}

Output:
User: 0 Likes: 1
User: 1 Likes: 0
```

Then a new user is appended to the collection and the pointer is used again to add a like to the first user. However, since the append function replaced the backing array with a new one, the pointer is updating the old backing array and the likes are lost. The output shows the likes for the first user did not increase.

I have to be careful to know if a slice is going to be used in an append operation during the course of a running program. How I share the slice needs to be considered. Sharing individual indexes may not be the best idea. Sharing an entire slice value may not work either when appending is in operation. Probably making a slice a field in a struct, and sharing the struct value is a better way to go.

3.20 Linear Traversal Efficiency

The beauty of a slice is its ability to allow for performing linear traversals that are mechanically sympathetic while sharing data using value semantics to minimize heap allocations.

```
x := []byte{0x0A, 0x15, 0x0e, 0x28, 0x05, 0x96, 0x0b, 0xd0, 0x0}

a := x[0]
b := binary.LittleEndian.Uint16(x[1:3])
c := binary.LittleEndian.Uint16(x[3:5])
d := binary.LittleEndian.Uint32(x[5:9])

println(a, b, c, d)
```

The code is performing a linear traversal by creating slice values that read different sections of the byte array from beginning to end. All the data in this code stays on the stack. No extra copies of the data inside the byte slice are copied.

3.21 UTF-8

Go's compiler expects all code to be encoded in the UTF-8 character set. Make sure any file with source code is saved with this encoding or literal strings may be wrong when the program runs.

UTF-8 is a character set where I have bytes, code points, and then characters. One to four bytes of data can represent a code point (int32) and one to many code points can represent a character.

```
s := "世界 means world"
```

The string above represents 18 bytes, 14 code points, and 14 characters. Each chinese character I see requires 3 bytes to represent the code point/character I see.

```
var buf [utf8.UTFMax]byte
```

There is a utf8 package in the standard library that declares a constant named UTFMax. This constant represents the max number of bytes a code point could require, which is 4.

```
for i, r := range s {
```

When iterating over a string, the iteration moves code point by code point. Go has an alias type named rune (alias of int32) that represents a code point. Hence the use of the variable r as the value being copied.

On the first iteration, i will equal 0. On the next iteration, i will equal 3. Then in the next iteration, i will equal 6. All subsequent iterations will increment i by 1.

```
    rl := utf8.RuneLen(r)
```

The RuneLen function returns the number of bytes required to store the rune value. For the first two iterations, rl will equal 3.

```
    si := i + rl
    copy(buf[:], s[i:si])
```

The si variable represents the index position for the slice operation to slice the bytes associated with the rune. Then the built-in function copy is used to copy the bytes for the rune into the array. Notice how an array can be sliced. This proves that every array in Go is just a slice waiting to happen.

```
    fmt.Printf("%2d: %q; codepoint: %#6x; encoded bytes: %#v\n",
        i, r, r, buf[:rl])
}
```

The print statement displays each character, code point, and the set of bytes.

```
Output:
 0: '世'; codepoint: 0x4e16; encoded bytes: []byte{0xe4, 0xb8, 0x96}
 3: '界'; codepoint: 0x754c; encoded bytes: []byte{0xe7, 0x95, 0x8c}
 6: ' '; codepoint:   0x20; encoded bytes: []byte{0x20}
 7: 'm'; codepoint:   0x6d; encoded bytes: []byte{0x6d}
 8: 'e'; codepoint:   0x65; encoded bytes: []byte{0x65}
 9: 'a'; codepoint:   0x61; encoded bytes: []byte{0x61}
10: 'n'; codepoint:   0x6e; encoded bytes: []byte{0x6e}
11: 's'; codepoint:   0x73; encoded bytes: []byte{0x73}
12: ' '; codepoint:   0x20; encoded bytes: []byte{0x20}
13: 'w'; codepoint:   0x77; encoded bytes: []byte{0x77}
14: 'o'; codepoint:   0x6f; encoded bytes: []byte{0x6f}
15: 'r'; codepoint:   0x72; encoded bytes: []byte{0x72}
```

```
16: 'l'; codepoint:   0x6c; encoded bytes: []byte{0x6c}
17: 'd'; codepoint:   0x64; encoded bytes: []byte{0x64}
```

3.22 Declaring And Constructing Maps

A map is a data structure that provides support for storing and accessing data based on a key. It uses a hash map and bucket system that maintains a contiguous block of memory underneath.

```
type user struct {
    name     string
    username string
}

// Construct a map set to its zero value,
// that can store user values based on a key of type string.
// Trying to use this map will result in a runtime error (panic).
var users map[string]user

// Construct a map initialized using make,
// that can store user values based on a key of type string.
users := make(map[string]user)

// Construct a map initialized using empty literal construction,
// that can store user values based on a key of type string.
users := map[string]user{}
```

There are several ways to construct a map for use. A map set to its zero value is not usable and will result in my program panicking. The use of the built-in function make and literal construction does construct a map ready for use.

```
func main() {
    users := make(map[string]user)

    users["Roy"] = user{"Rob", "Roy"}
    users["Ford"] = user{"Henry", "Ford"}
    users["Mouse"] = user{"Mickey", "Mouse"}
    users["Jackson"] = user{"Michael", "Jackson"}

    for key, value := range users {
        fmt.Println(key, value)
    }
}

Output:
Roy {Rob Roy}
Ford {Henry Ford}
Mouse {Mickey Mouse}
Jackson {Michael Jackson}
```

If the built-in function make is used to construct a map, then the assignment operator can be used to add and update values in the map. The order of how keys/values are returned when ranging over a map is undefined by the spec and up to the compiler to implement.

```
func main() {
    users := map[string]user{
        "Roy":     {"Rob", "Roy"},
        "Ford":    {"Henry", "Ford"},
        "Mouse":   {"Mickey", "Mouse"},
        "Jackson": {"Michael", "Jackson"},
    }

    for key, value := range users {
        fmt.Println(key, value)
    }
}

Output:
Ford {Henry Ford}
Jackson {Michael Jackson}
Roy {Rob Roy}
Mouse {Mickey Mouse}
```

If the values for the map are known at construction, literal construction is a great choice.

3.23 Lookups and Deleting Map Keys

Once data is stored inside of a map, to extract any data a key lookup is required.

```
user1, exists1 := users2["Bill"]
user2, exists2 := users2["Ford"]

fmt.Println("Bill:", exists1, user1)
fmt.Println("Ford:", exists2, user2)

Output:
Bill: false { }
Ford: true {Henry Ford}
```

To perform a key lookup, square brackets are used with the map variable. Two values can be returned from a map lookup, the value and a boolean that represents if the value was found or not. If I don't need to know this, I can leave the "exists" variable out.

When a key is not found in the map, the operation returns a value of the map type set to its zero value state. I can see this with the "Bill" key lookup. Don't use zero value to determine if a key exists in the map or not.

```
delete(users, "Roy")
```

There is a built-in function named delete that allows for the deletion of data from the map based on a key.

3.24 Key Map Restrictions

Not all types can be used as a key.

```
type slice []user
Users := make(map[slice]user)

Compiler Error:
invalid map key type users
```

A slice is a good example of a type that can’t be used as a key. Only values that can be run through the hash function are eligible. A good way to recognize types that can be a key is if the type can be used in a comparison operation. I can’t compare two slice values.

Chapter 4 - Decoupling

4.1 Methods

A method provides data the ability to exhibit behavior.

A function is called a method when that function has a receiver declared. The receiver is the parameter that is declared between the keyword func and the function name. There are two types of receivers. There are value receivers for implementing value semantics and pointer receivers for implementing pointer semantics.

```
type user struct {
      name  string
      email string
}

func (u user) notify() {
      fmt.Printf("Sending User Email To %s<%s>\n", u.name, u.email)
}

func (u *user) changeEmail(email string) {
      u.email = email
      fmt.Printf("Changed User Email To %s\n", email)
}
```

The notify function is implemented with a value receiver. This means the method operates under value semantics and will operate on its own copy of the value used to make the call.

The changeEmail function is implemented with a pointer receiver. This means the method operates under pointer semantics and will operate on shared access to the value used to make the call.

Outside of a few exceptions, a method set for a type should not contain a mix of value and pointer receivers. Data semantic consistency is critically important and this includes declaring methods. I will talk about this more soon.

4.2 Method Calls

When making a method call, the compiler doesn’t care if the value used to make the call matches the receiver exactly. The compiler just wants a value or pointer of the same type.

```
bill := user{"Bill", "bill@email.com"}
bill.notify()
bill.changeEmail("bill@hotmail.com")
```

I can see that a value of type user is constructed and assigned to the bill variable. In the case of the notify call, the bill variable matches the receiver type which is using a value receiver. In the case of the changeEmail call, the bill variable doesn't match the receiver type which is using a pointer receiver. However, the compiler accepts the method call and shares the bill variable with the method. Go will adjust to make the call.

This works the same when the variable used to make the call is a pointer variable.

```
bill := &user{"Bill", "bill@email.com"}
bill.notify()
bill.changeEmail("bill@hotmail.com")
```

In this case the bill variable is a pointer variable to a value of type user. Once again, Go adjusts to make the method call when calling the notify method.

If Go didn't adjust, then this is what I would have to do to make those same method calls.

```
bill := user{"Bill", "bill@email.com"}
(&bill).changeEmail("bill@hotmail.com")

bill := &user{"Bill", "bill@email.com"}
(*bill).notify()
```

I'm glad I don't have to do that to make method calls in Go.

4.3 Data Semantic Guideline For Internal Types

As a guidelines, if the data I'm working with is an internal type (slice, map, channel, function, interface) then use value semantics to move the data around my program. This includes declaring fields on a type. However, when I'm reading and writing I need to remember I'm using pointer semantics.

```
type IP []byte
type IPMask []byte
```

These types are declared in the net package that is part of the standard library. They are declared with an underlying type which is a slice of bytes. Because of this, these types follow the guidelines for internal types.

```
func (ip IP) Mask(mask IPMask) IP {
    if len(mask) == IPv6len && len(ip) == IPv4len && allFF(mask[:12]) {
        mask = mask[12:]
    }
    if len(mask) == IPv4len && len(ip) == IPv6len &&
```

```
        bytesEqual(ip[:12], v4InV6Prefix) {
        ip = ip[12:]
    }
    n := len(ip)
    if n != len(mask) {
        return nil
    }
    out := make(IP, n)
    for i := 0; i < n; i++ {
        out[i] = ip[i] & mask[i]
    }
    return out
}
```

With the Mask method, value semantics are in play for both the receiver, parameter, and return argument. This method accepts its own copy of a Mask value, it mutates that value and then it returns a copy of the mutation. This method is using value semantic mutation. This is not an accident or random.

A function can decide what data input and output it needs. What it can't decide is how the data flows in or out. The data drives that decision and the function must comply. This is why Mask implements a value semantic mutation api. It must respect how a slice is designed to be moved around the program.

```
func ipEmptyString(ip IP) string {
    if len(ip) == 0 {
        return ""
    }
    return ip.String()
}
```

The ipEmptyString function is also using value semantics for the input and output. This function accepts its own copy of an IP value and returns a string value. No use of pointer semantics because the data dictates the data semantics and not the function.

One exception to using value semantics is when I need to share a slice or map with a function that performs unmarshaling or decoding.

4.4 Data Semantic Guideline For Struct Types

As a guideline, if the data I'm working with is a struct type then I have to think about what the data represents to make a decision. Though it would be great to choose value semantics for everything, when I'm not sure at all, it's best to start with pointer semantics. This is because not all data is safe to be copied.

After some time working with a new data type, the data semantic could become self-evident and refactoring the data semantic is what I want to do. Don't get hung up if it's not self-evident from the start.

```
type Time struct {
    sec  int64
    nsec int32
    loc  *Location
}
```

Here is the Time struct from the time package. If I was being asked to implement the api for this data structure, what should I choose, value or pointer semantics?

Sometimes I can ask these question:

Does a change in the data completely create a new data point?
Is the data specific to a context, and are mutations isolated to that context?
Should there only ever be one instance of this data?

If I'm looking at an existing code base and I want to know what data semantic was chosen, look for a factory function. The return type of a factory function should dictate the data semantic.

```
func Now() Time {
    sec, nsec := now()
    return Time{sec + unixToInternal, nsec, Local}
}
```

This is the factory function for constructing Time values. Look at the return, it's using value semantics. This tells me that I should be using value semantics for Time values which means every function gets its own copy of a Time value and fields in a struct should be declared as values of type Time.

```
func (t Time) Add(d Duration) Time {
    t.sec += int64(d / 1e9)
    nsec := int32(t.nsec) + int32(d%1e9)
    if nsec >= 1e9 {
        t.sec++
        nsec -= 1e9
    } else if nsec < 0 {
        t.sec--
        nsec += 1e9
    }
    t.nsec = nsec
    return t
}
```

Add is a method that needs to perform a mutation operation. Look closely and I will see that it's using value semantic mutation. The Add method gets its own copy of the Time value used to make the call, it mutates its own copy, then it returns a copy back to the caller. Once again, this is the safest way to perform a mutation operation.

```
func div(t Time, d Duration) (qmod2 int, r Duration) {}
```

Here is another example where the div function accepts a value of type Time and Duration (int64), returns values of type int and Duration. Value semantics for the Time type and for the built-in types.

```
func (t *Time) UnmarshalBinary(data []byte) error {}
func (t *Time) GobDecode(data []byte) error {}
func (t *Time) UnmarshalJSON(data []byte) error {}
func (t *Time) UnmarshalText(data []byte) error {}
```

These four methods from the Time package seem to break the rules for data semantic consistency. They are using pointer semantics, why? These methods require the use of pointer semantics because they are unmarshal and decoding methods implementing an interface.

Here is a guideline: If value semantics are at play, I can switch to pointer semantics for some functions as long as I don't let the data in the remaining call chain switch back to pointer semantics. Once I switch to pointer semantics, all future calls from that point need to stick to pointer semantics. I can never, ever, never, go from pointer to value. It's never safe to make a copy of a value that a pointer points to.

```
func Open(name string) (file *File, err error) {
    return OpenFile(name, O_RDONLY, 0)
}
```

The Open function from the os package shows that when using a value of type File, pointer semantics are at play. File values need to be shared and should never be copied.

```
func (f *File) Chdir() error {
    if f == nil {
        return ErrInvalid
    }
    if e := syscall.Fchdir(f.fd); e != nil {
        return &PathError{"chdir", f.name, e}
    }
    return nil
}
```

The method Chdir is using a pointer receiver even though this method does not mutate the File value. This is because File values need to be shared and can't be copied.

```
func epipecheck(file *File, e error) {
    if e == syscall.EPIPE {
        if atomic.AddInt32(&file.nepipe, 1) >= 10 {
            sigpipe()
```

```
        }
    } else {
        atomic.StoreInt32(&file.nepipe, 0)
    }
}
```

The epipecheck function as well accepts File values using pointer semantics.

4.5 Methods Are Just Functions

Methods are really just functions that provide syntactic sugar to provide the ability for data to exhibit behavior.

```
type data struct {
    name string
    age  int
}

func (d data) displayName() {
    fmt.Println("My Name Is", d.name)
}

func (d *data) setAge(age int) {
    d.age = age
    fmt.Println(d.name, "Is Age", d.age)
}
```

A type and two methods are declared. The displayName method is using value semantics and setAge is using pointer semantics.

Note: Do not implement setters and getters in Go. These are not apis that provide anything and in these cases it's better to make those fields exported.

```
d := data{
    name: "Hoanh",
}

d.displayName()
d.setAge(21)
```

A value of type data is constructed and method calls are made.

```
data.displayName(d)
(*data).setAge(&d, 21)
```

Since methods are really just functions with syntactic sugar, the methods can be executed like functions. I can see that the receiver is really a parameter, it's the first parameter. When I call a method, the compiler converts that to a function call underneath.

Note: Do not execute methods like this, but I may see this syntax in tooling messages.

4.6 Know The Behavior of the Code

If I know the data semantic at play, then I know the behavior of the code. If I know the behavior of the code, then I know the cost of the code. Once I know the cost, I'm engineering.

Given this type and method set.

```
type data struct {
    name string
    age  int
}

func (d data) displayName() {
    fmt.Println("My Name Is", d.name)
}

func (d *data) setAge(age int) {
    d.age = age
    fmt.Println(d.name, "Is Age", d.age)
}
```

I can write the following code.

```
d := data{
    name: "Bill",
}

f1 := d.displayName
f1()
d.name = "Joan"
f1()

Output:
My Name Is Bill
My Name Is Bill
```

I start with constructing a value of type Data assigning it to the variable d. Then I take the method displayName, bound to d, and assign that to a variable named f1. This is not a method call but an assignment which creates a level of indirection. Functions are values in Go and belong to the set of internal types.

After the assignment, I can call the method indirectly through the use of the variable as shown: f1(). This displays the name Bill. Then I change the data so the name is now Joan, and call the method once again through the f1 variable. I don't see the change. Bill is the output once again. So Why?

It has to do with the data semantic at play. The displayName method is using a value receiver so value semantics are at play.

```
func (d data) displayName() {
    fmt.Println("My Name Is", d.name)
}
```

This means that the f1 variable maintains and operates against its own copy of d. So calling the method through the f1 variable, will always use the copy and that copy is protected against change. This is what I want with value semantics.

```
d := data{
    name: "Bill",
}

f2 := d.setAge
f2(45)
d.name = "Sammy"
f2(45)

Output:
Bill Is Age 45
Sammy Is Age 45
```

This time the setAge method is assigned to the variable f2. Once again, the method is executed indirectly through the f2 variable passing 45 for Bill's age. Then Bill's name is changed to Sammy and the f2 variable is used again to make the call. This time I see the name has changed.

```
func (d *data) setAge(age int) {
    d.age = age
    fmt.Println(d.name, "Is Age", d.age)
}
```

The setAge function is using a pointer receiver so setAge doesn't operate on its own copy of the d variable but is operating directly on the d variable. Therefore, f2 is operating on shared access and I see the change.

Without knowing the data semantic at play, I won't know the behavior of the code. These data semantics are real and affect the behavior.

4.7 Escape Analysis Flaw

There is an escape analysis flaw at play here as well. When a piece of data is decoupled, the escape analysis algorithm can't properly track the data. This results in an automatic allocation of the data being decoupled. Regardless of the data semantic at play.

When a method is assigned to a variable, an internal type is constructed that maintains a pointer to the method and a pointer to the data needed to make the call. The data needed to make the call hits up against the double indirection flaw with escape analysis.

The general rule is, decoupling comes with the cost of indirection and allocation. In the case of value semantics, I am making a copy of the data so that is an allocation regardless. In the case of pointer semantics, the double indirection flaw causes the allocation. I will see this again with interfaces since the same mechanics are at play.

4.8 Interfaces

Interfaces give programs structure and encourage design by composition. They enable and enforce clean divisions between components. The standardization of interfaces can set clear and consistent expectations. Decoupling means reducing the dependencies between components and the types they use. This leads to correctness, quality and maintainability.

Interfaces allow me to group concrete data together by what the data can do. It’s about focusing on what data can do and not what the data is. Interfaces also help my code decouple itself from change by asking for concrete data based on what it can do. It’s not limited to one type of data.

I must do my best to understand what data changes are coming and use interfaces to decouple my program from that change. Interfaces should describe behavior and not state. They should be verbs and not nouns.

Generalized interfaces that focus on behavior are best. Interfaces with more than one method have more than one reason to change. Interfaces that are based on nouns, tend to be less reusable, are more susceptible to change, and defeat the purpose of the interface.

Uncertainty about change is not a license to guess but a directive to STOP and learn more. I must distinguish between code that defends against fraud vs protects against accidents.

Use an interface when:

- users of the API need to provide an implementation detail.
- API’s have multiple implementations they need to maintain internally.
- parts of the API that can change have been identified and require decoupling.

Don't use an interface:

- for the sake of using an interface.
- to generalize an algorithm.

- when users can declare their own interfaces.
- if it's not clear how the interface makes the code better.

4.9 Interfaces Are Valueless

The first important thing to understand is that interface types declare a valueless type.

```
type reader interface {
      read(b []byte) (int, error)
}
```

Type reader is not a struct type, but an interface type. It’s declaration is not based on state, but on behavior. Interface types declare a method-set of behavior that concrete data must exhibit in order to satisfy the interface. There is nothing concrete about interface types, therefore they are valueless.

```
var r reader
```

Because they are valueless, the construction of a variable (like r) is odd because in our programming model, r does not exist, it’s valueless. There is nothing about r itself that I can manipulate or transform. This is a critical concept to understand. I am never working with interface values but only concrete values. An interface has a compiler representation (internal type), but from our programming model, interfaces are valueless.

4.10 Implementing Interfaces

Go is a language that is about convention over configuration. When it comes to a concrete type implementing an interface, there is no exception.

```
type file struct {
    name string
}

func (file) read(b []byte) (int, error) {
    s := "<rss><channel><title>Going Go</title></channel></rss>"
    copy(b, s)
    return len(s), nil
}
```

The code declares a type named file and then declares a method named read. Because of these two declarations, I can say the following:

“The concrete type file now implements the reader interface using value semantics”

Every word I just said is important. In Go, all I have to do is declare the full method-set of behavior defined by an interface to implement that interface. In this

case, that is what I've done since the reader interface only declares a single act of behavior named read.

```
type pipe struct {
    name string
}

func (pipe) read(b []byte) (int, error) {
    s := `{name: "hoanh", title: "developer"}`
    copy(b, s)
    return len(s), nil
}
```

This code declares a type named pipe and then declares a method name read. Because of these two declarations, I can say the following:

"The concrete type pipe now implements the reader interface using value semantics"

Now I have two concrete types implementing the reader interface. Two concrete types each with their unique implementation. One type is reading file systems and the other networks.

4.11 Polymorphism

Polymorphism means that a piece of code changes its behavior depending on the concrete data it's operating on. This was said by Tom Kurtz, who is the inventor of BASIC. This is the definition we will use moving forward.

```
// retrieve can read any device and process the data.
func retrieve(r reader) error {
    data := make([]byte, 100)

    len, err := r.read(data)
    if err != nil {
        return err
    }

    fmt.Println(string(data[:len]))
    return nil
}
```

Take a look at the type of data this function accepts. It wants a value of type reader. That's impossible since reader is an interface and interfaces are valueless types. It can't be asking for a reader value, they don't exist.

If the function is not asking for a reader value then what is the function asking for? It is asking for the only thing it can ask for, concrete data.

The function retrieve is a polymorphic function because it's asking for concrete data not

based on what the data is (concrete type), but based on what the data can do (interface type).

```
f := file{"data.json"}
p := pipe{"cfg_service"}

retrieve(f)
retrieve(p)
```

I can construct two concrete values, one of type file and one of type pipe. Then I can pass a copy of each value to the polymorphic function. This is because each of these values implement the full method set of behavior defined by the reader interface.

When the concrete file value is passed into retrieve, the value is stored inside a two word internal type representing the interface value. The second word of the interface value points to the value being stored. In this case, it's a copy of the file value since value semantics are at play. The first word points to a special data structure that is called the iTable.

The iTable serves 2 purposes:

- It describes the type of value being stored. In my case, it's a file value.
- It provides us a matrix of function pointers so the concrete implementation of the method set for the type of value being stored can be executed.

When the read call is made against the interface value, an iTable lookup is performed to find the concrete implementation of the read method associated with the type. Then the method call is made against the value being stored in the second word.

I can say retrieve is a polymorphic function because the concrete value pipe can be passed into retrieve and now the call to read against the interface value changes its behavior. This time that call to read is reading a network instead of reading a file.

4.12 Interfaces via Pointer or Value Semantics

Implementing an interface using pointer semantics apply some constraints on interface compliance.

```
type notifier interface {
      notify()
}

type user struct {
      name  string
      email string
}
```

```
func (u *user) notify() {
      fmt.Printf("Sending User Email To %s<%s>\n", u.name, u.email)
}

func sendNotification(n notifier) {
      n.notify()
}

func main() {
    u := user{"Hoanh", "hoanhan@email.com"}
    sendNotification(u)
}

Compiler Error:
cannot use u (type user) as type notifier in argument to sendNotification:
user does not implement notifier (notify method has pointer receiver)
```

The notifier interface is implemented by the user type using pointer semantics. When value semantics are used to make the polymorphic call, the following compiler message is produced.

cannot use u (type user) as type notifier in argument to sendNotification:
user does not implement notifier (notify method has pointer receiver)

This is because there is a special set of rules in the specification about method sets. These rules define what methods are attached to value and pointers of a type. They are in place to maintain the highest level of integrity in my program.

4.13 Method Set Rules

These are the rules defined in the specification:

- For any value of type T, only those methods implemented with a value receiver for that type belong to the method set of that value.
- For any address of type T, all methods implemented for that type belong to the method set of that value.

In other words, when working with an address (pointer), all methods implemented are attached and available to be called. When working with a value, only those methods implemented with value receivers are attached and available to be called.

In the previous lesson about methods, I was able to call a method against a concrete piece of data regardless of the data semantics declared by the receiver. This is because the compiler can adjust to make the call. In this case, a value is being stored inside an interface and the methods must exist. No adjustments can be made.

The question now becomes: Why can't methods implemented with pointer receivers be attached to values of type T? What is the integrity issue here?

One reason is because I can't guarantee that every value of type T is addressable. If a value doesn't have an address, it can't be shared.

```
type duration int

func (d *duration) notify() {
         fmt.Println("Sending Notification in", *d)
}

func main() {
    duration(42).notify()
}

Compiler Error:
cannot call pointer method on duration(42)
cannot take the address of duration(42)
```

In this example, the value of 42 is a constant of kind int. Even though the value is converted into a value of type duration, it's not being stored inside a variable. This means the value is never on the stack or heap. There isn't an address. Constants only live at compile time.

The second reason is the bigger reason. The compiler is telling me that I am not allowed to use value semantics if I have chosen to use pointer semantics. In other words, I am being forced to share the value with the interface since it's not safe to make a copy of a value that a pointer points to. If I chose to implement the method with pointer semantics, I am indirectly saying that a value of this type isn't safe to copy.

```
func main() {
    u := user{"Hoanh", "hoanhan@email.com"}
    sendNotification(&u)
}
```

To fix the compiler message, I must use pointer semantics on the call to the polymorphic function and share u. The answer is not to change the method to use value semantics.

4.14 Slice of Interface

When I declare a slice of an interface type, I'm capable of grouping different concrete values together based on what they can do. This is why Go doesn't need the concept of sub-typing. It's not about a common DNA, it's about common behavior.

```
type printer interface {
    print()
}
```

```
type canon struct {
    name string
}

func (c canon) print() {
    fmt.Printf("Printer Name: %s\n", c.name)
}

type epson struct {
    name string
}

func (e *epson) print() {
    fmt.Printf("Printer Name: %s\n", e.name)
}

func main() {
    c := canon{"PIXMA TR4520"}
    e := epson{"WorkForce Pro WF-3720"}

    printers := []printer{
        c,
        &e,
    }

    c.name = "PROGRAF PRO-1000"
    e.name = "Home XP-4100"

    for _, p := range printers {
        p.print()
    }
}

Output:
Printer Name: PIXMA TR4520
Printer Name: Home XP-4100
```

The code shows how a slice of the interface type printer allows me to create a collection of different concrete printer types. Iterating over the collection and leveraging polymorphism since the call to p.print changes its behavior depending on the concrete value the code is operating against.

The example also shows how the choice of data semantics changes the behavior of the program. When storing the data using value semantics, the change to the original value is not seen. This is because a copy is stored inside the interface. When pointer semantics are used, any changes to the original value are seen.

4.15 Embedding

This first example does not show embedding, just the declaration of two struct types working together as a field from one type to the other.

```
type user struct {
```

```
    name  string
    email string
}

type admin struct {
    person user // NOT Embedding
    level  string
}
```

This is embedding.

```
type user struct {
    name  string
    email string
}

type admin struct {
    user   // Value Semantic Embedding
    level  string
}
```

The person field is removed and just the type name is left. I can also embed a type using pointer semantics.

```
type user struct {
    name  string
    email string
}

type admin struct {
    *user  // Pointer Semantic Embedding
    level  string
}
```

In this case a pointer of the type is embedded. In either case, accessing the embedded value is done through the use of the type's name.

The best way to think about embedding is to view user as an inner type and admin as an outer type. It's this inner/outer type relationship that is magical because with embedding, everything related to the inner type (both fields and methods) are promoted up to the outer type.

```
func (u *user) notify() {
    fmt.Printf("Sending user email To %s<%s>\n",
        u.name,
        u.email)
}

func main() {
    ad := admin{
        user: &user{
```

```
            name:  "john smith",
            email: "john@yahoo.com",
        },
        level: "super",
    }

    ad.user.notify()
    ad.notify() // Outer type promotion
}

Output:
Sending user email To john smith<john@yahoo.com>
Sending user email To john smith<john@yahoo.com>
```

I can see the output is the same whether I call the notify method through the inner pointer value directly or through the outer type value. The notify method declared for the user type is accessible directly by the admin type value.

Though this looks like inheritance and reusability, I must be careful. This is not about reusing state, but about promoting behavior.

```
type notifier interface {
    notify()
}

func sendNotification(n notifier) {
    n.notify()
}
```

Now I add an interface and a polymorphic function that accepts any concrete value that implements the full method set of behavior defined by notifier. Which is just a method named notify.

Because of embedding and promotion, values of type admin now implement the notifier interface.

```
func main() {
    ad := admin{
        user: &user{
            name:  "john smith",
            email: "john@yahoo.com",
        },
        level: "super",
    }

    sendNotification(&ad)
}

Output:
Sending user email To john smith<john@yahoo.com>
```

I can send the address of the admin value into the polymorphic function since

embedding promoted the notify behavior up to the admin type.

```
type admin struct {
    *user  // Pointer Semantic Embedding
    level  string
}

func (a *admin) notify() {
    fmt.Printf("Sending admin Email To %s<%s>\n",
        a.name,
        a.email)
}
```

When the outer type implements a method already implemented by the inner type, the promotion doesn't take place.

```
func main() {
    ad := admin{
        user: &user{
            name:  "john smith",
            email: "john@yahoo.com",
        },
        level: "super",
    }

    sendNotification(&ad)
}

Output:
Sending admin email To john smith<john@yahoo.com>
```

I can see the outer type's method is now being executed.

4.16 Exporting

Exporting provides the ability to declare if an identifier is accessible to code outside of the package it's declared in. A package is the basic unit of compiled code in Go. It represents a physical compiled unit of code, usually as a compiled library on the host operating system. Exporting determines access to identifiers across package boundaries.

```
package counters

type AlertCounter int
```

In this case, since a capital letter is being used to name the type AlterCounter, the type is exported and can be referenced directly by code outside of the counters package.

```
package counters
```

```
type alertCounter int
```

Now that I changed the type's name to start with a lowercase letter, the type is unexported. This means only code inside the counters package can reference this type directly.

```
package counters

type alertCounter int

func New(value int) alertCounter {
   return alertCounter(value)
}
```

Even though the code above is legal syntax and will compile, there is no value in it. Returning a value of an unexported type is confusing since the caller (who will probably exist in a different package) can't reference the type name directly.

```
package main

import (
   "fmt"

   "github.com/ardanlabs/.../exporting/example3/counters"
)

func main() {
    counter := counters.New(10)
    fmt.Printf("Counter: %d\n", counter)
}
```

In this case, the main function in package main calls the counters.New function successfully and the compiler can declare and construct a variable of the unexported type. This doesn't mean I should do this nor does it mean I'm getting any real protections for this. This should be avoided, and if New will return a value, it should be of an exported type.

```
package users

type User struct {
    Name string
    ID   int

    password string
}
```

When it comes to fields in a struct, the first letter declares if the field is accessible to code outside of the package it's declared in. In this case, Name and ID are accessible, but password is not. It's an idiom to separate exported and unexported fields in this

manner if this is reasonable or practical to do. Normally all fields would be one or the other.

```
package users

type user struct {
    Name string
    ID   int
}

type Manager struct {
   Title string

   user
}
```

In this scenario, even though the user type is unexported, it has two exported fields. This means that when the user type is embedded in the exported Manager type, the user fields promote and are accessible. It’s common to have types that are unexported with exported fields because the reflection package can only operate on exported fields. Marshallers won’t work otherwise.

Chapter 5 - Software Design

5.1 Grouping Different Types of Data

It’s important to remember that in Go the concepts of sub-typing or sub-classing really don't exist and these design patterns should be avoided.

The following is an anti-pattern I shouldn’t follow or implement.

```
type Animal struct {
    Name     string
    IsMammal bool
}
```

The Animal type is being declared as a base type that tries to define data that is common to all animals. I also attempt to provide some common behavior to an animal as well.

```
func (a *Animal) Speak() {
    fmt.Println("UGH!",
        "My name is", a.Name, ", it is", a.IsMammal, "I am a mammal")
}
```

Most animals have the ability to speak in one way or the other. However, trying to apply this common behavior to just an animal doesn’t make any sense. At this point, I have no idea what sound this animal makes, so I wrote UGH.

```
type Dog struct {
    Animal
    PackFactor int
}
```

Now the real problems begin. I’m attempting to use embedding to make a Dog everything an Animal is plus more. On the surface this will seem to work, but there will be problems. With that being said, a Dog does have a specific way they speak.

```
func (d *Dog) Speak() {
    fmt.Println("Woof!",
        "My name is", d.Name,
        ", it is", d.IsMammal,
        "I am a mammal with a pack factor of", d.PackFactor)
}
```

In the implementation of the Speak method, I can change out UGH for Woof. This is specific to how a dog speaks.

```
type Cat struct {
    Animal
    ClimbFactor int
}
```

If I’m going to have a Dog that represents an Animal, then I have to have a Cat. Using embedding, a Cat is everything an Animal is plus more.

```
func (c *Cat) Speak() {
    fmt.Println("Meow!",
        "My name is", c.Name,
        ", it is", c.IsMammal,
        "I am a mammal with a climb factor of", c.ClimbFactor)
}
```

In the implementation of the Speak method, I can change out UGH for Meow. This is specific to how a cat speaks.

Everything seems fine and it looks like embedding is providing the same functionality as inheritance does in other languages. Then I try to go ahead and group dogs and cats by the fact they have a common DNA of being an Animal.

```
// This does not compile, a Dog and Cat are not an Animal.

animals := []Animal{
    Dog{
        Animal: Animal{
            Name:     "Fido",
            IsMammal: true,
        },
        PackFactor: 5,
},
    Cat{
        Animal: Animal{
            Name:     "Milo",
            IsMammal: true,
        },
        ClimbFactor: 4,
    },
}

for _, animal := range animals {
    animal.Speak()
}
```

When I try to do this, the compiler complains that a Dog and Cat are not an Animal and this is true. Embedding isn’t the same as inheritance and this is the pattern I need to stay away from. A Dog is a Dog, a Cat a Cat, and an Animal an Animal. I can’t pass Dog’s and Cat’s around as if they are Animal’s because they are not.

This kind of mechanic is also not very flexible. It requires configuration by the

developer and this isn't very flexible unless I have access to the code and can make configuration changes over time.

If this is not how we can construct a collection of Dog's and Cat's, how can we do this in Go? It's not about grouping through common DNA, it's about grouping through common behavior. Behavior is the key.

```
type Speaker interface {
    Speak()
}
```

If I use an interface, then I can define the common method set of behavior that I want to group different types of data against.

```
speakers := []Speaker{
    &Dog{
        Animal: Animal{
            Name:     "Fido",
            IsMammal: true,
        },
        PackFactor: 5,
},
    &Cat{
        Animal: Animal{
            Name:     "Milo",
            IsMammal: true,
        },
        ClimbFactor: 4,
    },
}

for _, speaker := range speakers {
    speaker.Speak()
}
```

In the new code, I can now group Dogs and Cats together based on their common set of behavior, which is the fact that Dogs and Cats can speak.

In fact, the Animal type is really type pollution because declaring a type just to share a set of common state is a smell and should be avoided.

```
type Dog struct {
    Name     string
    IsMammal bool
    PackFactor int
}

type Cat struct {
    Name     string
    IsMammal bool
    ClimbFactor int
}
```

In this particular case, I would rather see the Animal type removed and the fields copied and pasted into the Dog and Cat types. Later I will have notes about better patterns that eliminate these scenarios from happening.

Here are the code smells from the original code:

- The Animal type provides an abstraction layer of reusable state.
- The program never needs to create or solely use a value of Animal type.
- The implementation of the Speak method for the Animal type is generalized.
- The Speak method for the Animal type is never going to be called.

Guidelines around declaring types:

- Declare types that represent something new or unique.
- Don't create aliases just for readability.
- Validate that a value of any type is created or used on its own.
- Embed types not because I need the state, but because we need the behavior.
- If I am not thinking about behavior, I'm locking myself into the design that I can't grow in the future without cascading code changes.
- Question types that are aliases or abstractions for an existing type.
- Question types whose sole purpose is to share common state.

5.2 Don't Design With Interfaces

Unfortunately, too many developers attempt to solve problems in the abstract first. They focus on interfaces right away and this leads to interface pollution. As a developer, I exist in one of two modes: a programmer and then an engineer.

When I am programming, I am focused on getting a piece of code to work. Trying to solve the problem and break down walls. Prove that my initial ideas work. That is all I care about. This programming must be done in the concrete and is never production ready.

Once I have a prototype of code that solves the problem, then I need to switch to engineering mode. I need to focus on how to write the code at a micro-level for my data semantics, macro-level for mental models, readability, and maintainability. I need to think about errors and fail states. So much more.

This work is done in a cycle of refactoring. Refactoring for readability, efficiency, abstraction, and for testability. Abstracting is one of a few refactors that need to be performed. This works best when I start with a piece of concrete code and then DISCOVER the interfaces that are needed. Don't apply abstractions unless they are absolutely necessary.

Every problem I solve with code is a data problem requiring me to write data

transformations. If I don’t understand the data, I don’t understand the problem. If I don’t understand the problem, I can’t write any code. Starting with a concrete solution that is based on the concrete data structures is critical. As Rob Pike said,

“Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident“. - Rob Pike

When is abstraction necessary? When I see a place in the code where the data could change and I want to minimize the cascading code effects that would result. I might use abstraction to help make code testable but I should try to avoid this if possible. The best testable functions are functions that take raw data in and send raw data out. It shouldn’t matter where the data is coming from or going.

In the end, start with a concrete solution to every problem. Even if the bulk of that is just programming. Then discover the interfaces that are absolutely required for the code today.

"Don’t design with interfaces, discover them". - Rob Pike

5.3 Composition

The best way to take advantage of embedding is through the compositional design pattern. The idea is to compose larger types from smaller types and focus on the composition of behavior.

```
type Xenia struct {
    Host    string
    Timeout time.Duration
}

func (*Xenia) Pull(d *Data) error {
    switch rand.Intn(10) {
    case 1, 9:
        return io.EOF
    case 5:
        return errors.New("Error reading data from Xenia")
    default:
        d.Line = "Data"
        fmt.Println("In:", d.Line)
        return nil
    }
}
```

The Xenia type represents a system that I need to pull data from. The implementation is not important. What is important is that the method Pull can succeed, fail, or not have any data to pull.

```
type Pillar struct {
    Host    string
```

```
    Timeout time.Duration
}

func (*Pillar) Store(d *Data) error {
    fmt.Println("Out:", d.Line)
    return nil
}
```

The Pillar type represents a system that I need to store data into. What is important again is that the method Store can succeed or fail.

These two types represent a primitive layer of code that provides the base behavior required to solve the business problem of pulling data out of Xenia and storing that data into Pillar.

```
func pull(x *Xenia, data []Data) (int, error) {
    for i := range data {
        if err := x.Pull(&data[i]); err != nil {
            return i, err
        }
    }

    return len(data), nil
}

func store(p *Pillar, data []Data) (int, error) {
    for i := range data {
        if err := p.Store(&data[i]); err != nil {
            return i, err
        }
    }

    return len(data), nil
}
```

The next layer of code is represented by these two functions, pull and store. They build on the primitive layer of code by accepting a collection of data values to set or store in the respective systems. These functions focus on the Xenia and Pillar since those are the systems the program needs to work with at this time.

```
func Copy(sys *System, batch int) error {
    data := make([]Data, batch)

    for {
        i, err := pull(&sys.Xenia, data)
        if i > 0 {
            if _, err := store(&sys.Pillar, data[:i]); err != nil {
                return err
            }
        }

        if err != nil {
            return err
```

```
        }
    }
}
```

The Copy function builds on top of the pull and store functions to move all the data that is pending for each run. If I notice the first parameter to Copy, it's a type called System.

```
type System struct {
    Xenia
    Pillar
}
```

The initial idea of the System type is to compose a system that knows how to pull and store. In this case, composing the ability to pull and store from Xenia and Pillar.

```
func main() {
    sys := System{
        Xenia: Xenia{
            Host:    "localhost:8000",
            Timeout: time.Second,
        },
        Pillar: Pillar{
            Host:    "localhost:9000",
            Timeout: time.Second,
        },
    }

    if err := Copy(&sys, 3); err != io.EOF {
        fmt.Println(err)
    }
}
```

Finally, the main function can be written to construct a Xenia and Pillar within the composition of a System. Then the System can be passed to the Copy function and data can begin to flow between the two systems.

With all this code, I now have my **first draft** of a concrete solution to a concrete problem.

5.4 Decoupling With Interfaces

The next step is to understand what could change in the program. In this case, what can change is the systems themselves. Today it's Xenia and Pillar, tomorrow it could be Alice and Bob. With this knowledge, I want to decouple the existing concrete solution from this change. To do that, I want to change the concrete functions to be polymorphic functions.

```
func pull(p Puller, data []Data) (int, error) {
```

```
    for i := range data {
        if err := p.Pull(&data[i]); err != nil {
            return i, err
        }
    }

    return len(data), nil
}

func store(s Storer, data []Data) (int, error) {
    for i := range data {
        if err := s.Store(&data[i]); err != nil {
            return i, err
        }
    }

    return len(data), nil
}
```

In the first implementation, pull accepted a Xenia and store accepted a Pillar. In the end it wasn't Xenia and Pillar that was important, what's important is a concrete value that knows how to pull and store. I can change these concrete functions to be polymorphic by asking for data based on what it can do instead of what it is.

```
type Puller interface {
    Pull(d *Data) error
}

type Storer interface {
    Store(d *Data) error
}
```

These two interfaces describe what concrete data must do and it's these types that are replaced in the declaration of the pull and store functions. Now these functions are polymorphic. When Alice and Bob are declared and implemented as a Puller and a Storer, they can be passed into the functions.

I am not done yet, the Copy function needs to be polymorphic as well.

```
func Copy(ps PullStorer, batch int) error {
    data := make([]Data, batch)

    for {
        i, err := pull(ps, data)
        if i > 0 {
            if _, err := store(ps, data[:i]); err != nil {
                return err
            }
        }

        if err != nil {
            return err
        }
```

```
    }
}
```

The Copy function is no longer asking for a System, but any concrete value that knows how to pull and store.

```
type PullStorer interface {
    Puller
    Storer
}
```

The PullStorer interface is declared through the use of composition. It's composed of the Puller and Storer interfaces. Work towards composing larger interfaces from smaller ones.

Notice how the PullStorer variable is now being passed into the pull and store functions. How is this possible when the type information is different?

```
func pull(p Puller, data []Data) (int, error) {
func store(s Storer, data []Data) (int, error) {

i, err := pull(ps, data)
if _, err := store(ps, data[:i]); err != nil {
```

I always need to remember, I am never passing an interface value around my program since they don't exist and are valueless. I can only pass concrete data. So the concrete data stored inside of the interface ps variable is what's being passed to pull and store. Isn't it true, the concrete value stored inside of ps must know how to pull and store. This is what is happening.

Since a System is composed from a Xenia and Pillar, System implements the PullStorer interface. With these changes, I can now create new concrete types that implement the PullStorer interface.

```
type System1 struct {
    Xenia
    Pillar
}

type System2 struct {
    Alice
    Bob
}

type System3 struct {
    Xenia
    Bob
}
```

```
type System4 struct {
    Alice
    Pillar
}
```

When I think about this more, declaring different System types for all the possible combinations is not realistic. This will work, but the maintenance nightmare requires a better solution.

5.5 Interface Composition

What if I decided to compose my concrete system type from two interface types?

```
type System struct {
    Puller
    Storer
}
```

This is an interesting solution. This would allow the application to inject the concrete puller or storer into the system at application startup. This one system type implements the PullStorer interface for all possible combinations of concrete types.

```
func main() {
    sys := System{
        Puller: &Xenia{
            Host:    "localhost:8000",
            Timeout: time.Second,
        },
        Storer: &Pillar{
            Host:    "localhost:9000",
            Timeout: time.Second,
        },
    }

    if err := Copy(&sys, 3); err != io.EOF {
        fmt.Println(err)
    }
}
```

With this change, the application is fully decoupled from changes to a new system that may come online over time.

5.6 Readability Review

The next question to ask is, are the polymorphic functions as precise as they otherwise could be? This is a part of the engineering process that can’t be skipped. The answer is no, two changes can be made.

```
func Copy(sys *System, batch int) error {
```

The Copy function doesn’t need to be polymorphic anymore since there will only be a single System type. The PullStorer interface can be removed from the program. Remember, I moved the polymorphism inside the type when I used composition with the interface types.

```
func Copy(p Puller, s Storer, batch int) error {
```

This is another change that can be made to the Copy function. This change makes the function more precise and polymorphic again. Now the function is asking for exactly what it needs based on what the concrete data can do.

This idea of precision comes from Edsger W. Dijkstra

"The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise. - Edsger W. Dijkstra

5.7 Implicit Interface Conversions

As I saw in the last example, An interface value of one type can be passed for a different interface type if the concrete value stored inside the interface complies with both behaviors. This could be considered an implicit interface conversion, but it’s better to think how concrete data is being moved between interfaces in a decoupled state.

```
type Mover interface {
    Move()
}

type Locker interface {
    Lock()
    Unlock()
}

type MoveLocker interface {
    Mover
    Locker
}
```

Given these three interfaces, where the MoveLocker is the composition of Mover and Locker.

```
type bike struct{}

func (bike) Move() {
    fmt.Println("Moving the bike")
}

func (bike) Lock() {
    fmt.Println("Locking the bike")
```

```
}

func (bike) Unlock() {
    fmt.Println("Unlocking the bike")
}
```

And given this concrete type bike that implements all three interfaces. What can I do?

```
var ml MoveLocker
var m Mover
```

I can construct a value of type MoveLocker and Mover to its zero value state. These are interface values that are truly valueless.

```
ml = bike{}
```

Then I can construct a value of type bike to its zero value state and assign a copy to the MoveLocker variable ml. This is possible because a bike implements all three behaviors, and the compiler can see that the implementation exists.

```
m = ml
```

I can then assign the MoveLocker variable ml to the Mover variable m. This is possible because I'm not assigning the interface value ml but the concrete value stored inside of ml which is a bike value. The compiler knows that any concrete value stored inside of ml must also implement the Mover interface.

This assignment however is not valid.

```
ml = m

cannot use m (type Mover) as type MoveLocker in assignment:
     Mover does not implement MoveLocker (missing Lock method)
```

I can't assign the Mover variable m back to the MoverLocker variable ml because the compiler can only guarantee that the concrete value stored inside of m knows how to Move. It doesn't know at compile time if the concrete value also knows how to Lock and UnLock.

5.8 Type assertions

With all that being said, there is a way at runtime to test if the assignment is legal and then make it happen. That is by using a type assertion.

```
b := m.(bike)
```

```
ml = b
```

A type assertion allows me at runtime to ask a question, is there a value of the given type stored inside the interface. I see that with the m.(bike) syntax. In this case, I am asking if there is a bike value stored inside of m at the moment the code is executed. If there is, then the variable b is given a copy of the bike value stored. Then the copy can be copied inside of the ml interface variable.

If there isn't a bike value stored inside of the interface value, then the program panics. I want this if there absolutely should have been a bike value stored. What if there is a chance there isn't and that is valid? Then I need the second form of the type assertion.

```
b, ok := m.(bike)
```

In this form, if ok is true, there is a bike value stored inside of the interface. If ok is false, then there isn't and the program does not panic. The variable b however is still of type bike, but it is set to its zero value state.

I can experiment with this more.

```
func main() {
    rand.Seed(time.Now().UnixNano())

    mvs := []fmt.Stringer{
        car{},
        cloud{},
    }

    for i := 0; i < 10; i++ {
        rn := rand.Intn(2)

        if v, is := mvs[rn].(cloud); is {
            fmt.Println("Got Lucky:", v)
            continue
        }

        fmt.Println("Got Unlucky")
    }
}
```

Assuming the program does declare two types named car and cloud that each implement the fmt.Stringer interface, I can construct a collection that allows me to store a value of both car and cloud. Then 10 times, I randomly choose a number from 0 to 1, and perform a type assertion to see if that random index contains a cloud value. Since it is possible the index may not, the second form of the type assertion is critical here.

5.9 Interface Pollution

I can spot interface pollution from a mile away. It mostly comes from the fact that people are designing software with interfaces instead of discovering them. I should design a concrete solution to the problem first. Then I can discover where the program needs to be polymorphic, if at all.

These are things I've heard from other developers.

"I'm using interfaces because we have to use interfaces".

No. We don't have to use interfaces. We use interfaces when it's practical and reasonable to do so. There is a cost of using interfaces: a level of indirection and allocation when we store concrete values inside of them. Unless the cost of the allocation is worth whatever decoupling I'm getting, I shouldn't be using interfaces.

"I need to be able to test my code so I need to use interfaces".

No. I must design my API for the user first, not my test. If the API is not testable, I should question if it's usable. There are different layers of API's as well. The lower level unexported API's can and should focus on testability. The higher level exported API's need to focus on usability.

Functions that accept raw data in and return raw data out are the most testable. Separate the data transformation from where the data comes from and where it is going. This is a refactoring exercise I need to perform during the engineering coding cycle.

Below is an example that creates interface pollution by improperly using an interface when one is not needed.

```
type Server interface {
    Start() error
    Stop() error
    Wait() error
}
```

The Server interface defines a contract for TCP servers. The problem here is I don't need a contract, I need an implementation. There will only be one implementation as well, especially since I am the one implementing it. I do not need someone else to implement this for me.

Plus, this interface is based on a noun and not a verb. Concrete types are nouns since they represent the concrete problem. Interfaces describe the behavior and Server is

not behavior.

Here are some ways to identify interface pollution:

- A package declares an interface that matches the entire API of its own concrete type.
- The interfaces are exported but the concrete types implementing the interface are unexported.
- The factory function for the concrete type returns the interface value with the unexported concrete type value inside.
- The interface can be removed and nothing changes for the user of the API.
- The interface is not decoupling the API from change.

Guidelines around interface pollution:

Use an interface:

- When users of the API need to provide an implementation detail.
- When APIs have multiple implementations that need to be maintained.
- When parts of the APIs that can change have been identified and require decoupling.

Question an interface:

- When its only purpose is for writing testable API's (write usable APIs first).
- When it's not providing support for the API to decouple from change.
- When it's not clear how the interface makes the code better.

5.10 Interface Ownership

One thing that is different about Go from other languages is the idea of convention over configuration. This really shows itself with how Go handles interface compliance. Because the compiler can perform static code analysis to determine if a concrete value implements an interface, the developer declaring the concrete type doesn't need to provide interfaces as well.

```
package pubsub

type PubSub struct {
    host string
}

func New(host string) *PubSub {
    return &PubSub{
        host: host,
    }
}
```

```
func (ps *PubSub) Publish(key string, v interface{}) error {
    // PRETEND THERE IS A SPECIFIC IMPLEMENTATION.
    return nil
}

func (ps *PubSub) Subscribe(key string) error {
    // PRETEND THERE IS A SPECIFIC IMPLEMENTATION.
    return nil
}
```

I've just implemented a new API that provides a concrete implementation for publish and subscribe. There are no interfaces being provided because this API does not need one. This is a single concrete implementation.

What if the application developer wanting to use this new API needs an interface because they have the need to mock this implementation during tests? In Go, that developer can declare the interface and the compiler can identify the compliance.

```
package main

type publisher interface {
    Publish(key string, v interface{}) error
    Subscribe(key string) error
}

type mock struct{}

func (m *mock) Publish(key string, v interface{}) error {
    // ADD MY MOCK FOR THE PUBLISH CALL.
    return nil
}

func (m *mock) Subscribe(key string) error {
    // ADD MY MOCK FOR THE SUBSCRIBE CALL.
    return nil
}
```

This code in the main package is declaring an interface. This interface represents the API that the application is using from the pubsub package. Then the developer implements their own pubsub implementation for testing. The key here is that this application developer doesn't use any concrete implementation directly, but decouples themselves through their own interface.

```
func main() {
    pubs := []publisher{
        pubsub.New("localhost"),
        &mock{},
    }

    for _, p := range pubs {
        p.Publish("key", "value")
```

```
        p.Subscribe("key")
    }
}
```

To provide an example, the main function constructs a collection that is initialized with the pubsub implementation and the mock implementation. The publisher interface allows this. Then a for range loop is implemented to show how the application code is abstracted from any concrete implementation.

5.11 Error Handling

Integrity matters and it's a big part of the engineering process. At the heart of integrity is error handling. When it comes to Go, error handling is not an exception to be handled later or somewhere else in the code. It's a part of the main path and needs to be a main focus.

Developers have the responsibility to return enough context about any error so a user can make an informed decision about how to proceed. Handling an error is about three things: logging the error, determining if the goroutine/program needs to shutdown or can continue, not propagating the error any further.

In Go, errors are just values so they can be anything I need them to be. They can maintain any state or behavior.

```
// http://golang.org/pkg/builtin/#error
type error interface {
    Error() string
}
```

This is the error interface and it's an interface that is built into the language. This is why it appears to be an unexported identifier. Any concrete value that implements this interface can be used as an error value.

One important aspect of Go is that error handling is done in a decoupled state through this interface. A key reason for this is because error handling is an aspect of my application that is more susceptible to change and improvement. This is the type Go applications must use as the return type for error handling.

```
// http://golang.org/src/pkg/errors/errors.go
type errorString struct {
    s string
}

// http://golang.org/src/pkg/errors/errors.go
func (e *errorString) Error() string {
    return e.s
}
```

This is the most commonly used error value in Go programs. It's declared in the errors package from the standard library. Notice how the type is unexported and it has one unexported field which is a string. I can also see how pointer semantics are used to implement the error interface. This means only addresses to values of this type can be stored inside the interface. The method just returned the error string.

It's important to remember, the implementation of the Error method serves the purpose of implementing the interface and for logging. If any user needs to parse the string returned from this method, I have failed providing the user the right amount of context to make an informed decision.

```
// http://golang.org/src/pkg/errors/errors.go
func New(text string) error {
    return &errorString{text}
}
```

The New function is how an error using the concrete type errorString is constructed. Notice how the function returns the error using the error interface. Also notice how pointer semantics are being used.

```
func main() {
    if err := webCall(); err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println("Life is good")
}

func webCall() error {
    return New("bad request")
}
```

Context is everything with errors. Each error must provide enough context to allow the caller to make an informed decision about the state of the goroutine/application. In this example, the webCall function returns an error with the message Bad Request. In the main function, a call is made to webCall and then a check is made to see if an error has occurred with the call.

```
if err := webCall(); err != nil {
    fmt.Println(err)
    return
}
```

The key to the check is err != nil. What this condition is asking is, is there a concrete value stored inside the err interface value. When the interface value is storing a concrete value, there is an error. In this case, the context is literally just the fact that a

concrete value exists, it’s not important what the concrete value is.

What if it’s important to know what error value exists inside the err interface variable? Then error variables are a good option.

```
var (
    ErrBadRequest = errors.New("Bad Request")
    ErrPageMoved = errors.New("Page Moved")
)
```

Error variables provide a mechanic to identify what specific error is being returned. They have an idiom of starting with the prefix Err and are based on the concrete type errorString from the errors package.

```
func webCall(b bool) error {
    if b {
        return ErrBadRequest
    }
    return ErrPageMoved
}
```

In this new version of webCall, the function returns one or the other error variable. This allows the caller to determine which error took place.

```
func main() {
    if err := webCall(true); err != nil {
        switch err {
        case ErrBadRequest:
            fmt.Println("Bad Request Occurred")
            return

        case ErrPageMoved:
            fmt.Println("The Page moved")
            return

        default:
            fmt.Println(err)
            return
        }
    }

    fmt.Println("Life is good")
}
```

In the application after the call to webCall is made, a check can be performed to see if there is a concrete value stored inside the err interface variable. If there is, then a switch statement is used to determine which error it was by comparing err to the different error variables.

In this case, the context of the error is based on which error variable was returned.

What if an error variable is not enough context? What if some special state needs to be checked, like with networking errors? In these cases, a custom concrete error type is the answer.

```
type UnmarshalTypeError struct {
    Value string
    Type  reflect.Type
}

func (e *UnmarshalTypeError) Error() string {
    return "json: cannot unmarshal " + e.Value +
           " into Go value of type " + e.Type.String()
}
```

This is a custom concrete error type implemented in the json package. Notice the name has a suffix of Error in the naming of the type. Also notice the use of pointer semantics for the implementation of the error interface. Once again the implementation is for logging and should display information about all the fields being captured.

```
type InvalidUnmarshalError struct {
    Type reflect.Type
}

func (e *InvalidUnmarshalError) Error() string {
    if e.Type == nil {
        return "json: Unmarshal(nil)"
    }
    if e.Type.Kind() != reflect.Ptr {
        return "json: Unmarshal(non-pointer " + e.Type.String() + ")"
    }
    return "json: Unmarshal(nil " + e.Type.String() + ")"
}
```

This is a second custom concrete error type found in the json package. The implementation of the Error method is a bit more complex, but once again just for logging and using pointer semantics.

```
func Unmarshal(data []byte, v interface{}) error {
    rv := reflect.ValueOf(v)
    if rv.Kind() != reflect.Ptr || rv.IsNil() {
        return &InvalidUnmarshalError{reflect.TypeOf(v)}
    }
    return &UnmarshalTypeError{"string", reflect.TypeOf(v)}
}
```

Here is a portion of the Unmarshal function. Notice how it constructs the concrete error values in the return, passing them back to the caller through the error interface. Pointer semantic construction is being used because pointer semantics were used in the declaration of the Error method.

The context of the error here is more about the type of error stored inside the error interface. There needs to be a way to determine that.

```
func main() {
    var u user
    err := Unmarshal([]byte(`{"name":"bill"}`), u)
    if err != nil {
        switch e := err.(type) {
        case *UnmarshalTypeError:
            fmt.Printf("UnmarshalTypeError: Value[%s] Type[%v]\n",
                e.Value, e.Type)
        case *InvalidUnmarshalError:
            fmt.Printf("InvalidUnmarshalError: Type[%v]\n", e.Type)
        default:
            fmt.Println(err)
        }
        return
    }
    fmt.Println("Name:", u.Name)
}
```

A generic type assertion within the scope of the switch statement is how I can write code to test what type of value is being stored inside the err interface value. Type is the context here and now I can test and take action with access to all the state of the error.

However, this poses one problem. I'm no longer decoupled from the concrete error value. This means if the concrete error value is changed, my code can break. The beautiful part of using an interface for error handling is being decoupled from breaking changes.

If the concrete error value has a method set, then I can use an interface for the type check. As an example, the net package has many concrete error types that implement different methods. One common method is called Temporary. This method allows the user to test if the networking error is critical or just something that can recover on its own.

```
type temporary interface {
    Temporary() bool
}

func (c *client) BehaviorAsContext() {
    for {
        line, err := c.reader.ReadString('\n')
        if err != nil {
            switch e := err.(type) {
            case temporary:
                if !e.Temporary() {
                    log.Println("Temporary: Client leaving chat")
                    return
                }
            default:
```

```
                if err == io.EOF {
                    log.Println("EOF: Client leaving chat")
                    return
                }
                log.Println("read-routine", err)
            }
        }
        fmt.Println(line)
    }
}
```

In this code, the call to ReadString could fail with an error from the net package. In this case, an interface is declared that represents the common behavior a given concrete error value could implement. Then with a generic type assertion, I test if that behavior exists and I can call into it. The best part, I stay in a decoupled state with my error handling.

5.12 Always Use The Error Interface

One mistake Go developers can make is when they use the concrete error type and not the error interface for the return type for handling errors. If I were to do this, bad things can happen.

```
type customError struct{}

func (c *customError) Error() string {
    return "Find the bug."
}

func fail() ([]byte, *customError) {
    return nil, nil
}

func main() {
    var err error
    if _, err = fail(); err != nil {
        log.Fatal("Why did this fail?")
    }
    log.Println("No Error")
}

Output:
Why did this fail?
```

Why does this code think there is an error when the fail function returns nil for the error? It’s because the fail function is using the concrete error type and not the error interface. In this case, there is a nil pointer of type customError stored inside the err variable. That is not the same as a nil interface value of type error.

5.13 Handling Errors

Handling errors is more of a macro level engineering conversation. In my world, error handling means the error stops with the function handling the error, the error is logged with full context, and the error is checked for its severity. Based on the severity and ability to recover, a decision to recover, move on, or shutdown is made.

One problem is that not all functions can handle an error. One reason could be because not all functions are allowed to log. What happens when an error is being passed back up the call stack and can't be handled by the function receiving it? An error needs to be wrapped in context so the function that eventually handles it, can properly do so.

There are two options for wrapping extra context around an error. I can use Dave Cheney's errors package or I can use standard library support that can be found in the errors and fmt packages. Whatever I decide, it's important to annotate errors for enough context to help identify and fix problems. Both at runtime and after.

Using Dave Cheney's package:

```
package main

import (
    "fmt"

    "github.com/pkg/errors"
)

type AppError struct {
    State int
}

func (c *AppError) Error() string {
    return fmt.Sprintf("App Error, State: %d", c.State)
}

func main() {
    if err := firstCall(10); err != nil {
        switch v := errors.Cause(err).(type) {
        case *AppError:
            fmt.Println("Custom App Error:", v.State)
        default:
            fmt.Println("Default Error")
        }

        fmt.Printf("%v\n", err)
    }
}

func firstCall(i int) error {
    if err := secondCall(i); err != nil {
        return errors.Wrapf(err, "secondCall(%d)", i)
    }
```

```
    return nil
}

func secondCall(i int) error {
    return &AppError{99}
}

Output:
Custom App Error: 99
secondCall(10): App Error, State: 99
```

What's nice about this package is the errors.Wrap and errors.Cause API's. They make the code a bit more readable.

Using the standard library:

```
package main

import (
    "fmt"

    "github.com/pkg/errors"
)

type AppError struct {
    State int
}

func (c *AppError) Error() string {
    return fmt.Sprintf("App Error, State: %d", c.State)
}

func Cause(err error) error {
    root := err
    for {
        if err = errors.Unwrap(root); err == nil {
            return root
        }
        root = err
    }
}

func main() {
      if err := firstCall(10); err != nil {
        var ap *AppError
        if errors.As(err, &ap) {
            fmt.Println("As says it is an AppError")
        }

        switch v := Cause(err).(type) {
        case *AppError:
            fmt.Println("Custom App Error:", v.State)
        default:
            fmt.Println("Default Error")
        }

        fmt.Printf("%v\n", err)
```

```
    }
}

func firstCall(i int) error {
    if err := secondCall(i); err != nil {
        return fmt.Errorf("secondCall(%d) : %w", i, err)
    }
    return nil
}

func secondCall(i int) error {
    return &AppError{99}
}

Output:
As says it is an AppError
Custom App Error: 99
secondCall(10): App Error, State: 99
```

To use the standard library in a similar way, the Cause function needed to be written. In this example, I can see the use of the errors.As function.

Chapter 6 - Concurrency

6.1 Scheduler Semantics

When a Go program starts up, the Go runtime asks the machine (virtual or physical) how many operating systems threads can run in parallel. This is based on the number of cores that are available. For each thread that can be run in parallel, the runtime creates an operating system thread (M) and attaches that to a data structure that represents a logical processor (P) inside the program. This P and M represent the compute power or execution context for running the Go program.

Also, an initial goroutine (G) is created to manage the execution of instructions on a selected M/P. Just like a M manages the execution of instructions on the hardware, a G manages the execution of instructions on the M. This creates a new layer of abstraction above the operating system, but it moves execution control to the application level.

Since the Go scheduler sits on top of the operating system scheduler, it's important to have some semantic understanding of the operating system scheduler and the constraints it applies to the Go scheduler and applications.

The operating system scheduler has the job of creating the illusions that multiple pieces of work are being executed at the same time. Even when this is physically impossible. This requires some tradeoffs in the design of the scheduler. Before I go any farther, it's important to define some words.

Work: Instructions to be executed for a running application. This is accomplished by threads and an application can have 1 to many threads.

Thread: A path of execution that is scheduled and performed. Threads are responsible for the execution of instructions on the hardware.

Thread States: A thread can be in 1 of three states: Running, Runnable, or Waiting. Running means the thread is executing its assigned instructions on the hardware. Runnable means the thread wants time on the hardware to execute its assigned instructions. Waiting means the thread is waiting for something before it can resume its work. Waiting threads are not a concern of the scheduler.

Concurrency: This means undefined out of order execution. In other words, given a set of instructions that would be executed in the order provided, they are executed in a different undefined order, but all executed. The key is, the result of executing the full set of instructions in any undefined order produces the same result. I will say work can be done concurrently when the order the work is executed in doesn't matter, as long as all the work is completed.

Parallelism: This means doing a lot of things at once. For this to be an option, I need the ability to physically execute 2 or more operating system threads at the same time on the hardware.

CPU Bound Work: This is work that does not cause the thread to naturally move into a waiting state. Calculating fibonacci numbers would be considered CPU bound work.

I/O Bound Work: This is work that does cause the thread to naturally move into a waiting state. Fetching data from different URLs would be considered I/O bound work.

Synchronization: When two or more Goroutines will need to access the same memory location potentially at the same time, they need to be synchronized to take turns. If this synchronization doesn't take place, and at least 1 Goroutine is performing a write, I can end up with a data race. Data races are a cause of data corruption bugs that can be difficult to find.

Orchestration: When two or more Goroutines need to signal each other, with or without data, orchestration is the mechanic required. If orchestration does not take place, guarantees about concurrent work being performed and completed will be missed. This can cause all sorts of data corruption bugs.

These three blog posts go into deeper details about schedule semantics for both the operating system and the Go scheduler. This material is presented in class.

https://www.ardanlabs.com/blog/2018/08/scheduling-in-go-part1.html
https://www.ardanlabs.com/blog/2018/08/scheduling-in-go-part2.html
https://www.ardanlabs.com/blog/2018/12/scheduling-in-go-part3.html

Some changes to the Go scheduler have taken place since these were written. The Go scheduler since Go 1.15 has moved from a cooperating scheduler to a preemptive scheduler using many of the same operating system techniques. This means the scheduler can context switch a Goroutine at a finer level. It doesn't need to wait for the Goroutine to enter a function call.

6.2 Concurrency Basics

Starting with a basic concurrency problem that requires orchestration.

```
func init() {
    runtime.GOMAXPROCS(1)
}
```

The call to GOMAXPROCS is being used to run the Go program as a single threaded Go program. This program will be single threaded and have a single P/M to execute all

Goroutines. The function is capitalized because it’ s also an environment variable. Though this function call will overwrite the variable.

```
g := runtime.GOMAXPROCS(0)
```

This function is an important function when I set CPU quotas to a container configuration. When passing 0, the number of threads the Go program will be using is reported. I must make sure that number matches the number of operating system threads I have available in my containerized environment. If the numbers are not the same, the Go program won’t run as well as it otherwise could. I might want to use the environment variable or this call to match things up.

```
func main() {
    var wg sync.WaitGroup
    wg.Add(2)

    go func() {
        lowercase()
        wg.Done()
    }()

    go func() {
        uppercase()
        wg.Done()
    }()

    fmt.Println("Waiting To Finish")
    wg.Wait()

    fmt.Println("\nTerminating Program")
}
```

This program has an orchestration problem. The main Goroutine can’t allow the main function to return until there is a guarantee the 2 Goroutines being created finish their work first. A WaitGroup is a perfect tool for orchestration problems that don’t require data to be passed between Goroutines. The signaling here is performed through an API that allows a Goroutine to wait for other Goroutines to signal they’re done.

In this code, a WaitGroup is constructed to its zero value state and then immediately the Add method is called to set the WaitGroup to 2, which will match the number of Goroutines to be created. When I know how many Goroutines upfront that will be created, I should call Add once with that number. When I don’t know (like in a streaming service) then calling Add(1) is acceptable.

At the end of main is the call to Wait. Wait holds the main Goroutine from causing the function to return. When the main function returns, the Go program is shut down with extreme prejudice. This is why managing the orchestration with the proper guarantees is important. The Wait call will block until the WaitGroup is set back to 0.

In the middle of the program, I have the creation of the two Goroutines.

```
go func() {
    lowercase()
    wg.Done()
}()

go func() {
    uppercase()
    wg.Done()
}()
```

Literal functions are declared and executed with the use of the keyword Go. At this point, I am telling the Go scheduler to execute these functions concurrently. To execute them in an undefined order. Inside the implementation of each Goroutine is the call to Done. That call is what decrements the WaitGroup by 1. Once both calls to Done are made, the WaitGroup will change from 2 to 0, and then the main Goroutine will be allowed to be unblocked from the call to Wait. Terminating the program.

```
func main() {
    var wg sync.WaitGroup
    wg.Add(2)

    go func() {
        lowercase()
        wg.Done()
    }()

    . . .
}
```

An important part of this orchestration pattern is keeping the Add and Done calls in the same line of sight. Try not to pass the WaitGroup as a function parameter where the calls get lost. This will help to reduce bugs.

```
Output:

Start Goroutines
Waiting To Finish
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z A B C D E F G H I J K L M N O
P Q R S T U V W X Y Z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d
e f g h i j k l m n o p q r s t u v w x y z a b c d e f g h i j k l m n o p q r s
t u v w x y z a b c d e f g h i j k l m n o p q r s t u v w x y z
Terminating Program
```

When I build and run this program, I see how this program did run concurrently. The second Goroutine created was scheduled first. It got to finish its work and then the other Goroutine ran. Both ran to completion before the program terminated. The next time I run this program, there is no guarantee I see the same output. The only

guarantee in this program is that the program won't terminate until the two Goroutines are done.

Even if I run this program 100 times and see the same output, there is no guarantee it will happen again. It may be highly probable, but not guaranteed. Especially not guaranteed across different versions, operating systems and architectures.

What happens if I remove the Wait guarantee from the program?

```
func main() {
    . . .

    fmt.Println("Waiting To Finish")
    // wg.Wait()

    fmt.Println("\nTerminating Program")
}
```

If I comment the call to Wait what will happen when I run this program? Once again, there is no guarantee at all anymore with what will happen, but there are different possibilities.

The program could behave as before since calls to Println are system calls that do allow the scheduler to make a context switch. The program could execute just one of the two Goroutines or possibly terminate immediately.

```
func main() {
    var wg sync.WaitGroup
    wg.Add(2)

    go func() {
        lowercase()
        // wg.Done()
    }()

    . . .
}
```

What happens if I forget to call Done in one of the Goroutines? In this case, the program would deadlock since the WaitGroup can't get back down to 0. The Wait call will block forever.

```
Output:

Start Goroutines
Waiting To Finish
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z A B C D E F G H I J K L M N O
P Q R S T U V W X Y Z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d
e f g h i j k l m n o p q r s t u v w x y z a b c d e f g h i j k l m n o p q r s
```

```
t u v w x y z a b c d e f g h i j k l m n o p q r s t u v w x y z fatal error:
all goroutines are asleep - deadlock!

goroutine 1 [semacquire]:
sync.runtime_Semacquire(0xc00001a0a8)
      /usr/local/go/src/runtime/sema.go:56 +0x45
sync.(*WaitGroup).Wait(0xc00001a0a0)
      /usr/local/go/src/sync/waitgroup.go:130 +0x65
main.main()

/Users/bill/code/go/src/github.com/ardanlabs/gotraining/topics/go/concurrency/gor
outines/example1/example1.go:42 +0x145
exit status 2
```

I can see how the Go Runtime identified the program is deadlocked on line 42 where the call to Wait can be found. I shouldn't get too excited about deadlock detection since every single Goroutine needs to be blocked with no way out. This shows why keeping the Add and Done call together is so important.

```
func main() {
    var wg sync.WaitGroup
    wg.Add(1)

    go func() {
        lowercase()
        wg.Done()
    }()

    go func() {
        uppercase()
        wg.Done()
    }()

    . . .
}
```

What happens if I don't give the WaitGroup the correct number of Goroutines to wait on? If the number is too large, I will have another deadlock. If the number is too small, there are no guarantees that the work is done before the program moves on. The output of the program is undefined.

6.3 Preemptive Scheduler

Even though the scheduler runs within the scope of the application, it's important to see how the schedule is preemptive. This means I can't predict when a context switch will take place and this will change every time I run the program.

```
func main() {
    var wg sync.WaitGroup
    wg.Add(2)

    go func() {
```

```
        printHashes("A")
        wg.Done()
    }()

    go func() {
        printHashes("B")
        wg.Done()
    }()

    fmt.Println("Waiting To Finish")
    wg.Wait()

    fmt.Println("\nTerminating Program")
}
```

Using the same orchestration pattern as before, this program has each Goroutine doing a lot more work. Work that the scheduler won't give any given Goroutine enough time to finish completely in one time slice.

```
func printHashes(prefix string) {
    for i := 1; i <= 50000; i++ {
        num := strconv.Itoa(i)
        sum := sha1.Sum([]byte(num))
        fmt.Printf("%s: %05d: %x\n", prefix, i, sum)
    }
    fmt.Println("Completed", prefix)
}
```

This function is performing a lot of I/O bound work that has the potential of being context switched.

```
$ ./example2 | cut -c1 | grep '[AB]' | uniq
B
A
B
A
B
A
B
A  7 Context Switches


$ ./example2 | cut -c1 | grep '[AB]' | uniq
B
A
B
A
B
A
B
A
B
A
B
A  11 Context Switches
```

```
$ ./example2 | cut -c1 | grep '[AB]' | uniq
B
A
B
A
B
A  5 Context Switches
```

As I can see, everytime I run the program, there are a different number of context switches. This is a great thing because a scheduler shouldn't be predictable. Concurrency needs to remain undefined and I must remember that when I use concurrency to solve my performance problems.

```
func init() {
    runtime.GOMAXPROCS(2)
}
```

What happens if I go back to the original program but change GOMAXPROCS so the program runs as a two threaded Go program?

```
Output:

Start Goroutines
Waiting To Finish
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z A B C D E F G H I J K L M N a
b c d e f g h i j k l m n o O P Q R S T U V W X Y Z A B C D E F G H I J K L M N O
P Q R S T U V W X Y Z p q r s t u v w x y z a b c d e f g h i j k l m n o p q r s
t u v w x y z a b c d e f g h i j k l m n o p q r s t u v w x y z
Terminating Program
```

What I see is that the concurrency of the program is now more fine grained. The output to the letter is undefined and out of order.

6.4 Data Races

A data race is when two or more Goroutines are trying to access the same memory location at the same time where at least one Goroutine is performing a write. When this happens it is impossible to predict the result. These types of bugs are difficult to find because they cause issues that always appear random.

These ~8 minutes from Scott Myers is great to listen to here:

CPU Caches and Why You Care 30:09-38:30
https://youtu.be/WDIkqP4JbkE?t=1809

6.5 Data Race Example

This is a great example of a data race and how they can be hidden for years and eventually show up at odd times and cause data corruption.

```
var counter int

func main() {
    const grs = 2

    var wg sync.WaitGroup
    wg.Add(grs)

    for g := 0; g < grs; g++ {
        go func() {
            for i := 0; i < 2; i++ {
                value := counter
                value++
                counter = value
            }
            wg.Done()
        }()
    }

    wg.Wait()
    fmt.Println("Counter:", counter)
}
```

This program creates two goroutines that each access the same integer variable, incrementing the variable twice. The goroutine performs a read, modify, and write operation against the shared state manually.

```
go func() {
    for i := 0; i < 2; i++ {
        value := counter
        value++
        counter = value
    }
    wg.Done()
}()
```

I can see the access to the shared state inside the for loop. When I build and run this program I get the right answer of 4 each and every time.

```
$ ./example1
Final Counter: 4

$ ./example1
Final Counter: 4

$ ./example1
Final Counter: 4
```

How is this working?

```
G1                                  Shared State: 0
G2
-------------------------------------------------------------------------------
----
Read:   0
Modify: 1
                                     G1 Write: 1
Context Switch
                                                                        Read:   1
                                                                        Modify: 2
                                     G2 Write: 2
                                                                  Context Switch
Read:   2
Modify: 3
                                     G1 Write: 3
Terminate
                                                                        Read:   3
                                                                        Modify: 4
                                     G2 Write: 4
                                                                        Terminate
-------------------------------------------------------------------------------
----
                                   Shared State: 4
```

The read, modify and write operations are happening atomically. Just because I am getting the right answer doesn't mean there isn't a problem. What happens if I add a log statement in the middle of the read, modify, and write operation?

```
go func() {
    for i := 0; i < 2; i++ {
        value := counter
        value++

        log.Println("logging")

        counter = value
    }
    wg.Done()
}()
```

If I run this program I no longer get the same result of 4, I now get the answer of 2.

```
$ ./example1
Final Counter: 2

$ ./example1
Final Counter: 2

$ ./example1
```

```
Final Counter: 2
```

What is happening? I am running into a data race bug that did exist before but wasn't hitting. The call to log is now causing the scheduler to make a context switch between the two goroutines. The context switch causes the state of the change for one goroutine to be dirty.

```
G1                                  Shared State: 0
G2
--------------------------------------------------------------------------------
----
Read:   0
Modify: 1
Context Switch
                                                                        Read:   0
                                                                        Modify: 1
                                                                    Context Switch
                                     G1 Write: 1
Read:   1
Modify: 2
Context Switch
                                     G2 Write: 1
                                                                        Read:   1
                                                                        Modify: 2
                                                                    Context Switch
                                     G1 Write: 2
Terminate
                                     G1 Write: 2
                                                                        Terminate
--------------------------------------------------------------------------------
----
                                    Shared State: 2
```

I am very lucky this is happening every time and I can see it. But normally a data race like this happens "randomly" and is impossible to know about until it's too late. Luckily Go has a race detector to help find data races.

6.6 Race Detection

There are several ways to engage the race detector. I can use it with the run, build and test command. If I use it with the build command, I have to remember to run the program. They say an instrumented binary can slow my program down by 20%.

```
$ go build -race
$ ./example1
```

The -race flag is how to instrument the build with the race detector. I will probably use it more with "go test", but for this example I am instrumenting the binary and then running it.

```
2021/02/01 17:30:52 logging
2021/02/01 17:30:52 logging
2021/02/01 17:30:52 logging
==================
WARNING: DATA RACE
Write at 0x000001278d88 by goroutine 8:
  main.main.func1()
      /data_race/example1/example1.go:41 +0xa6

Previous read at 0x000001278d88 by goroutine 7:
  main.main.func1()
      /data_race/example1/example1.go:38 +0x4a

Goroutine 8 (running) created at:
  main.main()
      /data_race/example1/example1.go:36 +0xaf

Goroutine 7 (finished) created at:
  main.main()
      /data_race/example1/example1.go:36 +0xaf
==================
2021/02/01 17:30:52 logging
Final Counter: 2
Found 1 data race(s)
```

I can see a race was detected when running the program. This would happen with or without the log statement inserted. When a race is detected the program panics and provides this trace. The trace has two parts to it, it shows where there was unsynchronized access to the same shared state where at least one access was a write.

In this trace, a goroutine performed a write at address 0x000001278d88 on line 41, and there was an unsynchronized read at the same address by another goroutine on line 38. Both goroutines were created on line 36.

```
36 go func() {
37     for i := 0; i < 2; i++ {
38         value := counter
39         value++
40         log.Println("logging")
41         counter = value
42     }
43     wg.Done()
44 }()
```

I can clearly see the unsynchronized read and write. As a side note, the plus plus operation on line 39 would also be a data race if the code was accessing the counter variable. The plus plus operation is a read, modify, and write operation underneath and the operating system could easily context switch in the middle of that.

So how can I fix the code to make sure that I remove the data race? There are two tools I can use, atomic instructions and mutexes.

6.7 Atomics

Atomics provide synchronization at the hardware level. Because of this, it's limited to words and half-words of data. So they're great for counters or fast switching mechanics. The WaitGroup API's use atomics.

What changes do I need to make to apply atomics to the code?

```
var counter int32

func main() {
    const grs = 2

    var wg sync.WaitGroup
    wg.Add(grs)

    for g := 0; g < grs; g++ {
        go func() {
            for i := 0; i < 2; i++ {
                atomic.AddInt32(&counter, 1)
            }
            wg.Done()
        }()
    }

    wg.Wait()
    fmt.Println("Counter:", counter)
}
```

I only need to do a couple things. First, change the counter variable to be a precision based integer. I can see that at the top of the code listing. The atomic functions only work with precision based integers. Second, remove the manually read, modify, and write code for one call to atomic.AddInt32. That one call handles it all.

All of the functions associated with the atomic package take the address to the shared state to be synchronized. Synchronization only happens at the address level. So different goroutines calling the same function, but at a different address, won't be synchronized.

The core API for atomics looks like this:

```
func AddInt32(addr *int32, delta int32) (new int32)
func AddInt64(addr *int64, delta int64) (new int64)
func AddUint32(addr *uint32, delta uint32) (new uint32)
func AddUint64(addr *uint64, delta uint64) (new uint64)
func AddUintptr(addr *uintptr, delta uintptr) (new uintptr)

func CompareAndSwapInt32(addr *int32, old, new int32) (swapped bool)
func CompareAndSwapInt64(addr *int64, old, new int64) (swapped bool)
func CompareAndSwapPointer(addr *unsafe.Pointer, old, new unsafe.Pointer)
(swapped bool)
```

```
func CompareAndSwapUint32(addr *uint32, old, new uint32) (swapped bool)
func CompareAndSwapUint64(addr *uint64, old, new uint64) (swapped bool)
func CompareAndSwapUintptr(addr *uintptr, old, new uintptr) (swapped bool)

func LoadInt32(addr *int32) (val int32)
func LoadInt64(addr *int64) (val int64)
func LoadPointer(addr *unsafe.Pointer) (val unsafe.Pointer)
func LoadUint32(addr *uint32) (val uint32)
func LoadUint64(addr *uint64) (val uint64)
func LoadUintptr(addr *uintptr) (val uintptr)

func StoreInt32(addr *int32, val int32)
func StoreInt64(addr *int64, val int64)
func StorePointer(addr *unsafe.Pointer, val unsafe.Pointer)
func StoreUint32(addr *uint32, val uint32)
func StoreUint64(addr *uint64, val uint64)
func StoreUintptr(addr *uintptr, val uintptr)

func SwapInt32(addr *int32, new int32) (old int32)
func SwapInt64(addr *int64, new int64) (old int64)
func SwapPointer(addr *unsafe.Pointer, new unsafe.Pointer) (old unsafe.Pointer)
func SwapUint32(addr *uint32, new uint32) (old uint32)
func SwapUint64(addr *uint64, new uint64) (old uint64)
func SwapUintptr(addr *uintptr, new uintptr) (old uintptr)

type Value
    func (v *Value) Load() (x interface{})
    func (v *Value) Store(x interface{})
```

I can see that the first parameter is always the address to a precision based integer or pointer. There is also a type named Value that provides a synchronous value with a small API.

6.8 Mutexes

What if I wanted to keep the three lines of code I had. Then atomics aren't going to work. What I need then is a mutex. A mutex let's me box a group of code so only one goroutine at a time can execute that code.

```
var counter int

func main() {
    const grs = 2

    var wg sync.WaitGroup
    wg.Add(grs)

    var mu sync.Mutex

    for g := 0; g < grs; g++ {
        go func() {
            for i := 0; i < 2; i++ {
                mu.Lock()
                {
                    value := counter
                    value++
```

```
                    counter = value
                }
                mu.Unlock()
            }
            wg.Done()
        }()
    }

    wg.Wait()
    fmt.Println("Counter:", counter)
}
```

There are several changes to this code from the original. I added the construction of the mu variable to be a mutex set to its zero value. Then inside the for loop, I added calls to Lock and Unlock with an artificial code block. Inside the code block I have the code that needs to be synchronized. The code block is for readability.

With this code in place, the scheduler will only allow one goroutine to enter the code block at a time. It's important to understand that a mutex is not a queue. The first goroutine that calls Lock isn't necessarily the first goroutine who gets the Lock. There is a fair based algorithm but this is done on purpose so people don't use mutexes as queues.

It's important to remember the Lock creates back pressure, so the longer it takes to get from the Lock to the Unlock, the more chance of goroutines waiting for their turn. If I forget to call Unlock, then all goroutines waiting will deadlock. This is why it's critical that the call to Lock and Unlock happen in the same function. Make sure I'm doing the bare minimum I need in the code block, but at least the minimum.

This is very bad code where someone is trying to get in and out of the Lock so quickly they actually lose the synchronization and the race detector can't even discover the problem.

```
var counter int

func main() {
    const grs = 2

    var wg sync.WaitGroup
    wg.Add(grs)

    var mu sync.Mutex

    for g := 0; g < grs; g++ {
        go func() {
            for i := 0; i < 2; i++ {
                var value int
                mu.Lock()
                {
                    value = counter
                }
```

```
                mu.Unlock()

                value++

                mu.Lock()
                {
                    counter = value
                }
                mu.Unlock()
            }
            wg.Done()
        }()
    }

    wg.Wait()
    fmt.Println("Counter:", counter)
}
```

As a general guideline, if I see a call to Lock from the same mutex twice in the same function, stop the code review. There is probably a mistake or over complication. In this case the calls to read and write are being synchronized, however, two goroutines can end up at the value++ line of code with the same value. The data race still exists and the race detector is helpless in finding it.

6.9 Read/Write Mutexes

There is a second type of mutex called a read/write mutex. It allows me to separate the locks around reads and writes. This is important since reading data doesn't pose a threat unless a goroutine is attempting to write at the same time. So this type of mutex allows multiple goroutines to read at the same time. As soon as a write lock is requested, the reads are no longer issued, the write takes place, the reads can start again.

```
package main

import (
    "fmt"
    "math/rand"
    "sync"
    "time"
)

var data []string
var rwMutex sync.RWMutex

func init() {
    rand.Seed(time.Now().UnixNano())
}

func main() {
    var wg sync.WaitGroup
    wg.Add(1)

    go func() {
```

```
        for i := 0; i < 10; i++ {
            writer(i)
        }
        wg.Done()
    }()

    for i := 0; i < 8; i++ {
        go func(id int) {
            for {
                reader(id)
            }
        }(i)
    }

    wg.Wait()
    fmt.Println("Program Complete")
}

func writer(i int) {
    rwMutex.Lock()
    {
        time.Sleep(time.Duration(rand.Intn(100)) * time.Millisecond)
        fmt.Println("****> : Performing Write")
        data = append(data, fmt.Sprintf("String: %d", i))
    }
    rwMutex.Unlock()
}

func reader(id int) {
    rwMutex.RLock()
    {
        time.Sleep(time.Duration(rand.Intn(10)) * time.Millisecond)
        fmt.Printf("%d : Performing Read : Length[%d]\n", id, len(data))
    }
    rwMutex.RUnlock()
}
```

I can see the use of a read/write mutex where there are 8 goroutines reading the length of a slice within a 10 millisecond delay of each other, and 1 goroutine waking up within 100 milliseconds to append a value (write) to the slice.

The key is the implementation of the writer and reader functions. Notice how I use Lock for the writer and RLock for the reader. One of the biggest mistakes I can make with this is mixing up the Unlock calls with the wrong version. Having a Lock with a RUnlock will never end well.

```
7 : Performing Read : Length[0]
5 : Performing Read : Length[0]
0 : Performing Read : Length[0]
3 : Performing Read : Length[0]
7 : Performing Read : Length[0]
2 : Performing Read : Length[0]
1 : Performing Read : Length[0]
****> : Performing Write
0 : Performing Read : Length[1]
```

```
5 : Performing Read : Length[1]
3 : Performing Read : Length[1]
6 : Performing Read : Length[1]
7 : Performing Read : Length[1]
4 : Performing Read : Length[1]
1 : Performing Read : Length[1]
2 : Performing Read : Length[1]
****> : Performing Write
7 : Performing Read : Length[2]
1 : Performing Read : Length[2]
3 : Performing Read : Length[2]
```

The output shows how multiple goroutines are reading at the same time, but all the reading stops when the write takes place.

6.10 Channel Semantics

It's important to think of a channel not as a data structure, but as a mechanic for signalling. This goes in line with the idea that I send and receive from a channel, not read and write. If the problem in front of me can't be solved with signalling, if the word signalling is not coming out of my mouth, I need to question the use of channels.

There are three things that I need to focus on when thinking about signalling. The first one is, does the goroutine that is sending the signal, need a guarantee that the signal has been received? I might think that the answer to this question is always yes, but remember, there is a cost to every decision and there is a cost to having a guarantee at the signaling level.

The cost of having the guarantee at the signaling level is unknown latency. The sender won't know how long they need to wait for the receiver to take the signal. Having to wait for the receiver creates blocking latency. In this case, unknown amounts of blocking latency.

The mechanics behind this working is that the receive happens before the send. Nanoseconds before, but still before. The receiver takes the signal and then walks away, this action by the receiver allows the sender to move on.

What if the process can't wait for an unknown amount of time? What if that kind of latency won't work? Then the guarantee can't be at the signaling level, it needs to be outside of it. The mechanics behind this working is that the send happens before the receive. The sender can perform the signal without needing the receiver to be available. So the sender gets to walk away and not wait. Eventually, I hope, the receiver shows up and takes the signal.

This is reducing latency cost on the send, but it is creating uncertainty about signals being received and therefore knowing if there are problems upstream with receivers.

This can create the process to accept work that never gets started or finished. It could eventually cause massive back pressure and systems to crash.

The second thing to focus on is, do I need to send data with the signal? If the signal requires the transmission of data, then the signalling is a 1 to 1 between goroutines. If a second goroutine needs to receive a signal, a second send must be performed.

If data doesn't need to be transmitted with the signal, then the signal can be a 1 to 1 between goroutines or 1 to many. Signalling without data is primarily used for cancellation or shutdowns. It's done by turning off the lights.

The third thing to focus on is channel state. A channel can be in 1 of 3 states.

A channel can be in a nil state by constructing the channel to its zero value state. Sends and receives against channels in this state will block. This is good for situations where I want to implement short term stoppages of work.

A channel can be in an open state by using the built-in function make. Sends and receives against channels in this state will work under the following conditions:

Unbuffered Channels:
Guarantees at the signaling level with the receive happening before send. Sending and receiving goroutine need to come together in the same space and time for a signal to be processed.

Buffered Channels:
Guarantees outside of the signaling level with the send happening before the receive. If the buffer is not full, sends can complete or they block. If the buffer is not empty, receives can complete or they block.

A channel can be in a closed state by using the built-in function close. I don't need to close a channel to release memory, this is for changing the state. Sending on a closed channel will cause a panic. Receiving on a closed channel will return immediately, there is a way to identify if the receive occurred because there was data or because it was closed.

With all this information, I can focus on channel patterns. The focus on signaling is important. The idea if I need a guarantee at the signaling level or not, based on latency concerns. If I need to transmit data with the signal or not, based on handling cancellations or not. I want to convert the syntax to these semantics.

6.11 Channel Patterns

There are 7 channel patterns that are important to understand since they provide the building blocks to signalling.

6.11.1 Wait For Result

The wait for result pattern is a foundational pattern used by larger patterns like fan out/in. In this pattern, a goroutine is created to perform some known work and signals their result back to the goroutine that created them. This allows for the actual work to be placed on a goroutine that can be terminated or walked away from.

```
func waitForResult() {
    ch := make(chan string)

    go func() {
        time.Sleep(time.Duration(rand.Intn(500)) * time.Millisecond)
        ch <- "paper"
        fmt.Println("employee : sent signal")
    }()

    p := <-ch
    fmt.Println("manager : recv'd signal :", p)

    time.Sleep(time.Second)
    fmt.Println("--------------------------------------------------")
}
```

At the beginning of the function is the use of the builtin function make. In this case, an unbuffered channel is being constructed to its open state. It’s better to look at this as a channel is being constructed to signal string data with guarantees at the signaling level. Which means the sending goroutine wants a guarantee that the signal being sent has been received.

Once the channel is constructed, a new goroutine is created to perform work and the existing goroutine waits to receive a signal with string data. Because there are guarantees at the signaling level, the amount of time the existing goroutine will need to wait is unknown. It’s the unknown latency cost of this type of channel.

The new goroutine goes ahead and begins to perform its work immediately. To simulate the unknown latency problem, a sleep with a random number of milliseconds is employed to define the work. Once the work is done, the new goroutine performs a send with string data. The existing goroutine is already blocked waiting in a receive.

Since the receive happens nanoseconds before the send, which creates the guarantee, I would think the print call for the receive signal would always appear before the print for the send. But there is no guarantee in what order I will see the print calls execute. I need to remember, both goroutines are running on their own operating system thread in parallel, the receive is only happening nanoseconds before, after the channel operation, all things are equal again.

6.11.2 Fan Out/In

The fan out/in pattern uses the wait for result pattern just described.

```
func fanOut() {
    emps := 2000
    ch := make(chan string, emps)

    for e := 0; e < emps; e++ {
        go func(emp int) {
            time.Sleep(time.Duration(rand.Intn(200)) * time.Millisecond)
            ch <- "paper"
            fmt.Println("employee : sent signal :", emp)
        }(e)
    }

    for emps > 0 {
        p := <-ch
        emps--
        fmt.Println(p)
        fmt.Println("manager : recv'd signal :", emps)
    }

    time.Sleep(time.Second)
    fmt.Println("-------------------------------------------------")
}
```

The idea of this pattern is to create a goroutine for each individual piece of work that is pending and can be done concurrently. In this code sample, I am going to create 2000 goroutines to perform 2000 individual pieces of work. I am going to use a buffered channel since there is only one receiver and it’s not important to have a guarantee at the signaling level. That will only create extra latency.

Instead, the idea is to move the guarantee to know when all the signals have been received. This will reduce the cost of latency from the channels. That will be done with a counter that is decremented for each received signal until it reaches zero.

A buffered channel of 2000 is constructed, one for each goroutine being created. Then in a loop, 2000 goroutines are created and they are off to do their work. A random sleep is used to simulate the work and the unknown amount of time it takes to get the work done. The key is that the order of the work is undefined, out of order, execution which also changes each time the program runs. If this is not acceptable, I can’t use concurrency.

Once all the goroutines are created, the initial goroutine waits in a receive loop. Eventually as data is signaled into the buffered channel, the receiving goroutine will pick up the data and eventually all the work is received.

I must remember, a fan out is dangerous in a running service since the number of

goroutines I create for the fan are a multiplier. If I have a service handling 50k requests on 50 thousand goroutines, and I decide to use a fan out pattern of 10 goroutines for some of the requests, in a worse case scenario I would be talking 500k goroutines existing at the same time. Depending on the resources those goroutines needed, I might not have them available at that scale and the back pressure could bring the service down.

6.11.3 Wait For Task

The wait for task pattern is a foundational pattern used by larger patterns like pooling.

```
func waitForTask() {
    ch := make(chan string)

    go func() {
        p := <-ch
        fmt.Println("employee : recv'd signal :", p)
    }()

    time.Sleep(time.Duration(rand.Intn(500)) * time.Millisecond)
    ch <- "paper"
    fmt.Println("manager : sent signal")

    time.Sleep(time.Second)
    fmt.Println("-------------------------------------------------")
}
```

At the beginning the function creates an unbuffered channel so there is a guarantee at the signalling level. This is critically important for pooling so I can add mechanics to allow me to walk away on timeouts and cancellation. Once the channel is created, a goroutine is created immediately waiting for a signal with data to perform work. The function begins to prepare that work and finally signals the work to the waiting goroutine. Since the guarantee is at the signaling level, the waiting goroutine doesn’t know how long it needs to wait.

6.11.4 Pooling

The pooling pattern uses the wait for task pattern just described. The pooling pattern allows me to manage resource usage across a well defined number of goroutines. In Go, pooling is not needed for efficiency in CPU processing like at the operating system. It’s more important for efficiency in resource usage.

```
func pooling() {
    ch := make(chan string)

    g := runtime.GOMAXPROCS(0)
    for e := 0; e < g; e++ {
        go func(emp int) {
            for p := range ch {
                fmt.Printf("employee %d : recv'd signal : %s\n", emp, p)
            }
```

```
            fmt.Printf("employee %d : recv'd shutdown signal\n", emp)
        }(e)
    }

    const work = 100
    for w := 0; w < work; w++ {
        ch <- "paper"
        fmt.Println("manager : sent signal :", w)
    }

    close(ch)
    fmt.Println("manager : sent shutdown signal")

    time.Sleep(time.Second)
    fmt.Println("-------------------------------------------------")
}
```

In this pattern a group of goroutines are created to service the same channel. There is efficiency in this because the size of the pool dictates the amount of concurrent work happening at the same time. If I have a pool of 16 goroutines, that could represent 16 files being opened at any given time. Represents the amount of memory needed for 16 goroutines.

The code starts with the creation of an unbuffered channel. It's critically important that an unbuffered channel is used because without the guarantee at the signalling level, I can't walk away from the pool if it's busy for a long time. The next part of the code decides the number of goroutines the pool will contain.

```
g := runtime.GOMAXPROCS(0)
```

The call to runtime.GOMAXPROCS is important in that it queries the runtime (when passing 0 as a parameter) to the number of threads that exist for running goroutines. The number should always equal the number of cores/hardware_threads that are available to the program. It represents the amount of CPU capacity available to the program. When the size of the pool isn't obvious, start with this number as a baseline. It won't be uncommon for this number to provide a reasonable performance benchmark.

The for loop creates the pool of goroutines where each goroutine sits in a blocking receive call using the for/range mechanics for a channel.

```
for e := 0; e < g; e++ {
    go func(emp int) {
        for p := range ch {
            fmt.Printf("employee %d : recv'd signal : %s\n", emp, p)
        }
        fmt.Printf("employee %d : recv'd shutdown signal\n", emp)
    }(e)
}
```

The for range helps to minimize the amount of code I would otherwise need to receive a signal and then shutdown once the channel is closed. Without the for/range mechanics, I would have to write this code.

```
for e := 0; e < g; e++ {
    go func(emp int) {
        for {
            p, wd := <-ch
            if !wd {
                break
            }
            fmt.Printf("employee %d : recv'd signal : %s\n", emp, p)
        }
        fmt.Printf("employee %d : recv'd shutdown signal\n", emp)
    }(e)
}
```

The for/range eliminates 4 extra lines of code and streamlines the mechanics. It's important to note, it must not matter which of the goroutines in the pool are chosen to receive a signal. Depending on the amount of work being signalled, it could be the same goroutines over and over while others are never selected.

Then the call to close is executed which will cause the for loops to terminate and stop the program. If the channel being used was a buffered channel, data would flush out of the buffer first before the goroutine would receive the close signal.

6.11.5 Drop

The drop pattern is an important pattern for services that may experience heavy loads at times and can drop requests when the service reaches a capacity of pending requests. As an example, a DNS service would need to employ this pattern.

```
func drop() {
    const cap = 100
    ch := make(chan string, cap)

    go func() {
        for p := range ch {
            fmt.Println("employee : recv'd signal :", p)
        }
    }()

    const work = 2000
    for w := 0; w < work; w++ {
        select {
        case ch <- "paper":
            fmt.Println("manager : sent signal :", w)
        default:
            fmt.Println("manager : dropped data :", w)
        }
    }
```

```
    close(ch)
    fmt.Println("manager : sent shutdown signal")

    time.Sleep(time.Second)
    fmt.Println("-------------------------------------------------")
}
```

The code starts with the creation of a buffered channel. This is a case where it's reasonable to have a large buffer. Identifying the capacity value (buffer size) will require work in the lab. I want a number that allows the service to maintain reasonable levels of resource usage and performance when the buffer is full.

Next a Goroutine using the pooling pattern is created. This Goroutine is waiting for a signal to receive data to work on. In this example, having only one Goroutine will cause back pressure quickly on the sending side. One Goroutine will not be able to process all the work in time before the buffer gets full. Representing the service is at capacity.

Inside the for loop, I see the use of a select statement. The select statement is a blocking call that allows a single Goroutine to handle multiple channel operations at the same time. Each case represents a channel operation, a send or a receive. However, this select is using the default keyword as well, which turns the select into a non-blocking call.

The key to implementing this pattern is the use of default. If the channel buffer is full, that will cause the case statement to block since the send can't complete. When every case in a select is blocked, and there is a default, the default is then executed. This is where the drop code is placed.

In the drop code, I can now decide what to do with the request. I can return a 500 to tell the caller. I could store the request somewhere else. The key is I have options.

6.11.6 Cancellation

The cancellation pattern is used to tell a function performing some I/O how long I am willing to wait for the operation to complete. Sometimes I can cancel the operation, and sometimes all I can do is just walk away.

```
func cancellation() {
    duration := 150 * time.Millisecond
    ctx, cancel := context.WithTimeout(context.Background(), duration)
    defer cancel()

    ch := make(chan string, 1)

    go func() {
        time.Sleep(time.Duration(rand.Intn(200)) * time.Millisecond)
        ch <- "paper"
    }()
```

```
    select {
    case d := <-ch:
        fmt.Println("work complete", d)

    case <-ctx.Done():
        fmt.Println("work cancelled")
    }

    time.Sleep(time.Second)
    fmt.Println("-------------------------------------------------")
}
```

The code starts with defining a time.Duration variable named time set to 150 milliseconds. Then a Context value is created to support a timeout of the 150 seconds using the WithTimeout function. That function takes a Context value in and returns a new one out with the changes. In this case, I use the Background function which returns an empty parent Context.

It's important to call the cancel function that is returned as the second argument from WithTimeout using a defer. If that cancel function is not called at least once, there will be a memory leak.

After a buffer channel one 1 is created, a worker Goroutine is created to perform some I/O bound work. In this case, a random Sleep call is made to simulate blocking work that can't be directly cancelled. That work can take up to 200 milliseconds to finish. There is a 50 millisecond difference between the timeout and the amount of time the work could take.

With the worker Goroutine created and performing the work, the main Goroutine blocks in a select statement waiting on two signals. The first case represents the working Goroutine finishing the work on time and the result being received. That is what I want. The second case represents a timeout from the Context. This means the work didn't finish within the 150 millisecond time limit.

If the main Goroutine received the timeout signal it walks away. In this situation, it can't inform the worker Goroutine that it won't be around to receive it's signal. This is why it's so important for the work channel to be a buffer of 1. The worker Goroutine needs to be able to send its signal, whether or not the main Goroutine is around to receive it. If a non-buffered channel is used, the worker Goroutine will block forever and become a memory leak.

6.11.7 Fan Out/In Semaphore

The fan out/in semaphore pattern provides a mechanic to control the number of Goroutines executing work at any given time while still creating a unique Goroutine for each piece of work.

```
func fanOutSem() {
    emps := 2000
    ch := make(chan string, emps)

    g := runtime.GOMAXPROCS(0)
    sem := make(chan bool, g)

    for e := 0; e < emps; e++ {
        go func(emp int) {
            sem <- true
            {
                t := time.Duration(rand.Intn(200)) * time.Millisecond
                time.Sleep(t)
                ch <- "paper"
                fmt.Println("employee : sent signal :", emp)
            }
            <-sem
        }(e)
    }

    for emps > 0 {
        p := <-ch
        emps--
        fmt.Println(p)
        fmt.Println("manager : recv'd signal :", emps)
    }

    time.Sleep(time.Second)
    fmt.Println("-------------------------------------------------")
}
```

At the start of the function, a channel with a buffer size of 2000 is set. This is the same thing we need in the original fan out/in pattern. One buffer for each Goroutine that will be created. Then like the pooling pattern, the use of the GOMAXPROCS function is used to determine how many of the 2000 Goroutines will be allowed to execute their work at any given time.

With g configured, a second buffered channel is constructed next with a buffer sized to the number of Goroutines that can execute their work at the same time. This channel is the semaphore that will control the number of Goroutines performing work.

Then a for loop is used to create all 2000 Goroutines and each Goroutine finds itself in a send operation (sem <- true) against the semaphore channel. Here is where the rubber hits the road. Only a GOMAXPROCS number of goroutines can perform this send without blocking. The other 2000 - GOMAXPROCS Goroutines will block until the running Goroutines get to the receive operation (<-sem). This code uses a code block to show the code that is being executed between the semaphore locking. I like this for better readability.

At the end of the function, the main Goroutine waits to receive work from all 2000 Goroutines. For each piece of work received, the emps variable is decremented until it gets down to zero. Just like the original fan out/in pattern.

6.11.8 Bounded Work Pooling

The bounded work pooling pattern uses a pool of Goroutines to perform a fixed amount of known work.

```
func boundedWorkPooling() {
    work := []string{"paper", "paper", "paper", "paper", 2000: "paper"}

    g := runtime.GOMAXPROCS(0)
    var wg sync.WaitGroup
    wg.Add(g)

    ch := make(chan string, g)

    for e := 0; e < g; e++ {
        go func(emp int) {
            defer wg.Done()
            for p := range ch {
                fmt.Printf("employee %d : recv'd signal : %s\n", emp, p)
            }
            fmt.Printf("employee %d : recv'd shutdown signal\n", emp)
        }(e)
    }

    for _, wrk := range work {
        ch <- wrk
    }
    close(ch)
    wg.Wait()

    time.Sleep(time.Second)
    fmt.Println("-------------------------------------------------")
}
```

Right from the start, the function defines 2000 arbitrary pieces of work to perform. Then the GOMAXPROCS function is used to define the number of Goroutines to use in the pool and a WaitGroup is constructed to make sure the main Goroutine can be told to wait until all 2000 pieces of work are completed.

Just like I saw with the pooling pattern, a pool of worker Goroutines is created in the loop and they all wait on a receive call using the for range mechanics. One change is the call to Done using a defer when each of the worker Goroutines in the pool eventually terminate. This will happen when all the work is completed and this is how the pool will report back to the main Goroutine they are aware they are not needed any longer.

After the creation of the pool of work Goroutines, A loop is executed by the main Goroutine to start signaling work into the pool. Once the last piece of work is signaled, the channel is closed. Each of the worker Goroutines will receive the closed signal once the signals in the buffer are emptied.

6.11.9 Retry Timeout

The retry timeout pattern is great when I have to ping something (like a database) which might fail, but I don't want to fail immediately. I want to retry for a specified amount of time before I fail.

```
func retryTimeout(ctx context.Context, retryInterval time.Duration,
                  check func(ctx context.Context) error) {

    for {
        fmt.Println("perform user check call")
        if err := check(ctx); err == nil {
            fmt.Println("work finished successfully")
            return
        }

        fmt.Println("check if timeout has expired")
        if ctx.Err() != nil {
            fmt.Println("time expired 1 :", ctx.Err())
            return
        }

        fmt.Printf("wait %s before trying again\n", retryInterval)
        t := time.NewTimer(retryInterval)

        select {
        case <-ctx.Done():
            fmt.Println("timed expired 2 :", ctx.Err())
            t.Stop()
            return
        case <-t.C:
        fmt.Println("retry again")
        }
    }
}
```

The function takes a context for the amount of time the function should attempt to perform work unsuccessfully. It also takes a retry interval that specifies how long to wait between attempts, and finally a function to execute. This function is coded by the caller for the specific work (like pinging the database) that needs to be performed and could fail.

The core of the function runs in an endless loop. The first step in the loop is to run the check function passing in the context so the caller's function can also respect the context. If that doesn't fail, the function returns that life is good. If it fails, the code goes on to the next step.

Next the context is checked to see if the amount of time given has expired. If it has, the function returns the timeout error, else it continues to the next step which is to create a timer value. The time value is set to the retry interval. The timer could be

created above the for loop and reused, which would be good if this function was going to be running a lot. To simplify the code, a new timer is created every time.

The last step is to block on a select statement waiting to receive one of two signals. The first signal is that the context expires. The second signal is the retry interval expires. In the case of the second signal, the loop is restarted and the process runs again.

6.11.10 Channel Cancellation

With channel cancellation, I can take an existing channel being used already for cancellation purposes (legacy code) and convert its use with a context, where a context is needed for a future function call.

```
func channelCancellation(stop <-chan struct{}) {
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()

    go func() {
        select {
        case <-stop:
            cancel()
        case <-ctx.Done():
        }
    }()

    func(ctx context.Context) error {
        req, err := http.NewRequestWithContext(
            ctx,
            http.MethodGet,
            "https://www.ardanlabs.com/blog/index.xml",
            nil,
       )
       if err != nil {
           return err
       }

       _, err = http.DefaultClient.Do(req)
       if err != nil {
           return err
       }
       return nil
    }(ctx)
}
```

This function accepts a channel typed with the empty struct to signal cancellation. This is code that could be found in Go programs prior to the inclusion of context. A function this function needs to call works with the “new” context package.

A context is created using the Background function for the parent context in the WithCancel call. This returns a new context value that can be cancelled with the returned cancel function.

The key is the creation of the Goroutine that blocks in a select statement waiting on two signals. The first signal is the legacy channel that may be closed by the originator. The second is the context itself, which is important if future functions decide to cancel the context directly. On receiving a stop signal, the cancel function is then executed, cancelling the context for all functions that were passed the context.

As an example, a literal function is declared and executed that performs a web request that supports a context for cancellable I/O.

Chapter 7 - Generics

7.1 Basic Syntax

If I want to write a single print function that can output a slice of any given type and not use reflection, I can use the new generics syntax.

```
func print[T any](slice []T) {
    fmt.Print("Generic: ")
    for _, v := range slice {
        fmt.Print(v, " ")
    }
    fmt.Print("\n")
}
```

This is an implementation of a single print function that can output a slice of any given type using the new generics syntax. What's nice about this syntax is that the code inside the function can use syntax and built-in functions that would work with a concrete type. This is not the case when I use the empty interface to write generic code.

I need a way to tell the compiler that I won't be declaring type T explicitly, but it has to be determined by the compiler at compile time. The new syntax uses square brackets for this. The brackets define a list of generic type identifiers that represent types specific to the function that need to be determined at compile time. It's how I tell the compiler that types with these names won't be declared before the program is compiled. These types need to be figured out at compile time.

Note: I can have multiple type identifiers defined inside the brackets though the current example is only using one. Ex. [T, S, R any]

I can name these type identifiers anything I want to help with the readability of the code. In this case, I'm using the capital letter T to describe that a slice of some type T (to be determined at compile time) will be passed in. I like the use of single capitalized letters when it comes to collections and it's also a convention that goes back to older programming languages like C++ and Java.

There is the use of the word any inside the brackets as well. This represents a constraint on what type T can be. The compiler requires that all generic types have a well defined constraint. The any constraint is predeclared by the compiler and states there are no constraints on what type T can be.

```
numbers := []int{1, 2, 3}
print[int](numbers)
```

```
strings := []string{"A", "B", "C"}
print[string](strings)

floats := []float64{1.7, 2.2, 3.14}
print[float64](floats)
```

This is how to make calls to the generic print function where the type information for T is explicitly provided at the call site. The syntax emulates the idea that the function declaration func name[T any](slice []T) { defines two sets of parameters. The first set is the type that maps to the corresponding type identifiers, and the second is the data that maps to the corresponding input variables.

Luckily, the compiler can infer the type and eliminate the need to explicitly pass in the type information at the call site.

```
numbers := []int{1, 2, 3}
print(numbers)

strings := []string{"A", "B", "C"}
print(strings)

floats := []float64{1.7, 2.2, 3.14}
print(floats)
```

This code shows how I can call the generic print functions without the need to pass the type information explicitly. At the function call site, the compiler is able to identify the type to use for T and construct a concrete version of the function to support slices of that type. The compiler has the ability to infer the type with the information it has at the call site from the data being passed in.

7.2 Underlying Types

What if I wanted to declare my own generic type using an underlying type?

```
type vector[T any] []T

func (v vector[T]) last() (T, error) {
    var zero T
    if len(v) == 0 {
        return zero, errors.New("empty")
    }
    return v[len(v)-1], nil
}
```

This example shows a generic vector type that restricts the construction of a vector to a single type of data. The use of square brackets declares that type T is a generic type to be determined at compile time. The use of the constraint any describes there is no constraint on what type T can become.

The last method is declared with a value receiver of type vector[T] to represent a value of type vector with an underlying slice of some type T. The method returns a value of that same type T.

```
func main() {
    fmt.Print("vector[int] : ")
    vGenInt := vector{10, -1}
    i, err = vGenInt.last()
    if i < 0 {
        fmt.Print("negative integer: ")
    }
    fmt.Printf("value: %d error: %v\n", i, err)

    fmt.Print("vector[string] : ")
    vGenStr := vector{"A", "B", string([]byte{0xff})}
    s, err = vGenStr.last()
    if !utf8.ValidString(s) {
        fmt.Print("non-valid string: ")
    }
    fmt.Printf("value: %q error: %v\n", s, err)
}

Output:
vector[int] : negative integer: value: -1 error: <nil>
vector[string] : non-valid string: value: "\xff" error: <nil>
```

This is how to construct a value of type vector with an underlying type of int when I will set values in the vector at construction. An important aspect of this code is the construction calls.

```
// Zero Value Construction
var vGenInt vector[int]
var vGenStr vector[string]

// Non-Zero Value Construction
vGenInt := vector{10, -1}
vGenStr := vector{"A", "B", string([]byte{0xff})}
```

When it comes to constructing these generic types to their zero value, it’s not possible for the compiler to infer the type. However, in cases where there is initialization during construction, the compiler can infer the type.

There is an aspect of the spec that focuses on the construction of a generic type to its zero value state.

```
type vector[T any] []T

func (v vector[T]) last() (T, error) {
    var zero T
    if len(v) == 0 {
        return zero, errors.New("empty")
    }
```

```
    return v[len(v)-1], nil
}
```

I need to focus on the method declaration for last and how the method returns a value of the generic type T. On the first return is a situation where I need to return the zero value for type T. The current draft provides two solutions to write this code. The first solution I see already. A variable named zero is constructed to its zero value state of type T and then that variable is used for the return.

The other option is to use the built-in function new and deference the returned pointer within the return statement.

```
type vector[T any] []T

func (v vector[T]) last() (T, error) {
    if len(v) == 0 {
        return *new(T), errors.New("empty")
    }
    return v[len(v)-1], nil
}
```

This version of last is using the built-in function new for zero value construction and dereferencing of the returned pointer to satisfy return type T.

Note: I might think why not use T{} to perform zero value construction? The problem is this syntax does not work with all types, such as the scalar types (int, string, bool). So it's not an option.

7.3 Struct Types

What if I wanted to declare my own generic type using a struct type?

```
type node[T any] struct {
    Data T
    next *node[T]
    prev *node[T]
}
```

This struct type is declared to represent a node for the linked list. Each node contains an individual piece of data that is stored and managed by the list. The use of square brackets declares that type T is a generic type to be determined at compile time. The use of the constraint any describes there is no constraint on what type T can become.

With type T declared, the Data field can now be defined as a field of some type T to be determined later. The next and prev fields need to point to a node of that same type T. These are the pointers to the next and previous node in the linked list, respectively. To

make this connection, the fields are declared as pointers to a node that is bound to type T through the use of the square brackets.

```
type list[T any] struct {
    first *node[T]
    last  *node[T]
}
```

The second struct type is named list and represents a collection of nodes by pointing to the first and last node in a list. These fields need to point to a node of some type T, just like the next and prev fields from the node type.

Once again, the identifier T is defined as a generic type (to be determined later) that can be substituted for "any" concrete type. Then the first and last fields are declared as pointers to a node of some type T using the square bracket syntax.

```
func (l *list[T]) add(data T) *node[T] {
    n := node[T]{
        Data: data,
        prev: l.last,
    }
    if l.first == nil {
        l.first = &n
        l.last = &n
        return &n
    }
    l.last.next = &n
    l.last = &n
    return &n
}
```

This is an implementation of a method named add for the list type. No formal generic type list declaration is required (as with functions) since the method is bound to the list through the receiver. The add method's receiver is declared as a pointer to a list of some type T and the return is declared as a pointer to a node of the same type T.

The code after the construction of a node will always be the same, regardless of what type of data is being stored in the list since that is just pointer manipulation. It's only the construction of a new node that is affected by the type of data that will be managed. Thanks to generics, the construction of a node can be bound to type T which gets substituted later at compile time.

Without generics, this entire method would need to be duplicated since the construction of a node would need to be hard coded to a known, declared type prior to compilation. Since the amount of code (for the entire list implementation) that needs to change for different data types is very small, being able to declare a node and list to manage data of some type T reduces the cost of code duplication and maintenance.

```
type user struct {
    name string
}

func main() {

    // Store values of type user into the list.
    var lv list[user]
    n1 := lv.add(user{"bill"})
    n2 := lv.add(user{"ale"})
    fmt.Println(n1.Data, n2.Data)

    // Store pointers of type user into the list.
    var lp list[*user]
    n3 := lp.add(&user{"bill"})
    n4 := lp.add(&user{"ale"})
    fmt.Println(n3.Data, n4.Data)
}

Output:
{bill} {ale}
&{bill} &{ale}
```

Here is a small application. A type name user is declared and then a list is constructed to its zero value state to manage values of type user. A second list is then constructed to its zero value state and this list manages pointers to values of type user. The only difference between these two lists is one manages values of type user and the other pointers of type user.

Since type user is explicitly specified during the construction of the list, the add method in turn accepts values of type user. Since a pointer of type user is explicitly specified during the construction of the list, the add method accepts pointers of type user.

I can see in the output of the program, the Data field for the nodes in the respective lists match the data semantic used in the construction.

Chapter 8 - Conclusion

The content in this book for version 0.3 is only the beginning. We do believe these 100 pages are a good start and can help a lot of people.

If you find issues, typos, anything that is inaccurate, or can be improved, please send Bill an email at bill@ardanlabs.com.

Our plan is to focus on this book more than the blog for a while. Maybe incorporate more of the blog content here instead of providing links. There is a lot more to be done, but this is a good start.

Thanks,
-- Bill Kennedy
-- Hoanh An

ULTIMATE GO NOTEBOOK

HOANH AN

Hoanh An is a software engineer with a background in distributed systems and the author of several popular open-source projects. In his free time, he enjoys reading and sharing his thoughts on different topics of software engineering, product development, and entrepreneurship on his blog.

WILLIAM KENNEDY

Bill has been developing software for more than 30 years. In 2013 he became a pioneer using Go and now has trained over 10,000 engineers that work for Fortune 100 companies. He also is the author of Go in Action and is the main contributor to our blog. He can be found online at **www.ardanlabs.com**