Practical Go: Real world advice for writing maintainable Go programs

Dave Cheney

Introduction

Hello,

Thank you for joining me today.

The story of how this workshop came about needs some exposition.

A few years ago I made a tweet called "three rules for dating my code base", it was intended to be a shortlist of the things you should do for a healthy Go project. Tweet sizes being what they were at the time, this tweet was followed a day later with another three things, and I could have probably continued for a good week.

Related to this, in 2015 I had dabbled with an agreement to write a book for O'Reilly. The project fizzled out after a few months for two reasons:

O'Reilly's document preparation system was good, but not as flexible as I wanted. There were too many undefined steps between my words and the final book.

The unforgiving calculus of the number of hours a project like this would take meant if I failed to find the hundreds of hours necessary there would be no partial credit — O'Reilly may re-author the book, or it may never see the light of day.

Both of these struck me as rather old world approaches to developing a book. The parallel with many departments hand carrying software from one checklist to another, and the distinctly non agile way that complete manuscript was copy edited, was not lost on me.

After the "three rules" tweet I began to think again about how I would like to see a book about Go written in a way that delivered continual benefit to its potential audience even if I were to drop out at an arbitrary point.

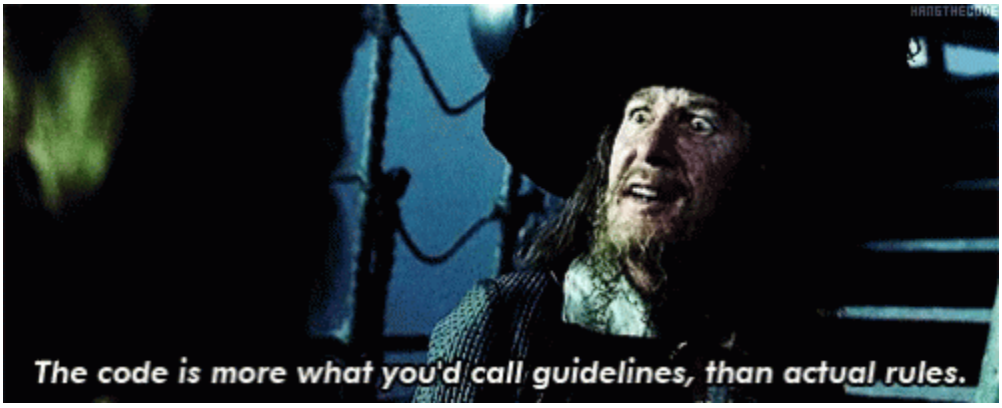
This workshop, the document you are reading, the articles on my blog, and the presentations I give are my answer to this problem. This document represents the best version of the material I want to write about that exists today. As I present more, give more workshops, the material will grow, however, Go programmers continually reap the benefits of this material without waiting for me to declare arbitrarily that Practical Go is *done*.

Today this is a workshop style presentation, I'm going to dispense with the usual slide deck and we'll work

directly from this document which you can take away with you today.

Caveat emptor

Before I begin I want to mention that the more I research, read, talk to folks, and think about programming, the more I'm convinced that there are no *rules*. There are only guidelines.



What's the difference? I'd say probably experience.

The mastery of a topic, Malcolm Gladwell's 10,000 hour yardstick, is equal parts observation and practice. You watch someone else do the thing, while concurrently you practice doing the thing.

Knowing when to stick to the code and when to bend the rules probably comes down to experience, gained from observation, experimentation and research.

With that said, these are my experiences, and the goal today is not to be prescriptive. This is a discussion, not a lecture.

1. Guiding principles

If I'm going to talk about best practices in any programming language I need some framework by which to define what I mean by *best*.

Last year Bryan Cantrill gave a wonderful presentation on operating system principles, wherein he highlighted that different programming languages or operating systems have different core values. It is not that they *ignore* the principles that differ between their competitors, just that when the chips are down, they each choose indepdendantly, accoding to the core values of their respective projects.

So what is that core set of prinicples for Go?

You may have seen this quote from the Go team lead, Russ Cox:

> Software engineering is what happens to programming when you add time and other programmers.
>
> — Russ Cox

Russ is making the distinction between software *programming* and software *engineering*. The former is a program you write for yourself, the latter is a product that many people will work with over time. Engineers will come and go, teams will grow and shrink, requirements will change, features will be added and bugs fixed. This is the nature of software engineering.

I'm possibly one of the earliest users of Go in this room, but to argue that my seniority gives my views more weight is *false*. Instead, the advice I'm going to present today is informed by what I believe to be the guiding principles underlying Go itself. They are:

Clarity

Simplicity

Productivity

Note	You'll note that I didn't say *performance*, or *concurrency*. There are languages which are a bit faster than Go, but they're certainly not as simple as Go. There are languages which make concurrency their highest goal, but they are not as readable, nor as productive. Performance and concurrency are important attributes, but not as important as *clarity*, *simplicity*, and *productivity*.

1.1. Clarity

> Programs must be written for people to read, and only incidentally for machines to execute.
>
> — Hal Abelson and Gerald Sussman
> *Structure and Interpretation of Computer Programs*

Code is read many more times than it is written. A single piece of code will, over its lifetime, be read hundreds, maybe thousands of times.

Clarity is important because all software, not just Go programs, is written by humans to be read by other humans. The fact that software is also consumed by machines is secondary.

> The most important skill for a programmer is the ability to effectively communicate ideas.
>
> — Gastón Jorquera [1]

If you're writing a program for yourself, maybe it only has to run once, or you're the only person who'll ever see it, then do what ever works for you. But if this is a piece of software that more than one person will contribute to, or that will be used by people over a long enough time that requirements, features, or the environment it runs in may change, then your goal must be for your program to be *maintainable*.

1.2. Simplicity

> Simplicity is prerequisite for reliability.
>
> — Edsger W. Dijkstra

Why should we strive for simplicity? Why is important that Go programs be simple?

> Controlling complexity is the essence of computer programming.
>
> — Brian Kernighan

We've all been in a situation where you say "I can't understand this code", yes? We've all worked on programs where you're scared to make a change because you're worried it'll break another part of the program; a part you don't understand and don't know how to fix. This is complexity.

> There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.
>
> — C. A. R. Hoare

Complexity turns reliable software in unreliable software. Complexity is what kills software projects. Therefore simplicity is the highest goal of Go. Whatever programs we write, we should be able to agree that they are simple.

1.3. Productivity

> Design is the art of arranging code to work today, and be changeable forever.
>
> — Sandi Metz

The last underlying principle I want to highlight is *productivity*. Developer productivity is a sprawling topic but it boils down to this; how much time do you spend doing useful work verses waiting for your tools or hopelessly lost in a foreign code-base. Go programmers should feel that they can get a lot done with Go.

The joke goes that Go was designed while waiting for a C++ program to compile. Fast compilation is a key feature of Go and a key recruiting tool to attract new developers. While compilation speed remains a constant battleground, it is fair to say that operations which take minutes in other languages, take seconds in Go. This helps Go developers feel as productive as their counterparts working in dynamic languages without the reliability issues inherent in those languages.

More fundamental to the question of developer productivity, Go programmers realise that code is written to be read and so place the act of reading code above the act of writing it. Go goes so far as to enforce, via tooling and custom, that all code be formatted in a specific style. This removes the friction of learning a project specific dialect and helps spot mistakes because they just *look* incorrect.

Go programmers don't spend days debugging inscrutable compile errors. They don't waste days with

complicated build scripts or deploying code to production. And most importantly they don't spend their time trying to understand what their coworker wrote.

To say that Go is a language designed to be productive is an understanding it is built for software design in the large, at industrial scale.

Productivity is what the Go team mean when they say the language must *scale*.

2. Identifiers

The first topic I'd like to discuss is *identifiers*. An identifier is a fancy word for a *name*; the name of a variable, the name of a function, the name of a method, the name of a type, the name of a package, and so on.

> Poor naming is symptomatic of poor design.

Given the limited syntax of Go, the names we choose for things in our programs have an oversized impact on the readability of our programs. Readability is a defining quality of good code, thus choosing good names is crucial to the readability of Go code.

2.1. Choose identifiers for clarity, not brevity

> Obvious code is important. What you *can* do in one line you *should* do in three.

Go is not a language that optimises for clever one liners. Go is not a language which optimises for the least number of lines in a program. We're not optimising for the size of the source code on disk, nor how long it takes to type the program into an editor. Rather, we want to optimise our code to be clear to the reader.

> Good naming is like a good joke. If you have to explain it, it's not funny.

Key to this clarity is the names we choose for identifies in Go programs. Let's talk about the qualities of a good name:

A good name is concise. A good name need not be the shortest possible, but a good name should waste no space on things which are extraneous. Good names have a high signal to noise ratio.

A good name is descriptive. A good name should describe the application of a variable or constant, *not* their contents. A good name should describe the result of a function, or behaviour of a method, *not* their implementation. A good name should describe the purpose of a package, *not* its contents. The better the choice of a name, the more accurately it describes the thing it identifies.

A good name should be predictable. You should be able to infer the way a symbol will be used from its name alone. This is a function of choosing descriptive names, but it also about following tradition. This is what Go programmers talk about when they say *idiomatic*.

Let's talk about each of these properties in depth.

2.2. Identifier length

Sometimes people criticise the Go style for recommending short variable names. As Rob Pike said, "Go programmers want the *right* length identifiers". [1]

Andrew Gerrand suggests that by using longer identifies to indicate to the reader things of higher importance.

> The greater the distance between a name's declaration and its uses, the longer the name should be.
>
> — Andrew Gerrand [2]

From this we can draw some guidelines:

Short variable names work well when the distance between their declaration and *last* use is short.

Long variable names need to justify themselves; the longer they are the more value they need to provide. Lengthy bureaucratic names carry a low amount of signal compared to their weight on the page.

Don't include the name of your type in the name of your variable.

Constants should describe the value they hold, *not* how that value is used.

Prefer single letter variables for loops and branches, single words for parameters and return values, multiple words for functions and package level declarations

Prefer single words for methods, interfaces, and packages.

Remember that the name of a package is part of the name the caller uses to to refer to it, so make use of that.

Let's look at an example:

```
type Person struct {
        Name string
        Age  int
}

// AverageAge returns the average age of people.
func AverageAge(people []Person) int {
        if len(people) == 0 {
                return 0
        }

        var count, sum int
        for _, p := range people {
                sum += p.Age
```

```
                count += 1
        }

        return sum / count
}
```

In this example, the range variable `p` is declared on line 10 and only referenced once, on the following line. `p` lives for a very short time on the page and in limited scope during the execution of the function. A reader who is interested in the effect values of `p` have on the program need only read the three loop's three lines.

By comparison `people` is declared in the function parameters, is live for the body of the function, and is referenced three times over seven lines. The same is true for `sum`, and `count`, thus they justify their longer names. The reader has to scan a wider number of lines to locate them so they are given more distinctive names.

I could have chosen `s` for `sum` and `c` (or possibly `n`) for `count` but this would have reduced all the variables in the program to the same level of importance. I could have chosen `p` instead of `people` but that would have left the problem of what to call the `for … range` iteration variable and the singular `person` would look odd as the loop iteration variable which lives for little time has a longer name than the slice of values it was derived from.

Tip	Use blank lines to break up the flow of a function in the same way you use paragraphs to break up the flow of a document. In `AverageAge` we have three operations occurring in sequence. The first is the precondition, checking that we don't divide by zero if people is empty, the second is the accumulation of the sum and count, and the final is the computation of the average.

2.2.1. Context is key

It's important to recognise that most advice on naming is contextual. I like to say it is a guideline, not a rule.

What is the difference between two identifiers, `i`, and `index`. We cannot say conclusively that one is better than another, for example is

```
for index := 0; index < len(s); index++ {
        //
}
```

fundamentally more readable than

```
for i := 0; i < len(s); i++ {
        //
}
```

I argue it is not, because it is likely the scope of `i`, and `index` for that matter, is limited to the body of the `for`

loop and the extra verbosity of the latter adds little to *comprehension* of the program.

However, which of these functions is more readable?

```
func (s *SNMP) Fetch(oid []int, index int) (int, error)
```

versus

```
func (s *SNMP) Fetch(o []int, i int) (int, error)
```

In this example, `oid` is an abbreviation for SNMP Object ID, so shortening it to `o` would mean programmers have to translate from the common notation that they read in documentation to the shorter notation in your code. Similarly, reducing `index` to `i` obscures what `i` stands for; in SNMP messages a sub value of each OID is called an Index.

Tip	Don’t mix and match long and short formal parameters in the same declaration.

2.3. A variable’s name should describe its contents

You shouldn’t name your variables after their types for the same reason you don’t name your pets "dog" and "cat". You shouldn’t include the name of your type in the name of your variable’s name for the same reason.

The name of the variable should describe its contents, not the *type* of its contents. Consider this example:

```
var usersMap map[string]*User
```

What’s good about this declaration? We can see that its a map, and it has something to do with the `*User` type, that’s probably good. But `usersMap` *is* a map, and Go being a statically typed language won’t let us accidentally use it where a different type is required. The `Map` suffix is redundant from the point of view of the compiler. Hence utility of the suffix is entirely down to whether we can prove it is of use to the reader.

Now, consider what happens if we were to declare other variables:

```
var (
        companiesMap map[string]*Company
        productsMap  map[string]*Products
)
```

Now we have three map type variables in scope, `usersMap`, `companiesMap`, and `productsMap`, all mapping strings to different types. We know they are maps; it’s right there in their declaration. We also know that their map declarations prevent us from using one in place of another—the compiler will throw an error if we try to use `companiesMap` where code was expecting a `map[string]*User`. In this situation it’s clear that the `Map` suffix does not improve the clarity of the code, its just extra boilerplate to type.

Removing the suffix leaves us with the more concise and equally descriptive:

```
var (
        users     map[string]*User
        companies map[string]*Company
        products  map[string]*Products
)
```

Note	usersMap versus userMap If we remove the suffix denoting the type's type from its name, `usersMap` becomes `users` which is descriptive, but `userMap` becomes `user` which is misleading.

My suggestion is to avoid any suffix that resembles the type of the variable.

Tip	If `users` isn't descriptive enough, then `usersMap` won't be either.

This advice also applies to function parameters. For example:

```
type Config struct {
        //
}

func WriteConfig(w io.Writer, config *Config)
```

Naming the `*Config` parameter `config` is redundant. We know its a `*Config`, it says so right there.

In this case consider `conf` or maybe `c` will do if the lifetime of the variable is short enough.

If there is more that one `*Config` in scope at any one time then calling them `conf1` and `conf2` is less descriptive than calling them `original` and `updated` as the latter are less likely to be mistaken for one another.

Tip	Don't let package names steal good variable names.
	
	The name of an imported identifier includes its package name. For example the `Context` type in the `context` package will be known as `context.Context`. This makes it impossible to use `context` as a variable or type in your package.
	
	`func WriteLog(context context.Context, message string)`
	
	Will not compile. This is why the local declaration for `context.Context` types is traditionally `ctx`. eg.
	
	`func WriteLog(ctx context.Context, message string)`

2.4. Use a consistent naming style

Another property of a good name is it should be predictable. The reader should be able to understand the use of a name when they encounter it for the first time. When they encounter a *common* name, they should be able to

assume it has not changed meanings since the last time they saw it. You could also say that a good name should feel familiar.

For example, if your code passes around a database handle, make sure each time the parameter appears it has the same name. Rather than a combination of `d *sql.DB`, `dbase *sql.DB`, `DB *sql.DB`, and `database *sql.DB`, instead consolidate on something like;

and use it consistently across parameters, return values, local declarations, and potentially receivers. Doing so promotes familiarity; if you see a `db`, you know it's a `*sql.DB` and that it has either been declared locally or provided for you by the caller.

Similar advice applies to method receivers; use the same receiver name every method on that type. This makes it easier for the reader to internalise the use of the receiver across the methods in this type.

Note	The convention for short receiver names in Go is at odds with the advice provided so far. This is just one of the choices made early on that has become the preferred style, just like the use of `CamelCase` rather than `snake_case`.

Tip	Go style dictates that receivers have a single letter name, or acronyms derived from their type. You may find that the name of your receiver sometimes conflicts with name of a parameter in a method. In this case, consider making the parameter name slightly longer, and don't forget to use this new parameter name consistently.

Finally, certain single letter variables have traditionally been associated with loops and counting. For example, `i`, `j`, and `k` are commonly the loop induction variable for simple `for` loops. [2] `n` is commonly associated with a counter or accumulator. `v` is a common shorthand for a value in a generic encoding function, `k` is commonly used for the key of a map. `a` and `b` are generic names for parameters comparing two variables of the same type. `x` and `y` are generic names for local variables created for comparision, and `s` is often used as shorthand for parameters of type `string`.

As with the `db` example above programmers *expect* `i` to be a loop induction variable. If you ensure that `i` is *always* a loop variable, not used in other contexts outside a `for` loop. When readers encounter a variable called `i`, or `j`, they know that a loop is close by.

Tip	If you found yourself with so many nested loops that you exhaust your supply of `i`, `j`, and `k` variables, its probably time to break your function into smaller ones.

2.5. Use a consistent declaration style

Go has at least six different ways to declare a variable

```
x := 1

var y = 2
```

```
var z int = 3

var a int; a = 4

var b = int(5)

c := int(6)
```

This list does not include receivers, formal parameters and named return values. I'm sure there are more that I haven't thought of.

This is something that Go's designers recognise was probably a mistake, but its too late to change it now, and, they argue, the bigger problem is shadowing. With all these different ways of declaring a variable, how do we avoid each Go programmer choosing their own style?

In Go each variable has a purpose because each variable we declare has to be used within the same scope. Here is a suggestion for how to make the purpose of each declaration clear to the reader. This is the style I try to use where possible.

When declaring, but not initialising, a variable, use `var`. When declaring a variable that will be explicitly initialised later, use the `var` keyword.

```
var players int    // 0

var things []Thing // an empty slice of Things

var thing Thing    // empty Thing struct
json.Unmarshall(reader, &thing)
```

The `var` acts as a clue to say that this variable has been *deliberately* declared as the zero value of the indicated type. This is also consistent with the requirement to declare variables at the package level using `var` as opposed to the short declaration syntax. I'll argue later that you shouldn't be using package level variables at all.

When declaring and initialising, use `:=`. When declaring *and* initialising the variable at the same time—that is to say we're not letting the variable be implicitly initialised to its zero value—I recommend using the short variable declaration form. This makes it clear to the reader that the variable on the left hand side of the `:=` is being deliberately initialised to the expression on the right.

To explain why, Let's look at the previous example, but this time deliberately initialising each variable:

```
var players int = 0

var things []Thing = nil
```

```
var thing *Thing = new(Thing)
json.Unmarshall(reader, thing)
```

In the first and third examples, because in Go there are no automatic conversions from one type to another; the type on the left hand side of the assignment operator *must* be identical to the type on the right hand side. The compiler can infer the type of the variable being declared from the type on the right hand side, to the example can be written more concisely like this:

```
var players = 0

var things []Thing = nil

var thing = new(Thing)
json.Unmarshall(reader, thing)
```

This leaves us with explicitly initialising `players` to `0` which is redundant because `0` *is* `players' zero value. So it's better to make it clear that we're going to use the zero value by instead writing

What about the second statement? We cannot elide the type and write

Because `nil` does not have a type. [3] Instead we have a choice, do we want the zero value for a slice?

or do we want to create a slice with zero elements?

```
var things = make([]Thing, 0)
```

If we wanted the latter then this is *not* the zero value for a slice so we should make it clear to the reader that we're making this choice by using the short declaration form:

```
things := make([]Thing, 0)
```

Which tells the reader that we have chosen to initialise `things` explicitly.

This brings us to the third declaration,

Which is both explicitly initialising a variable and introduces the uncommon use of the `new` keyword which some Go programmer dislike. If we apply our short declaration syntax recommendation then the statement becomes

Which makes it clear that `thing` is explicitly initialised to the result of the expression `new(Thing)`--a pointer to a `Thing`--but still leaves us with the unusual use of `new`. We could address this by using the *compact literal* struct initialiser form,

Which does the same as `new(Thing)`, hence why some Go programmers are upset by the duplication. However this means we're explicitly initialising `thing` with a pointer to the literal `Thing{}`, which itself is

the zero value for a `Thing`.

Instead we should recognise that `thing` is being declared as its zero value and use the address of operator to pass the address of `thing` to `json.Unmarshall`

```
var thing Thing
json.Unmarshall(reader, &thing)
```

Note	Of course, with any rule of thumb, there are exceptions. For example, sometimes two variables are closely related so writing would look odd. The declaration may be more readable like this

In summary:

When declaring a variable without initialisation, use the `var` syntax.

When declaring and explicitly initialising a variable, use `:=`.

Tip	Make tricky declarations obvious When something is complicated, it should *look* complicated. Here length may be being used with a library which requires a specific numeric type and is more explicit that `length` is being explicitly declared to be `uint32` than the short declaration form: In the first example I'm deliberately breaking my rule of using the `var` declaration form with an explicit initialiser. This decision to vary from my usual form is a clue to the reader that something unusual is happening.

2.6. Be a team player

I talked about a goal of software engineering to produce maintainable code. Therefore you will likely spend most of your career working on projects of which you are not the sole author. My advice in this situation is: follow the local style.

Changing styles in the middle of a file is jarring. Uniformity, even if its not your preferred approach, is more valuable for maintenance over the long run than your personal preference. The rule I try to follow is; if it fits through `go fmt` then it's usually not worth holding up a code review for.

Tip	If you want to do a renaming across a codebase, do not mix this into another change. If someone is using git bisect they don't want to wade through thousands of lines of renaming to find the code you changed as well.

Before we move on to larger items I want to spend a little time talking about comments.

> Good code has lots of comments, bad code *requires* lots of comments.
>
> — Dave Thomas and Andrew Hunt

The Pragmatic Programmer

Comments are very important to the readability of a Go program. Each comments should do one—and only one—of three things:

The comment should explain *what* the thing does.

The comment should explain *how* the thing does what it does.

The comment should explain *why* the thing is the way it is.

The first form is ideal for commentary on public symbols:

```
The second form is ideal for commentary inside a method:
```

```
var results []chan error
for _, dep := range a.Deps {
        results = append(results, execute(seen, dep))
}
```

The third form, the *why* , is unique as it does not displace the first two, but at the same time it's not a replacement for the *what*, or the *how*. The *why* style of commentary exists to explain the external factors that drove the code you read on the page. Frequently those factors rarely make sense taken out of context, the comment exists to provide that context.

```
return &v2.Cluster_CommonLbConfig{
        // Disable HealthyPanicThreshold
                // See https://www.envoyproxy.io/docs/envoy/v1.9.0/intro
/arch_overview/load_balancing/panic_threshold#arch-overview-load-
balancing-panic-threshold
        HealthyPanicThreshold: &envoy_type.Percent{
                Value: 0,
        },
}
```

In this example it may not be immediately clear what the effect of setting `HealthyPanicThreshold` to zero percent will do. The comment is needed to clarify that the value of `0` will disable the panic threshold behaviour.

Comments such as these record hard won battles for understanding deep in the business logic. When you have the opportunity to write them, be sure to include enough hints that the next reader can follow your research. Links to issues, design documents, RFCs, or specifications that provide more background are always helpful.

3.1. Comments on variables and constants should describe their contents not their purpose

I stated earlier that the name of a variable, or a constant, should describe its purpose. When you add a comment to a variable or constant, that comment should describe the variables *contents*, not the variables *purpose*.

```
const RandomNumber = 6 // determined from roll of an unbiased die
```

In this example the comment describes *why* `RandomNumber` is assigned the value six, and where the six was derived from. The comment *does not* describe where `RandomNumber` will be used. This is deliberate, `RandomNumber` may be used many times by any package that references it. It is not possible to keep a record of all those uses at the site that `RandomNumber` is declared. Instead the name of the constant should be a guide the appropriate use for potential users.

Here are some more examples:

```
const (
    StatusContinue           = 100 // RFC 7231, 6.2.1
    StatusSwitchingProtocols = 101 // RFC 7231, 6.2.2
    StatusProcessing         = 102 // RFC 2518, 10.1

    StatusOK                 = 200 // RFC 7231, 6.3.1
```

In general use the untyped constant `100` is just the number one hundred. *In the context of HTTP* the number `100` is known as `StatusContinue`, as defined in RFC 7231, section 6.2.1. The comment included with that declaration helps the reader understand *why* `100` has special significance as a HTTP response code.

Tip	For variables without an initial value, the comment should describe who is responsible for initialising this variable.
	`// sizeCalculationDisabled indicates whether it is safe`
	`// to calculate Types' widths and alignments. See dowidth.`
	`var sizeCalculationDisabled bool`
	This example comes deep from the bowels of the Go compiler. Here, the comment lets the reader know that the `dowidth` function is responsible for maintaining the state of `sizeCalculationDisabled`.
	The fact that this advice runs contrary to previous advice that comments should not describe who uses them is a hint that `dowidth` and `sizeCalculationDisabled` are intimately entwined. The comments presence suggests a possible design weakness.

Tip	Hiding in plain sight
	This is a tip from Kate Gregory. [4] Sometimes you'll find a better name for a variable hiding in a comment.
	`// registry of SQL drivers`
	`var registry = make(map[string]*sql.Driver)`
	The comment was added by the author because `registry` doesn't explain enough about its purpose—it's a registry, but a registry of what?
	By renaming the variable to `sqlDrivers` its now clear that the purpose of this variable is to hold SQL

drivers.

```
var sqlDrivers = make(map[string]*sql.Driver)
```

Now the comment is redundant and can be removed.

3.2. Always document public symbols

Because godoc *is* the documentation for your package, you should always add a comment for every public symbol—variable, constant, function, and method—declared in your package.

Here are two rules from the Google Style guide:

Any public function that is not both obvious *and* short must be commented.

Any function in a library must be commented regardless of length or complexity.

```
package ioutil

// ReadAll reads from r until an error or EOF and returns the data it
read.
// A successful call returns err == nil, not err == EOF. Because ReadAll
is
// defined to read from src until EOF, it does not treat an EOF from Read
// as an error to be reported.
func ReadAll(r io.Reader) ([]byte, error)
```

There is one exception to this rule; you don't need to document methods that implement an interface. Specifically don't do this:

```
// Read implements the io.Reader interface
func (r *FileReader) Read(buf []byte) (int, error)
```

This comment says nothing. It doesn't tell you what the method does, in fact it's worse, it tells you to go look somewhere else for the documentation. In this situation I suggest removing the comment entirely.

Here is an example from the `io` package

```
// LimitReader returns a Reader that reads from r
// but stops with EOF after n bytes.
// The underlying implementation is a *LimitedReader.
func LimitReader(r Reader, n int64) Reader { return &LimitedReader{r, n} }

// A LimitedReader reads from R but limits the amount of
// data returned to just N bytes. Each call to Read
```

```
// updates N to reflect the new amount remaining.
// Read returns EOF when N <= 0 or when the underlying R returns EOF.
type LimitedReader struct {
        R Reader // underlying reader
        N int64  // max bytes remaining
}

func (l *LimitedReader) Read(p []byte) (n int, err error) {
        if l.N <= 0 {
                return 0, EOF
        }
        if int64(len(p)) > l.N {
                p = p[0:l.N]
        }
        n, err = l.R.Read(p)
        l.N -= int64(n)
        return
}
```

Note that the `LimitedReader` declaration is directly preceded by the function that uses it, and the declaration of `LimitedReader.Read` follows the declaration of `LimitedReader` itself. Even though `LimitedReader.Read` has no documentation itself, its clear from that it is an implementation of `io.Reader`.

Tip	Before you write the function, write the comment describing the function. If you find it hard to write the comment, then it's a sign that the code you're about to write is going to be hard to understand.

> Don't comment bad code, rewrite it.
>
> — Brian Kernighan

Comments highlighting the grossness of a particular piece of code are not sufficient. If you encounter one of these comments, you should raise an issue as a reminder to refactor it later. It is okay to live with technical debt, as long as the amount of debt is known.

The tradition in the standard library is to annotate a TODO style comment with the username of the person who noticed it.

```
// TODO(dfc) this is O(N^2), find a faster way to do this.
```

The username is not a promise that that person has committed to fixing the issue, but they may be the best person to ask when the time comes to address it.

> Good code is its own best documentation. As you're about to add a comment, ask yourself, 'How can I improve the code so that this comment isn't needed?' Improve the code and then document it to make it even clearer.

— Steve McConnell

Functions should do one thing only. If you find yourself commenting a piece of code because it is unrelated to the rest of the function, consider extracting it into a function of its own.

In addition to being easier to comprehend, smaller functions are easier to test in isolation. Once you've isolated the orthogonal code into its own function, its name may be all the documentation required.

4. Package Design

> Write shy code - modules that don't reveal anything unnecessary to other modules and that don't rely on other modules' implementations.

In his book, *Test Driven Design and Development*, Kent Beck describes the idea of a unit of software. In software, the unit is atomic, indivisible.

In the physical world atoms are composed of quarks, mesons, bosons, and gluons. We cannot observe them directly, only infer them from their *behaviour*--mass, charge, gravitational attraction. In the software world, if a unit is composed of smaller subatomic particles, as a user—a caller of that software—we are unable to directly observe the imlementation details of the unit. Instead we rely on the *behaviour* of a unit.

The size of a unit of software differs by language. In C the unit is a function, as C offers little else. In Java, the unit of software is commonly *mis_believed to be the class. In Go, the unit of software is not the function, or the type, or the method, but instead the _package*.

Just as the implementation of a function or method is unimportant to the caller, the implementation of the functions, methods and types that comprise your package's public API—its behaviour—is unimportant for the caller. The public API of a package describes *what* it does not *how* it does it. Moreover, when designed well, the *implementation* of your package is obscured from the caller

In this section we'll talk about designing a package around its behaviour as exposed via its public API.

4.1. A good package starts with its name

If the goal of a well designed Go package is to provide a set of related behaviours, writing a good package starts with choosing a good name. Think of your package's name as a one world elevator pitch to describe what your package can do for you elevator companion.

Just as I talked earlier about nameing variables, the name of a package is very important. I start by asking myself questions like, "what is the purpose of this package" or "what does service does package provide?". Hopefully the answer to that question is "this package let's you speak HTTP", not "this package provides the X

type", otherwise its time to go back to the drawing board.

Tip	Name your package for what it *provides*, not what it *contains*.

4.2. Good package names should be unique.

Within your project, each package name should be unique. This should pretty easy to if you've followed the previous advice that a package's name should derive from its purpose. If you find you have two packages which need the same name, it is likely either;

The name of the package is too generic--`client`, `worker`, `shared`, etc.

The package overlaps another package of a similar name. In this case either you should review your design, or consider merging the packages, or renaming the conflicting packages to make their purpose more specific. Consdider the `io/ioutil` and `net/http/httputil` packages as weak supporting evidence.

4.3. Avoid package names like `base`, `common`, or `util`

A common cause of poor package names is what I call *utility packages*. These are packages where helpers and utility code congeals over time. As these packages contain an assortment of unrelated functions, their utility is hard to describe in terms of what the package provides. This often leads to the package's name being derived from what the package *contains*--utilities.

Package names like `utils` or `helpers` are commonly found in larger projects which have developed deep package hierarchies and want to share helper functions without encountering import loops. By extracting utility functions to new package the import loop is broken, but because the package stems from a design problem in the project its name doesn't reflect its purpose, only its function of breaking the import cycle.

My recommendation to improve the name of `utils` or `helpers` packages is to analyse where they are called and if possible move the relevant functions into their caller's package. Even if this involves duplicating some helper code this is better than introducing an import dependency between two packages.

> [A little] duplication is far cheaper than the wrong abstraction.
>
> — Sandy Metz

In the case where utility functions are used in many places prefer multiple packages, each focused on a single aspect, to a single monolithic package.

Tip	Use plurals for naming utility packages. For example the `strings` for string handling utilities.

Packages with names like `base` or `common` are often found when functionality common to two or more implementations, or common types for a client and server, has been refactored into a separate package. Their

names also represent design holdovers from languages like Java and C++ where the relationship between packages followed similar rules to those of inheretence. I believe the solution to packeges like `base` or `common` is to reduce the number of packages, combine the client, server, and common code into a single package named after the behaviour delivered from the previously fractured packages.

For example, the `net/http` package does not have `client` and `server` sub packages, instead it has a `client.go` and `server.go` file, each holding their respective types, and a `transport.go` file for the common message transport code.

Tip	A public identifier includes its package name. It's important to remember that the name of an identifier includes the name of its package. • The `Get` function from the `net/http` package becomes `http.Get` when referenced by another package. • The `Reader` type from the `strings` package becomes `strings.Reader` when imported into other packages. • The `Error` interface from the `net` package is clearly related to network errors.

4.4. Return early rather than nesting deeply

As Go does not use exceptions for control flow there is no requirement to deeply indent your code just to provide a top level structure for the `try` and `catch` blocks. Rather than the successful path nesting deeper and deeper to the right, Go code is written in a style where the success path continues down the screen as the function progresses. My friend Mat Ryer calls this practice 'line of sight' coding. [5]

This is achieved by using *guard clauses*; conditional blocks with assert preconditions upon entering a function. Here is an example from the `bytes` package,

```
func (b *Buffer) UnreadRune() error {
        if b.lastRead <= opInvalid {
                return errors.New("bytes.Buffer: UnreadRune: previous
operation was not a successful ReadRune")
        }
        if b.off >= int(b.lastRead) {
                b.off -= int(b.lastRead)
        }
        b.lastRead = opInvalid
        return nil
}
```

Upon entering `UnreadRune` the state of `b.lastRead` is checked and if the previous operation was not `ReadRune` an error is returned immediately. From there the rest of the function proceeds with the assertion that

`b.lastRead` is greater that `opInvalid`.

Compare this to the same function written without a guard clause,

```
func (b *Buffer) UnreadRune() error {
        if b.lastRead > opInvalid {
                if b.off >= int(b.lastRead) {
                        b.off -= int(b.lastRead)
                }
                b.lastRead = opInvalid
                return nil
        }
        return errors.New("bytes.Buffer: UnreadRune: previous operation
was not a successful ReadRune")
}
```

The body of the successful case, the most common, is nested inside the first `if` condition and the successful exit condition, `return nil`, has to be discovered by careful matching of *closing* braces. The final line of the function now returns an error, and the reader must trace the execution of the function back to the matching *opening* brace to know when control will reach this point.

This is more error prone for the reader, and the maintenance programmer, hence why Go prefer to use guard clauses and returning early on errors.

4.5. Make the zero value useful

Every variable declaration, assuming no explicit initialiser is provided, will be automatically initialised to a value that matches the contents of zeroed memory. This is the value's *zero value*. The type of the value determines it's zero value; for numeric types it is zero, for string types it is `""`, for pointer types nil, the same for slices, maps, and channels.

This property of always setting a value to a known default is important for safety and correctness of your program and can make your Go programs simpler and more compact. This is what Go programmers talk about when they say "give your structs a useful zero value".

Consider the `sync.Mutex` type. `sync.Mutex` contains two unexported integer fields, representing the mutex's internal state. Thanks to the zero value those fields will be set to will be set to 0 whenever a `sync.Mutex` is declared. `sync.Mutex` has been deliberately coded to take advantage of this property, making the type usable without explicit initialisation.

```
type MyInt struct {
        mu  sync.Mutex
```

```
        val int
}

func main() {
        var i MyInt

        // i.mu is usable without explicit initialisation.
        i.mu.Lock()
        i.val++
        i.mu.Unlock()
}
```

Another example of a type with a useful zero value is `bytes.Buffer`. You can declare a `bytes.Buffer` and start writing to it without explicit initialisation.

```
func main() {
        var b bytes.Buffer
        b.WriteString("Hello, world!\n")
        io.Copy(os.Stdout, &b)
}
```

A useful property of slices is their zero value is `nil`. This makes sense if we look at the runtime's (pseudo) definition of a slice header.

```
type slice struct {
        array *[...]T // pointer to the underlying array
        len   int
        cap   int
}
```

The zero value of this struct would imply `len` and `cap` have the value `0`, and `array`, the pointer to memory holding the contents of the slice's backing array, would be `nil`. This means unless you need to specify a size you don't need to explicitly `make` a slice, you can just declare it.

```
func main() {
        // s := make([]string, 0)
        // s := []string{}
        var s []string

        s = append(s, "Hello")
```

```
        s = append(s, "world")
        fmt.Println(strings.Join(s, " "))
}
```

Note	`var s []string` is similar to the two commented lines above it, but not identical. It is possible to detect the difference between a slice value that is `nil` and a slice value that has zero length.
	`func main() {`
	`  var s1 = []string{}`
	`  var s2 []string`
	`  fmt.Println(reflect.DeepEqual(s1, s2)) // false`
	`}`

A useful, albeit surprising, property of uninitialised pointer variables—nil pointers—is you can call methods on types that have a nil value. This can be used to provide default values simply.

```
type Config struct {
        path string
}

func (c *Config) Path() string {
        if c == nil {
                return "/usr/home"
        }
        return c.path
}

func main() {
        var c1 *Config
        var c2 = &Config{
                path: "/export",
        }
        fmt.Println(c1.Path(), c2.Path())
}
```

4.6. Avoid package level state

The key to writing maintainable programs is that they should be loosely coupled. A change to one package should have a low probability of affecting another.

There are two excellent ways to achieve loose coupling in Go:

Use interfaces to describe the behaviour your functions or methods require.

Avoid the use of global state.

In Go we can declare variables at the block, function, or method scope, and also at the package scope. When the variable is public, given a identifier starting with a capital letter, then its scope is effectively global to the entire program—any package may observe the type and contents of that variable *at any time*.

Mutable global state introduces tight coupling between independent parts of your program as global variables become an invisible parameter to every function in your program! Any function that relies on a global variable can be broken if that variable's type changes. Any function that relies on the state of a global variable can be broken if another part of the program changes that variable.

If you want to reduce the coupling a global variable creates,

Move the relevant variables as fields on structs that need them.

Use interfaces to reduce the coupling between the behaviour and the implementation of that behaviour.

5. Project Structure

Let's talk about combining packages together into a project. Commonly this will be a single git repository. In the future Go developers will use the terms *module* and *project* interchangeably.

Just like a package, each project should have a clear purpose. If your project is a library, it should provide one thing, say XML parsing, or logging. You should avoid combining multiple purposes into a single project, this will help avoid the dreaded `common` library.

Tip	In my experience, the `common` repo ends up tightly coupled to its biggest consumer and that makes it hard to back-port fixes without upgrading both common and consumer in lock step, bringing in a lot of unrelated changes and API breakage along the way.

5.1. Consider fewer, larger packages

One of the things I tend to pick up in code review for programmers who are transitioning from other languages to Go is they tend to overuse packages. To be fair, mastery of Go is effectively the art of moderation in the use of all Go's features, but package overuse seems to be one of the most common misstep. I suspect driven by programmers' innate drive to categorise and neatly organise the things they see.

Go does not provide elaborate ways of establishing visibility. Go lacks Java's `public`, `protected`, `private`, and implicit `default` access modifiers. There is no equivalent of C++'s notion of a `friend` classes.

In Go we have only two access modifiers, public and private, the former indicated by the capitalisation of the first letter of the identifier. If an identifier is public, it's name starts with a capital letter, that identifier can be referenced by *any* other Go package.

Note	You may hear people say *exported* and *not exported* as synonyms for public and private.

Given the limited controls available to control access to a package's symbols, what practices should Go programmers follow to avoid creating over-complicated package hierarchies?

The advice I find myself repeating is to prefer fewer, larger packages. Your default position should be to not create a new package. That will lead to too many types being made public creating a wide, shallow, API surface for your package..

The sections below explores this suggestion in more detail.

Tip	Every package, with the exception of `cmd/` and `internal/`, should contain some source code.

Note	Possibly because of the early use of a `pkg/` directory to hold package—and the corresponding `cmd/` directory to hold commands (`package main`) this practice of putting your packages in an empty `pkg/` directory has spread to other Go projects. This practice was never a recommendation, just a result of the original `Makefile` based build system. In September 2014, the stdlib moved away from storing package code in an otherwise empty `pkg/` directory, and you should follow their lead. Other than a superficial symetary with `cmd/` putting packages in a `pkg/` directory is needless boilerplate and distracts from the potentially more useful `internal/` directory.

Tip	Coming from Java? If you're coming from a Java or C# background, consider this guideline. • A Java package is equivalent to a single `.go` source file. • A Go package is equivalent to a whole Maven module or .NET assembly.

5.1.1. Arrange code into files by import statements

If you're arranging your packages by what they provide to callers, should you do the same for files within a Go package? How do you know when you should break up a `.go` file into multiple ones? How do you know when you've gone to far and should instead consolidate several `.go` files together?

Here are the guidelines I use:

Start each package with one `.go` file. Give that file the same name as the name of the folder. For example the source for `package http` should be placed in a file called `http.go` in a directory named `http`.

As your package grows you may decide to split apart the various *responsibilities* into different files. eg, `messages.go` contains `the `Request` and `Response` types, `client.go` contains the `Client` type, `server.go` contains the `Server` type.

If you find your files have similar `import` declarations, consider combining them. Alternatively, identify the

differences between the import sets and move those types/functions/methods into their own file.

Different files should be responsible for different areas of the package. `messages.go` may be responsible for marshalling of HTTP requests and responses on and off the network, `http.go` may contain the low level network handling logic, `client.go` and `server.go` implement the HTTP business logic of request construction or routing, and so on.

Tip	Prefer nouns for source file names. They are containers for source code after all.

Note	The Go compiler compiles each package in parallel. Within a package the compiler compiles each *function* (methods are just fancy functions in Go) in parallel. Changing the layout of your code within a package should not affect compilation time.

Tip	Avoid elaborate package hierarchies With one exception, which we'll talk about next, the hierarchical directory structure a Go project has no meaning to the `go` tool. For example, the `net/http` package is *not* a child or sub-package of the `net` package. Go eschewed elaborate type hierarchy. This is generally considered to be a good thing. Don't make the mistake or replacing that with an elaborate package hierarchy. If you find you have created intermediate directories in your project which contain no `.go` files, you may have fallen afoul of the desire to create a taxonomy of your source code.

5.1.2. Use `internal` packages to reduce your public API surface

If your project contains multiple packages you may find you have some exported functions which are intended to be used by other packages in your project, but are not intended to be part of your project's public API. If you find yourself in this situation the `go` tool recognises a special folder name—not package name--`internal/` which can be used to indicate code which is public to your project, but private to others.

To create an internal package, place it within a directory named `internal/` or in a sub-directory of a directory named `internal/`. When the `go` command sees an import of a package with `internal` in its path, it verifies that the importing package is within the tree rooted at the *parent* of the `internal` directory.

For example, a package `.../a/b/c/internal/d/e/f` can be imported only by code in the directory tree rooted at `.../a/b/c`. It cannot be imported by code in `.../a/b/g` or in any other repository. [6]

5.2. Keep package main as small as possible

Your `main` function, and `main` package should do as little as possible. This is because `main.main` acts as a singleton; there can only be one `main` function in a program.

Because `main.main` is a singleton there are a lot of assumptions built into the things that `main.main` will

call, that they will only be called during `main.main` or `main.init`, and only called *once*. This makes it hard to write tests for code written in `main.main`. Main packages often invoke singletons, parse command line flags, expect files to be on disk in a certain place, and never expect to be executed concurrently. You can't even reference `main.main` from a test.

Thus you should aim to move as much of your business logic out of your main function and ideally out of your main package. `func main()` should parse flags, open connections to databases, loggers, and such, then hand off execution to a high level object.

6. API Design

The last piece of design advice I'm going to give today, I feel, is the most important.

All of the suggestions I've made so far are just that, suggestions. These are the way I try to write my Go, but I'm not going to push them too hard in code review.

However when it comes to reviewing APIs during code review, I am less forgiving. This is because everything I've talked about so far can be fixed without breaking backward compatibility; they are, for the most part, implementation details.

When it comes to the public API of a package, it pays to put considerable thought into the initial design, because changing that design later is going to be disruptive for people who are already using your API. Changing your public API forces the existing user base to have to dedicate engineering resources to upgrading across your API break. The larger the break, the more likely this task will be considered *low impact*, but *high risk*, and likely to be pushed off in light of other business priorities.

6.1. Design APIs that are hard to misuse.

> APIs should be easy to use and hard to misuse.
>
> — Josh Bloch [3]

If you take anything away from this section, it should be this advice from Josh Bloch. If an API is hard to use for simple things, then every invocation will look complicated. When the actual invocation of the API *is* complicated it will be less obvious and more likely to be overlooked.

6.1.1. Be wary of functions which take several parameters of the same type

A good example of a simple looking, but hard to use correctly, API is one which takes two or more parameters of the same type. Let's compare two function signatures:

```
func Max(a, b int) int
func CopyFile(to, from string) error
```

What's the difference between these two functions? Obviously one returns the maximum of two numbers, the other copies a file, but that's not the important thing.

```
Max(8, 10) // 10
Max(10, 8) // 10
```

Max is *commutative*; the order of its parameters does not matter. The maximum of eight and ten is ten regardless of if I compare eight and ten or ten and eight.

However, this property does not hold true for `CopyFile`.

```
CopyFile("/tmp/backup", "presentation.md")
CopyFile("presentation.md", "/tmp/backup")
```

Which one of these statements made a backup of your presentation and which one overwrite your presentation with last week's version? You can't tell without consulting the documentation. A code reviewer cannot know if you've got the order correct without consulting the documentation.

The general advice is to try to avoid this situation. Just like long parameter lists, indistinct parameter lists are a design smell.

However, a possible solution to this class of problem is to introduce a helper type which will be responsible for calling `CopyFile` correctly.

```
type Source string

func (src Source) CopyTo(dest string) error {
        return CopyFile(dest, string(src))
}

func main() {
        var from Source = "presentation.md"
        from.CopyTo("/tmp/backup")
}
```

In this way `CopyFile` is always called correctly and, given its poor API can possibly be made private, further reducing the likelihood of misuse.

Tip	APIs with multiple parameters of the same type are hard to use correctly.

6.2. Design APIs for their default use case

A few years ago I gave a talk [7] about using functional options [8] to make APIs easier to use for their default

case.

The gist of this talk was you should design your APIs for the common, or default, use case. Said another way, your API *should not* require the caller to provide parameters which they don't care about. More than being hard to use, you place the user in the position of *guessing* reasonable values. If they are lucky, the values they YOLO'd have no impact. That's if they are lucky.

6.2.1. Discourage the use of `nil` as a parameter

I opened this chapter with the suggestion that you shouldn't force the caller of your API into providing you parameters when they don't really care what those parameters mean. This is what I mean when I say *design APIs for their default use case*.

Here's an example from the `net/http` package.

Note	I pick on the `net/http` package a lot. I don't mean to imply it, or the engineers who contributed to it, are bad. On the contrary, `net/http` has been tremendously successful and with that success has come a process of extension via accretion which makes it a great candidate for case studies.

```
package http

// ListenAndServe listens on the TCP network address addr and then calls
// Serve with handler to handle requests on incoming connections.
// Accepted connections are configured to enable TCP keep-alives.
//
// The handler is typically nil, in which case the DefaultServeMux is
used.
//
// ListenAndServe always returns a non-nil error.
func ListenAndServe(addr string, handler Handler) error {
```

`ListenAndServe` takes two parameters, a TCP address to listen for incoming connections, and `http.Handler` to handle the incoming HTTP request. `Serve` allows the second parameter to be `nil`, and notes that usually the caller *will* pass `nil` indicating that they want to use `http.DefaultServeMux` as the implicit parameter.

Now the caller of `Serve` has two ways to do the same thing.

```
http.ListenAndServe("0.0.0.0:8080", nil)
http.ListenAndServe("0.0.0.0:8080", http.DefaultServeMux)
```

Both do exactly the same thing.

This `nil` behaviour is viral. The `http` package also has a `http.Serve` helper, which you can reasonably imagine that `ListenAndServe` builds upon like this

```
func ListenAndServe(addr string, handler Handler) error {
        l, err := net.Listen("tcp", addr)
        if err != nil {
                return err
        }
        defer l.Close()
        return Serve(l, handler)
}
```

Because `ListenAndServe` permits the caller to pass `nil` for the second parameter, `http.Serve` also supports this behaviour. In fact, `http.Serve` is the one that implements the "if `handler` is `nil`, use `DefaultServeMux`" logic. Accepting `nil` for one parameter may lead the caller into thinking they can pass `nil` for both parameters. However calling `Serve` like this,

results in an ugly panic.

Tip	Don't mix `nil` and non `nil`-able parameters in the same function signature.

The author of `http.ListenAndServe` was trying to make the API user's life easier in the common case, but possibly made the package harder to use safely.

There is no difference in line count between using `DefaultServeMux` explicitly, or implicitly via `nil`.

```
        const root = http.Dir("/htdocs")
        http.Handle("/", http.FileServer(root))
        http.ListenAndServe("0.0.0.0:8080", nil)
```

verses

```
        const root = http.Dir("/htdocs")
        http.Handle("/", http.FileServer(root))
        http.ListenAndServe("0.0.0.0:8080", http.DefaultServeMux)
```

and a was this confusion really worth saving one line?

```
        const root = http.Dir("/htdocs")
        mux := http.NewServeMux()
        mux.Handle("/", http.FileServer(root))
        http.ListenAndServe("0.0.0.0:8080", mux)
```

Tip	Give serious consideration to how much time helper functions will save the programmer. Clear is better than concise.

Tip	Avoid public APIs with test only parameters Avoid exposing APIs with values which only differ in test scope. Instead, use Public wrappers to hide those parameters, use test scoped helpers to set the property in test scope.

6.2.2. Prefer var args to []T parameters

It’s very common to write a function or method that takes a slice of values.

```
func ShutdownVMs(ids []string) error
```

This is just an example I made up, but its common to a lot of code I’ve worked on. The problem with signatures like these is they presume that they will be called with more than one entry. However, what I have found is many times these type of functions are called with only one argument, which has to be "boxed" inside a slice just to meet the requirements of the functions signature.

Additionally, because the `ids` parameter is a slice, you can pass an empty slice or `nil` to the function and the compiler will be happy. This adds extra testing load because you *should* cover these cases in your testing.

To give an example of this class of API, recently I was refactoring a piece of logic that required me to set some extra fields if at least one of a set of parameters was non zero. The logic looked like this:

```
if svc.MaxConnections > 0 || svc.MaxPendingRequests > 0 || svc.MaxRequests
> 0 || svc.MaxRetries > 0 {
        // apply the non zero parameters
}
```

As the `if` statement was getting very long I wanted to pull the logic of the check out into its own function. This is what I came up with:

```
// anyPostive indicates if any value is greater than zero.
func anyPositive(values ...int) bool {
        for _, v := range values {
                if v > 0 {
                        return true
                }
        }
        return false
}
```

This enabled me to make the condition where the inner block will be executed clear to the reader:

```
if anyPositive(svc.MaxConnections, svc.MaxPendingRequests,
svc.MaxRequests, svc.MaxRetries) {
        // apply the non zero parameters
}
```

However there is a problem with `anyPositive`, someone could accidentally invoke it like this

In this case `anyPositive` would return `false` because it would execute zero iterations and immediately return `false`. This isn't the worst thing in the world — that would be if `anyPositive` returned `true` when passed no arguments.

Nevertheless it would be be better if we could change the signature of `anyPositive` to enforce that the caller should pass at least one argument. We can do that by combining normal and vararg parameters like this:

```
// anyPostive indicates if any value is greater than zero.
func anyPositive(first int, rest ...int) bool {
        if first > 0 {
                return true
        }
        for _, v := range rest {
                if v > 0 {
                        return true
                }
        }
        return false
}
```

Now `anyPositive` cannot be called with less than one argument.

6.3. Let functions define the behaviour they require

Let's say I've been given a task to write a method that persists a Document structure to disk.

```
type Document struct {
        // mo' state
}

// Save writes the contents of the Document to the file f.
func (d *Document) Save(f *os.File) error
```

I could specify this method, `Save`, which takes an `*os.File` as the destination to write the `Document`. But this has a few problems.

The signature of `Save` precludes the option to write the data to a network location. Assuming that in the new world of lambda functions and microservices, network storage is likely to become requirement, the signature of this function would have to change, impacting all its callers.

`Save` is also unpleasant to test, because it operates directly with files on disk. To verify its operation the test would have to read the contents of the file after being written. You would also have to ensure that `f` was written to a temporary location and always removed afterwards.

Moreover `*os.File` defines a lot of methods which are not relevant to `Save`, like reading directories and checking to see if a path is a symlink. It would be useful if the signature of `Save` could describe only the parts of `*os.File` that were relevant.

```
// Save writes the contents of d to the supplied ReadWriterCloser.
func (d *Document) Save(rwc io.ReadWriteCloser) error
```

Using `io.ReadWriteCloser` we can apply the interface segregation principle to redefine `Save` to take an interface that describes more general file shaped things. With this change, any type that implements the `io.ReadWriteCloser` interface can be substituted for the previous `*os.File`. This makes `Save` both broader in its application, and clarifies to the caller of `Save` which methods of the `*os.File` type are relevant to its operation. As the author of `Save` I no longer have the option to call those unrelated methods on `*os.File` as it is hidden behind the `io.ReadWriteCloser` interface. But we can take the interface segregation principle a bit further.

Firstly, it is unlikely that if `Save` follows the single responsibility principle, it will read the file it just wrote to verify its contents—that should be responsibility of another piece of code.

```
// Save writes the contents of d to the supplied WriteCloser.
func (d *Document) Save(wc io.WriteCloser) error
```

We can narrow the specification for the interface we pass to Save to just writing and closing.

Secondly, by providing `Save` with a mechanism to close its stream, which we inherited in this desire to make it still look like a file, this raises the question of under what circumstances will `wc` be closed. Possibly `Save` will call `Close` unconditionally, or perhaps `Close` will be called in the case of success. Neither of these is a good option. Unconditionally closing `wc` after the call to `Save` precludes the caller from writing additional data after the document is written. Conditionally closing the `WriteCloser` — it doesn't matter if its on success, or failure—means the caller must grow intricate knowledge of the operation of `Save`.

```
// Save writes the contents of d to the supplied Writer.
func (d *Document) Save(w io.Writer) error
```

A better solution would be to redefine `Save` to take only an `io.Writer`, stripping it completely of the responsibility to do anything but write data to a stream.

By applying the interface segregation principle to our `Save` function, the results has simultaneously been a function which is the most specific in terms of its requirements—it only needs a thing that is writable—and the most general in its function, we can now use `Save` to save our data to anything which implements `io.Writer`.

As a side effect it is clear that the name of the method is no longer accurate. A better name may be

```
func (d *Document) WriteTo(w io.Writer) error
```

6.4. Understanding `nil`

`nil` is a curious beast.

`nil` is a constant, as I'll show in a minute, but `nil` cannot be assigned to a named constant. `nil` can be assigned to a value, and values can be compared to `nil`, but `nil` cannot be compared to itself.

```
package main

import "fmt"

const T = nil != nil // (1)

func main() {
        fmt.Println(nil == nil) // (2)

        if nil == new(int) {
                fmt.Println("hmm") // (3)
        }
}
```

`nil` cannot be compared with itself for inequality

not with itself for equality

however `nil` may appear on either side of a binary operation

So, given all these restrictions, `nil` sounds out of place in the orthogonal Go world, why would such a concept exist? The answer is, while `nil` may appear inconsistent, it makes a lot of other interactions in Go simpler.

If you assign `nil` to a pointer the pointer will not point to anything.

If you assign `nil` to an interface, the interface will not contain anything.

If you assign `nil` to a slice and the slice will have zero len and zero cap and no longer point an underlying array.

`nil` 's meaning, or it's type, is fully determined by the static type of the variable it's assigned to. When you write a statement like

```
var f *os.File
if f == nil {
    // ...
}
```

The rule of expressions dictates that all the variables in the expression must have the same type. We know the type of `f`, it is a `*os.File`, therefore we know that `nil` has been coerced from an ideal constant to an expression which evaluates to the value also of type `*os.File`.

Here is a more complicated example

```
var s []string
if s == nil {
    // ...
}
```

Again, the type of `s` is known, and as there are no conversions in the expression, the type of `nil` on the other side of the comparison must be the same, `[]string`.

Note	`nil` has a strong relationship with the zero value.

Tip	Be wary of `nil` and interfaces
	
	As we saw above `nil` can be simple, or complicated, depending on how you reason about it. One area where `nil` is complicated, until you memorise the rule is the dreaded typed `nil`.
	
	`func Open(path string) io.Writer {`
	`  var f *os.File`
	`  f, _ = os.Open(path)`
	`  return f`
	`}`
	
	`func main() {`
	`  f := Open("/missing")`
	`  fmt.Println(f == nil) // (1)`
	`}`
	
	1. prints false because the returned value has a *type* of `os.Writer` not `nil`.
	
	If your method returns an *interface type*, be sure to always `return nil` explicitly. Assigning nil to a value of a concrete type and returning that will convert it to a typed `nil`

7. Error handling

Error handling is important for reliable programs. Error handling is as important as the rest of your code. Error handling is as important as checking a loop index for the exit condition, or checking the result of a shift operation, or testing the result of a multiplication is within the expected bounds, that's how fundamental error handling is to Go. And, just like shifting or comparisons or multiplication, error handling is a first class responsibility of all Go programmers. So important that Go makes it a first class citizen. Because, you have to plan for failure.

When you write to a network, assume the other side never gets the request. When you write to a channel, assume the other side never picks up the write.

The `error` interface is the key to Go's composable error handling story.

Tip	Try / catch are not sufficient • Exceptions obfuscate control flow just as badly as return codes. • Reliable software cannot be written with unchecked exceptions, yet checked exceptions have never been repeated by any other language other than java.

7.1. Errors are just values

This statement is almost universal in the Go programmer's phrase book, but what do Go programmers mean when they say "errors are just values", and what does this technique imply? By way of explanation, consider the counter example of `panic` and `recover`, often mistaken for exceptions.

`panic` and `recover`, two keywords added to the language for a single purpose. `recover` can only be used for one purpose; to access a value previously passed to `panic`. If that wasn't enough `recover's use case is so specific, it can only be used inside a `defer` block. You cannot use `recover` for any other purpose, it can only be used in concert with `panic`.

This pair of features sit by themselves in a corner of the language. How's that for non orthogonal?

By contrast, error values are not limited to the rarefied semantics of `panic` and `recover`.

7.2. Errors should be opaque

> With a sufficient number of users of an API, it does not matter what you promise in the contract, all observable behaviours of your system will be depended on by somebody.
>
> — Hyram's Law [9]

— Joe Tsai
GopherCon 2017

Programmers will rely on whatever behaviour, guaranteed or not, they observe from your API. Simply put, the more observable state your API returns, the larger the yoke of backwards compatibility you are implicitly committing to.

To the caller, the type and contents of an error value, if not `nil`, should be considered opaque. To do otherwise introduces brittle coupling between the function and its caller.

The exception to this rule are are sentinel values like `io.EOF`. These are however, the exception to the rule, not a pattern to be emulated.

Tip	Almost all errors terminate processing. Almost all errors are opaque. Design APIs that take advantage of these properties.

Note	Issue 12866 is an example of an unintended consequence of overspecifying the error value returned.

7.3. Assert errors for behaviour, not type

The common contract for functions which return a value of the interface type `error`, is the caller should not presume anything about the state of the other values returned from that call without first checking the error. In the majority of cases, error values returned from functions should be opaque to the caller. That is to say, a test that error is `nil` indicates if the call succeeded or failed, and that's all there is to it.

The methodology I follow is; if a function can return an error, you cannot make any assumptions about the state of any other values returned until you check the error. If it was found that the error was set (ie, not `nil`), then the state of those other values is *unknown*.

A small number of cases, require that the caller investigate the nature of the error to decide if it is reasonable to retry the operation. A common request for package authors is to return errors of a known public type, so the caller can type assert and inspect them. I believe this practice leads to a number of undesirable outcomes:

Public error types increase the surface area of the package's API.

New implementations must only return types specified in the interface's declaration, even if they are a poor fit. This also introduces coupling. My implementation must import the package that declares the specific error type required.

The error type cannot be changed or deprecated after introduction without breaking compatibility, making for a brittle API.

You should feel no more comfortable asserting an error is a particular type than they would be asserting the `string` returned from `Error()` matches a particular pattern.

Instead I present a suggestion that permits package authors and consumers to communicate about their intention, without having to overly couple their implementation to the caller. This suggestion fits the *has a*

[behaviour] nature of Go's implicit interfaces, rather than the *is a* [subtype of] nature of inheritance based languages. Consider this example:

```
func isTimeout(err error) bool {
        type timeout interface {
                Timeout() bool
        }
        te, ok := err.(timeout)
        return ok && te.Timeout()
}
```

The caller can use `isTimeout` to determine if the error is related to a timeout, and if so confirm if the error was timeout related, all without knowing anything about the type, or the original source of the `error` value.

Gift wrapping errors, usually by libraries that annotate the error path, is enabled by this method; providing that the wrapped error types also implement the interfaces of the error they wrap. This may seem like a generally intractable problem, but in practice there are relatively few interface methods that are in common use, so `Timeout() bool` and `Temporary() bool` cover a large set of use cases.

For package authors, if your package generates errors of a temporary nature, ensure you return error types that implement the respective interface methods. If you wrap error values on the way out, ensure that your wrappers respect the interface(s) that the underlying error value implemented.

For package users, *if* you need to inspect an error—and hopefully this should be infrequent—declare and assert an interface to assert the behaviour you expect, not the error's type. Don't ask package authors for public error types; instead ask that they make their types conform to common interfaces as appropriate.

Tip	Don't assert an error value is a specific type, but rather assert that the value implements a particular behaviour.

7.4. Never use nil to indicate failure

When Go programmers discover that you can call a method on nil receiver it generally blows their mind.

```
type Bar struct{}

func (b *Bar) Whoa() {
        fmt.Println("whaaaaaat?")
}

func main() {
        var b *Bar // (1)
```

```
        b.Whoa()   // (2)
}
```

b is of type *Bar but is nil.

Prints "whaaaaaat".

This is because a method in Go is just syntactic sugar for a function who's first parameter is the receiver.

```
func Whoa(b *Bar) {
        fmt.Println("whaaaaaat?")
}
```

Restated like this it is clear to see why passing a nil value for b is uneventful. However, a problem arises when the code attempts to access b or one of it's fields.

```
type Bar struct {
        message string
}

func (b *Bar) Whoops() {
        fmt.Println(b.message) // (1)
}

func main() {
        var b *Bar
        b.Whoops() // (2)
}
```

panics occurs here

not here

Faced with this realisation Go programmers are grippe with fear that someone could call their code's methods on an accidental nil method. Their usual reaction is a creeping panic that they will have to pepper their code with nil checks like this protect against this scenario.

```
func (b *Bar) Whoops() {
        if b != nil { // (1)
                fmt.Println(b.message)
        }
}
```

only execute the *body* of the method if `b` is not nil

The solution is to realise that the check for a `nil` receiver *before* attempting the call is in the right place.

```
func main() {
        var b *Bar
        if b != nil {
                b.Whoops() // (1)
        }
}
```

only call the method if `b` is not nil

Rather than checking inside the method when it is too late, the check should be executed by the caller. But this is seen as unsatisfactory because it force the check to *every* call site, rather than in one place, the receiver.

For arguments sake let's explore the options to effectively handle a nil receiver *inside the method*. What are the options for an author to handle this situation?

7.4.1. Panic

Given that calling a method on a `nil` receiver, expect where the method was written to explicitly handle this behaviour (there are types in the stdlib that do this, but not many), is an unrecoverable programming error, a reasonable response would be to panic.

```
func (b *Bar) Whoops() {
        if b == nil {
                panic("b is nil") // (1)
        }
        fmt.Println(b.message)
}
```

But given that dereferencing `b` to access the `message` field is going to panic anyway, apart from having control over the panic message, this seems to add little other than boilerplate.

7.4.2. Return an error to the caller

The next option to the reporting problem may be treat this coding error like any other non fatal error and return an `error` value.

```
func (b *Bar) GetMessage() (string, error) {
        if b == nil {
```

```
                return "", errors.New("b is nil") // (1)
        }
        return message, nil
}
```

return a descriptive error to the caller.

This has serious implications for the caller of *any* method.

Every method will have to return an error. Every. Method.

Every caller will have to check the error *after* a call to *any* method.

Every interface you define will have to include an error parameter so that an implementation can report it was called on a nil receiver.

Every interface you implement will have to proved you with an error return parameter.

7.4.3. Elide execution

We saw that option above, `Whoops` would print nothing if it was called on a `nil` receiver. This is perhaps the worst choice as now the operation will silently do nothing. Imagine trying to debug a complex failure in your application because some logic did not fire because it was passed a `nil` receiver?

Given there is no reasonable way for the method executed on to protected against this the reasonably response is to not worry about it. After all a `nil` receiver is a symptom of a bug that happened elsewhere in your code. The most likely cause was a failure to check the error from a previous call. That is the place where you should spend your efforts, not defensively trying to code around a failure to follow proper error handling.

Equally you should not intentionally let `nil` flow our of your API, except in the `error` value. In other languages, especially those that don't support multiple return values, it is extremely common to return a `nil` like value a failure happens inside the method.

On one hand this is eminently sensible, exceptions are overused, and for most failures they are hardly unexpected, so some mechanism of representing a failure that doesn't warrant the four alarm fire of an exception is called for. Obviously this has some major downsides. As the flow of execution is not redirected a catch block this `nil` (or `null`, not naming any names) sentinel value now represent a silent failure condition.

Tip	Don't check for a `nil` receiver, employ high test coverage and vet/lint tools to spot unhandled error conditions resulting in `nil` receivers.

7.5. Don't panic

Go's error handling strategy is via the `error` interface and returning `error` values. Go does have `panic`,

which is a by-product of the counterpart in the runtime's internal `throw` function. There are few cases of using `recover` that I know of, and all of those are used to simulate non local transfer of control *not* exception handling. Using recover has all the problems of sensing errors by type, with the added complication that the set of types returned is unbounded.

While it is true that any Go function can call `panic`, any Go procedure can fail due to out of memory, the program can be killed by a process manager, or the serve can simply fail. Always write your programs to assume failure, not success. Avoid `panic` and eschew `recover`, they're not the tool you are looking for.

7.5.1. Avoid selfish panics

If a function or method returns an error value, there is no call for a `panic`. `panic` must be truly the last resort; exiting on impossible conditions, or in scenarios where the applications truly cannot recover. Panicing in a library must be the absolute last resort. Not only does it have direct impact on the reliability of the program your code is embedded into, but engenders a belief that your library is hard to work with, or itself unreliable.

Panic will, during unwinding the stack, execute any deferred statements. However just as a panic in one goroutine cannot be recovered in another, a panic in one goroutine will not allow defer statements in other goroutines to exit. For a goroutine spawned by a library to panic the entire program is selfish and *must* be avoided.

Tip	The common party line is panics, if used, should not leak beyond the API boundary. I would strengthen this statement by simply saying, do not use panic in library code.

7.5.2. Avoid log.Fatal

The `log` package provides two ways to exit your program, `log.Fatal` and `log.Panic`. These are effectively the same as panic, and the same rules for panic should apply. They were a mistake and should not have been added. The convenience of being able to log and crash the program in one line, not two, created a misleading precident.

7.6. Eliminate error handling by eliminating errors

Some of what I've discussed in this chapter will be obsoluted by the Go 2 changes to error handling I talked about the draft proposals for improving error handling. But do you know what is better than an improved syntax for handling errors? Not needing to handle errors at all.

Note	I'm not saying "remove your error handling". What I am suggesting is, change your code so you do not have errors to handle.

This section draws inspiration from John Ousterhout's recently book, A philosophy of Software Design [10].

One of the chapters in that book is called "Define Errors Out of Existence". We're going to try to apply this advice to Go.

7.6.1. Counting lines

Let's write a function to count the number of lines in a file.

```
func CountLines(r io.Reader) (int, error) {
        var (
                br    = bufio.NewReader(r)
                lines int
                err   error
        )

        for {
                _, err = br.ReadString('\n')
                lines++
                if err != nil {
                        break
                }
        }

        if err != io.EOF {
                return 0, err
        }
        return lines, nil
}
```

Because we're following our advice from previous sections, `CountLines` takes an `io.Reader`, not a `*os.File`; its the job of the caller to provide the `io.Reader` who's contents we want to count.

We construct a `bufio.Reader`, and then sit in a loop calling the `ReadString` method, incrementing a counter until we reach the end of the file, then we return the number of lines read.

At least that's the code we want to write, but instead this function is made more complicated by error handling. For example, there is this strange construction,

```
                _, err = br.ReadString('\n')
                lines++
                if err != nil {
```

```
                        break
                }
```

We increment the count of lines *before* checking the error—that looks odd.

The reason we have to write it this way is `ReadString` will return an error if it encounters and end-of-file before hitting a newline character. This can happen if there is no final newline in the file.

To try to fix this, we rearrange the logic to increment the line count, then see if we need to exit the loop.

Note	this logic still isn't perfect, can you spot the bug?

But we're not done checking errors yet. `ReadString` will return `io.EOF` when it hits the end of the file. This is expected, `ReadString` needs some way of saying *stop, there is nothing more to read*. So before we return the error to the caller of `CountLine`, we need to check if the error was *not* `io.EOF`, and in that case propagate it up, otherwise we return `nil` to say that everything worked fine.

I think this is a good example of Russ Cox's observation that error handling can obscure the operation of the function. Let's look at an improved version.

```
func CountLines(r io.Reader) (int, error) {
        sc := bufio.NewScanner(r)
        lines := 0

        for sc.Scan() {
                lines++
        }
        return lines, sc.Err()
}
```

This improved version switches from using `bufio.Reader` to `bufio.Scanner`.

Under the hood `bufio.Scanner` uses `bufio.Reader`, but it adds a nice layer of abstraction which helps remove the error handling with obscured the operation of `CountLines`.

Note	`bufio.Scanner` can scan for any pattern, but by default it looks for newlines.

The method, `sc.Scan()` returns `true` if the scanner *has* matched a line of text and *has not* encountered an error. So, the body of our `for` loop will be called only when there is a line of text in the scanner's buffer. This means our revised `CountLines` correctly handles the case where there is no trailing newline, and also handles the case where the file was empty.

Secondly, as `sc.Scan` returns `false` once an error is encountered, our `for` loop will exit when the end-of-

file is reached or an error is encountered. The `bufio.Scanner` type memoises the first error it encountered and we can recover that error once we've exited the loop using the `sc.Err()` method.

Lastly, `sc.Err()` takes care of handling `io.EOF` and will convert it to a `nil` if the end of file was reached without encountering another error.

Tip	When you find yourself faced with overbearing error handling, try to extract some of the operations into a helper type.

7.6.2. WriteResponse

My second example is inspired from the *Errors are values* blog post [11].

Earlier in this presentation We've seen examples dealing with opening, writing and closing files. The error handling is present, but not overwhelming as the operations can be encapsulated in helpers like `ioutil.ReadFile` and `ioutil.WriteFile`. However when dealing with low level network protocols it becomes necessary to build the response directly using I/O primitives the error handling can become repetitive. Consider this fragment of a HTTP server which is constructing the HTTP response.

```
type Header struct {
        Key, Value string
}

type Status struct {
        Code   int
        Reason string
}

func WriteResponse(w io.Writer, st Status, headers []Header, body
io.Reader) error {
        _, err := fmt.Fprintf(w, "HTTP/1.1 %d %s\r\n", st.Code, st.Reason)
        if err != nil {
                return err
        }

        for _, h := range headers {
                _, err := fmt.Fprintf(w, "%s: %s\r\n", h.Key, h.Value)
                if err != nil {
                        return err
                }
```

```
        }

        if _, err := fmt.Fprint(w, "\r\n"); err != nil {
                return err
        }

        _, err = io.Copy(w, body)
        return err
}
```

First we construct the status line using `fmt.Fprintf`, and check the error. Then for each header we write the header key and value, checking the error each time. Lastly we terminate the header section with an additional `\r\n`, check the error, and copy the response body to the client. Finally, although we don't need to check the error from `io.Copy`, we need to translate it from the two return value form that `io.Copy` returns into the single return value that `WriteResponse` returns.

That's a lot of repetitive work. But we can make it easier on ourselves by introducing a small wrapper type, `errWriter`.

`errWriter` fulfils the `io.Writer` contract so it can be used to wrap an existing `io.Writer`. `errWriter` passes writes through to its underlying writer until an error is detected. From that point on, it discards any writes and returns the previous error.

```
type errWriter struct {
        io.Writer
        err error
}

func (e *errWriter) Write(buf []byte) (int, error) {
        if e.err != nil {
                return 0, e.err
        }
        var n int
        n, e.err = e.Writer.Write(buf)
        return n, nil
}

func WriteResponse(w io.Writer, st Status, headers []Header, body
io.Reader) error {
```

```
        ew := &errWriter{Writer: w}
        fmt.Fprintf(ew, "HTTP/1.1 %d %s\r\n", st.Code, st.Reason)

        for _, h := range headers {
                fmt.Fprintf(ew, "%s: %s\r\n", h.Key, h.Value)
        }

        fmt.Fprint(ew, "\r\n")
        io.Copy(ew, body)
        return ew.err
}
```

Applying `errWriter` to `WriteResponse` dramatically improves the clarity of the code. Each of the operations no longer needs to bracket itself with an error check. Reporting the error is moved to the end of the function by inspecting the `ew.err` field, avoiding the annoying translation from `io.Copy's return values.

7.7. Only handle an error once

Lastly, I want to mention that you should only handle errors once. Handling an error means inspecting the error value, and making a *single* decision.

```
// WriteAll writes the contents of buf to the supplied writer.
func WriteAll(w io.Writer, buf []byte) {
        w.Write(buf)
}
```

If you make less than one decision, you're ignoring the error. As we see here, the error from `w.WriteAll` is being discarded.

But making *more than one* decision in response to a single error is also problematic. The following is code that I come across frequently.

```
func WriteAll(w io.Writer, buf []byte) error {
        _, err := w.Write(buf)
        if err != nil {
                log.Println("unable to write:", err) // annotated error goes to log file
                return err                           // unannotated error returned to caller
        }
        return nil
```

```
}
```

In this example if an error occurs during `w.Write`, a line will be written to a log file, noting the file and line that the error occurred, and the error is also returned to the caller, who possibly will log it, and return it, all the way back up to the top of the program.

The caller is probably doing the same

```
func WriteConfig(w io.Writer, conf *Config) error {
        buf, err := json.Marshal(conf)
        if err != nil {
                log.Printf("could not marshal config: %v", err)
                return err
        }
        if err := WriteAll(w, buf); err != nil {
                log.Println("could not write config: %v", err)
                return err
        }
        return nil
}
```

So you get a stack of duplicate lines in your log file,

unable to write: io.EOF
could not write config: io.EOF

but at the top of the program you get the original error without any context.

err := WriteConfig(f, &conf)
fmt.Println(err) // io.EOF

I want to dig into this a little further because I don't see the problems with logging *and* returning as just a matter of personal preference.

```
func WriteConfig(w io.Writer, conf *Config) error {
        buf, err := json.Marshal(conf)
        if err != nil {
                log.Printf("could not marshal config: %v", err)
                // oops, forgot to return
        }
        if err := WriteAll(w, buf); err != nil {
                log.Println("could not write config: %v", err)
```

```
                return err
        }
        return nil
}
```

The problem I see a lot is programmers forgetting to return from an error. As we talked about earlier, Go style is to use guard clauses, checking preconditions as the function progresses and returning early.

In this example the author checked the error, logged it, but *forgot* to return. This has caused a subtle bug.

The contract for error handling in Go says that you cannot make any assumptions about the contents of other return values in the presence of an error. As the JSON marshalling failed, the contents of `buf` are unknown, maybe it contains nothing, but worse it could contain a half written JSON fragment.

Because the programmer forgot to return after checking and logging the error, the corrupt buffer will be passed to `WriteAll`, which will probably succeed and so the config file will be written incorrectly. However the function will return just fine, and the only indication that a problem happened will be a single log line complaining about marshalling JSON, *not* a failure to write the config.

8. Testing

Someone asked me recently how much time I spend testing vs coding. I found the question to be curious for two reasons.

The first was the notion that testing was *not* coding. Perhaps the person asking me was used to repetitive manual testing. For me, my goal is to make as much of my testing automated, which means my tests are just more code.

The second was I realised that I spend a *lot* of time thinking agout testing. Specifically I think a lot about how to write software in a way that is testable. I worry about wandering into designs which will not be amenable to automated testing. So maybe the answer is I spend the majority of my time thinking about designing *for* testing.

The reality is the heady days of one to two QA engineers to each software engineer of the 90's are long gone, development teams are expected to be directly responsible for the quality of their products. This is probably a good thing.

Therefore, if we as individual contributors are expected to test the software we write, why do we need to automate it? Why is a manual testing plan not good enough?

Manual testing should not be the majority of your testing because manual testing is O(n) As the number of manual tests grows engineers are tempted to skip them, or only execute the scenarios they *think* are affected. Manual testing is expensive, and boring, 99.9% of the tests that passed last time will pass again.

This means that your initial response when given either a bug to fix or a feature to implement, should be to write a failing test *first*. This doesn't need to be a unit test, but it should be an automated test. But once you've

fixed the bug or added the feature you have the test case to prove it works, and you check them in together.

8.1. Tests lock in behaviour

In an earlier section I asserted that your packages should be designed around the behaviour they present. Unit tests at the package level should lock in the behaviour of the package's API. They describe, in code, what the package promises to do.

If there is a unit test for each input permutation, you have defined the contract for what the code will do *in code*, not documentation. This is a contract you can assert as simply as typing `go test`. At any stage, you can *know* with a high degree of confidence, that the behaviour people relied on before your change continues to function after your change.

8.2. Table driven testing

Let's say we have a function to split strings.

```
// Split slices s into all substrings separated by sep and
// returns a slice of the substrings between those separators.
func Split(s, sep string) []string {
        var result []string
        i := strings.Index(s, sep)
        for i > -1 {
                result = append(result, s[:i])
                s = s[i+len(sep):]
                i = strings.Index(s, sep)
        }
        return append(result, s)
}
```

How can we write a unit test for this function?

```
package split

import (
        "reflect"
        "testing"
)

func TestSplit(t *testing.T) {
        got := Split("a/b/c", "/")
```

```
        want := []string{"a", "b", "c"}
        if !reflect.DeepEqual(want, got) {
                t.Fatalf("expected: %v, got: %v", want, got)
        }
}
```

We start with a file in the same directory, with the same package name, `strings`. Tests are just regular Go functions with a few rules.

The name of the test function must start with `Test`.

The test function must take one argument of type `*testing.T`. A `*testing.T` is a type injected by the testing package itself, to provide ways to print, skip, and fail the test.

In our test we call `Split` with some inputs, then compare it to the result we expected.

8.2.1. Test scope

Because test functions are just regular public Go functions, we don't want to include them as part of our package's API. Instead, we want to only compile them in the context of running our tests. To do this the go test commands understands that files ending with `_test.go` are only compiled in *test scope*.

8.2.2. Code coverage

The next question is, what is the coverage of this package? Luckily the go tool has a built in branch coverage. We can invoke it like this:

```
% go test -coverprofile=c.out
PASS
coverage: 100.0% of statements
ok      split   0.010s
```

Which tells us we have 100% branch coverage, which isn't really surprising, there's only one branch in this code, a loop.

If we want to dig in to the coverge report the coverage tool has several options to print the coverage report. The first is to break down the coverage per function:

```
% go tool cover -func=c.out
split/split.go:8:       Split           100.0%
total:                  (statements)    100.0%
```

Which isn't that exciting as we only have one function in this package, but I'm sure you'll find more exciting

packages to test the coverage of.

Tip	This is so useful for me I have a shell alias which runs the test coverage and the report in one command: .bashrc `cover () { local t=$(mktemp -t cover) go test $COVERFLAGS -coverprofile=$t $@ \ && go tool cover -func=$t \ && unlink $t }`

So, we wrote one test case, got 100% coverage, but this isn't really the end of the story. We have good branch coverage but we probably need to test some of the boundary conditions.

For example, what happens if we try to split it on comma?

```
func TestSplitWrongSep(t *testing.T) {
        got := Split("a/b/c", ",")
        want := []string{"a/b/c"} // (1)
        if !reflect.DeepEqual(want, got) {
                t.Fatalf("expected: %v, got: %v", want, got)
        }
}
```

Or, what happens if there are no seperators in the source string?

```
func TestSplitNoSep(t *testing.T) {
        got := Split("abc", "/")
        want := []string{"abc"}
        if !reflect.DeepEqual(want, got) {
                t.Fatalf("expected: %v, got: %v", want, got)
        }
}
```

Now we're starting build a set of test cases that exercise boundary conditions. This is good.

However the there is a lot of duplication in our tests. For each test case only the input, the expected output, and name of the test case change. Everything else is boilderplate.

What we'd like to to set up all the inputs and expected outputs and feel them to the test harness. This is a great time to introduce a table driven test.

```
func TestSplit(t *testing.T) {
        type test struct {
```

```
                input string
                sep   string
                want  []string
        }

        tests := []test{
                {input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
                {input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
                {input: "abc", sep: "/", want: []string{"abc"}},
        }

        for _, tc := range tests {
                got := Split(tc.input, tc.sep)
                if !reflect.DeepEqual(tc.want, got) {
                        t.Fatalf("expected: %v, got: %v", tc.want, got)
                }
        }
}
```

We declare a structure to hold our test inputs and expected outputs. This will be a local declaration because we want to reuse this structure for other tests in this package.

In fact, we don't even need to give it a name, we can use an anonymous struct literal to reduce the boilerplate a little more.

```
func TestSplit(t *testing.T) {
        tests := []struct {
                input string
                sep   string
                want  []string
        }{
                {input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
                {input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
                {input: "abc", sep: "/", want: []string{"abc"}},
        }

        for _, tc := range tests {
                got := Split(tc.input, tc.sep)
                if !reflect.DeepEqual(tc.want, got) {
```

```
                        t.Fatalf("expected: %v, got: %v", tc.want, got)
                }
        }
}
```

Now, adding a new test is a straight forward matter; simply add another line the test ficture strutures. For example, what will happen if our input string has a trailing separator?

```
                {input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
                {input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
                {input: "abc", sep: "/", want: []string{"abc"}},
                {input: "a/b/c/", sep: "/", want: []string{"a", "b",
"c"}}, // trailing sep
```

And when we run go test, we get

```
% go test
--- FAIL: TestSplit (0.00s)
    split_test.go:24: expected: [a b c], got: [a b c ]
FAIL
exit status 1
FAIL    split   0.006s
```

There are a few problems to talk about here.

The first is by rewriting each test from a function to a row in a table we've lost the name of the failing test. We added a comment in the test file to call out this case, but we don't have access to that in the go test output.

There are a few ways to resolve this, and you'll see these styles in use Go code bases because the table testing idiom is evolving as people continue to experiment with the form.

8.2.3. Enumerating test cases

As tests are stored in a slice we can print out the index of the test case in the failure message:

```
func TestSplit(t *testing.T) {
        tests := []struct {
                input string
                sep   string
                want  []string
        }{
                {input: "a/b/c", sep: "/", want: []string{"a", "b", "c"}},
```

```
                {input: "a/b/c", sep: ",", want: []string{"a/b/c"}},
                {input: "abc", sep: "/", want: []string{"abc"}},
                {input: "a/b/c/", sep: "/", want: []string{"a", "b",
"c"}}, // trailing sep
        }

        for i, tc := range tests {
                got := Split(tc.input, tc.sep)
                if !reflect.DeepEqual(tc.want, got) {
                        t.Fatalf("test %d: expected: %v, got: %v", i+1,
tc.want, got)
                }
        }
}
```

So now we get this

```
% go test
--- FAIL: TestSplit (0.00s)
    split_test.go:24: test 4: expected: [a b c], got: [a b c ]
FAIL
exit status 1
FAIL    split   0.005s
```

Which is a little better. Now we know that the fourth test is failing, although we have to do a little bit of fudging because slice indexing—and range iteration—is zero based. This requires consistency across your test cases; if some use zero base reporting and others use one based, it's going to be confusing. And, if the list of test cases is long, it could be difficult to count braces to figure out exactly which line is test case 4.

8.2.4. Name your tests

Another common pattern is to include a name field in the test fixture.

```
func TestSplit(t *testing.T) {
        tests := []struct {
                name  string
                input string
                sep   string
                want  []string
        }{
```

```
                {name: "simple", input: "a/b/c", sep: "/", want:
[]string{"a", "b", "c"}},
                {name: "wrong sep", input: "a/b/c", sep: ",", want:
[]string{"a/b/c"}},
                {name: "no sep", input: "abc", sep: "/", want:
[]string{"abc"}},
                {name: "trailing sep", input: "a/b/c/", sep: "/", want:
[]string{"a", "b", "c"}},
        }

        for _, tc := range tests {
                got := Split(tc.input, tc.sep)
                if !reflect.DeepEqual(tc.want, got) {
                        t.Fatalf("%s: expected: %v, got: %v", tc.name,
tc.want, got)
                }
        }
}
```

Now when the test fails we have a descriptive name for what the test was doing. We no longer have to try to figure it out from the arguments—and a string we can search on.

```
% go test
--- FAIL: TestSplit (0.00s)
    split_test.go:25: trailing sep: expected: [a b c], got: [a b c ]
FAIL
exit status 1
FAIL    split   0.005s
```

We can dry this up even more using a map literal syntax:

```
func TestSplit(t *testing.T) {
        tests := map[string]struct {
                input string
                sep   string
                want  []string
        }{
                "simple":       {input: "a/b/c", sep: "/", want:
[]string{"a", "b", "c"}},
```

```
                "wrong sep":    {input: "a/b/c", sep: ",", want:
[]string{"a/b/c"}},
                "no sep":       {input: "abc", sep: "/", want:
[]string{"abc"}},
                "trailing sep": {input: "a/b/c/", sep: "/", want:
[]string{"a", "b", "c"}},
        }

        for name, tc := range tests {
                got := Split(tc.input, tc.sep)
                if !reflect.DeepEqual(tc.want, got) {
                        t.Fatalf("%s: expected: %v, got: %v", name,
tc.want, got)
                }
        }
}
```

Using a map literal syntax we define our test cases not as a slice of structs, but as map of test names to test fixtures. There's also a side benefit of using a map that is going to potentially improve the utility of our tests.

Map iteration order is *undefined*. This means each time we run `go test`, our tests are going to be run in a random order.

This is super useful for spotting conditions where two test pass when run in a certain order, but not otherwise. If you find that happens you probably have some global state that is being mutated by one test, with subsiquent tests depending on that modification.

8.2.5. Sub tests

Before we fix the failing test there are a few other issues to fix with our table driven test harness.

The first is we're calling `t.Fatalf` when one of the test cases fails. This means the first failing test case and we stop testing the other cases. Because test cases are run in an undefined order, if there is a test failure, we'd like to know was it the only one or is it just the first.

The testing package would do this for us if we go to the effort to write out each test case as its own function and in Go 1.7 a new feature was added that lets us do this easily for table driven tests.

```
func TestSplit(t *testing.T) {
        tests := map[string]struct {
                input string
```

```
                sep   string
                want  []string
        }{
                "simple":       {input: "a/b/c", sep: "/", want:
[]string{"a", "b", "c"}},
                "wrong sep":    {input: "a/b/c", sep: ",", want:
[]string{"a/b/c"}},
                "no sep":       {input: "abc", sep: "/", want:
[]string{"abc"}},
                "trailing sep": {input: "a/b/c/", sep: "/", want:
[]string{"a", "b", "c"}},
        }

        for name, tc := range tests {
                t.Run(name, func(t *testing.T) {
                        got := Split(tc.input, tc.sep)
                        if !reflect.DeepEqual(tc.want, got) {
                                t.Fatalf("expected: %v, got: %v", tc.want,
got)
                        }
                })
        }
}
```

Each subtest now has a name. We get that name automatically printed out in any test runs.

```
% go test
--- FAIL: TestSplit (0.00s)
    --- FAIL: TestSplit/trailing_sep (0.00s)
        split_test.go:25: expected: [a b c], got: [a b c ]
FAIL
exit status 1
FAIL    split   0.005s
```

Each subtest is its own anonymous function, therefore we can use `t.Fatalf`, `t.Skipf`, and all the other `testing.T` helpers, while retaining the compactness of a table driven test.

Tip	Individual sub test cases can be executed directly Because sub tests have a name, you can run a selection of sub tests by name using the `go test -run`

flag.

```
% go test -run=.*/trailing -v
=== RUN   TestSplit
=== RUN   TestSplit/trailing_sep
--- FAIL: TestSplit (0.00s)
    --- FAIL: TestSplit/trailing_sep (0.00s)
        split_test.go:25: expected: [a b c], got: [a b c ]
FAIL
exit status 1
```

8.2.6. Comparing expected an actual

Now we're ready to fix the test case. Let's look at the error.

```
--- FAIL: TestSplit (0.00s)
    --- FAIL: TestSplit/trailing_sep (0.00s)
        split_test.go:25: expected: [a b c], got: [a b c ]
```

Can you spot the problem? Clearly the slices are different, that's what `reflect.DeepEqual` is upset about. But spotting the actual difference isn't easy, you have to spot that extra space after `c`.

We can improve the output if we switch to the `%#v` syntax to view the value as a Go(ish) declaration:

```
                        got := Split(tc.input, tc.sep)
                        if !reflect.DeepEqual(tc.want, got) {
                                t.Fatalf("expected: %#v, got: %#v",
tc.want, got)
                        }
```

Now when we run out test it's clear that the problem is there is an extra blank element in the slice.

```
% go test
--- FAIL: TestSplit (0.00s)
    --- FAIL: TestSplit/trailing_sep (0.00s)
        split_test.go:25: expected: []string{"a", "b", "c"}, got:
[]string{"a", "b", "c", ""}
FAIL
exit status 1
FAIL    split   0.005s
```

But before we go to fix this I want to talk a little bit more about choosing the right way to present test failures. Our Split function is simple, it takes a primitive string and returns a slice of strings, but what if it worked with structs, or worse, pointers to structs?

Here is an example where %#v does not work as well:

```
func main() {
        type T struct {
                I int
        }
        x := []*T{{1}, {2}, {3}}
        y := []*T{{1}, {2}, {4}}

        fmt.Printf("%v %v\n", x, y) // (1)
        fmt.Printf("%#v %#v\n", x, y) // (2)
}
```

```
[0xc000096000 0xc000096008 0xc000096010] [0xc000096018 0xc000096020
0xc000096028]
```

```
[]*main.T{(*main.T)(0xc000096000), (*main.T)(0xc000096008), (*main.T)
(0xc000096010)} []*main.T{(*main.T)(0xc000096018), (*main.T)(0xc000096020),
(*main.T)(0xc000096028)}
```

I want to introduce the go-cmp library from Google. This was introduced about two years ago by Joe Tsai. He gave a talk about it at GopherCon in 2017

The goal of the compare library is it is specifically to compare two values. This is similar to `reflect.DeepEqual`, but it has more capabilities. You can of course write:

```
func main() {
        type T struct {
                I int
        }
        x := []*T{{1}, {2}, {3}}
        y := []*T{{1}, {2}, {4}}

        fmt.Println(cmp.Equal(x, y)) // false
}
```

But far more useful for us with our test function is the Diff method which will produce a textual description of what is different between the two values, recursively.

```
func main() {
        type T struct {
                I int
```

```
        }
        x := []*T{{1}, {2}, {3}}
        y := []*T{{1}, {2}, {4}}

        diff := cmp.Diff(x, y)
        fmt.Printf(diff) // (1)
}
```

```
% go run .
{[]*main.T}[2].I:
        -: 3
        +: 4
```

Putting this all together we have our table driven go-cmp test

```
func TestSplit(t *testing.T) {
        tests := map[string]struct {
                input string
                sep   string
                want  []string
        }{
                "simple":       {input: "a/b/c", sep: "/", want:
[]string{"a", "b", "c"}},
                "wrong sep":    {input: "a/b/c", sep: ",", want:
[]string{"a/b/c"}},
                "no sep":       {input: "abc", sep: "/", want:
[]string{"abc"}},
                "trailing sep": {input: "a/b/c/", sep: "/", want:
[]string{"a", "b", "c"}},
        }

        for name, tc := range tests {
                t.Run(name, func(t *testing.T) {
                        got := Split(tc.input, tc.sep)
                        diff := cmp.Diff(tc.want, got)
                        if diff != "" {
                                t.Fatalf(diff)
                        }
                })
```

```
        }
}
```

Running this we get

```
% go test
--- FAIL: TestSplit (0.00s)
    --- FAIL: TestSplit/trailing_sep (0.00s)
        split_test.go:27: {[]string}[?->3]:
                -: <non-existent>
                +: ""
FAIL
exit status 1
FAIL    split   0.006s
```

Now our test isn't just telling us that what we got and what we wanted were different. Our test is telling usthat the strings are different lengths, the third index in the fixture didn't exist, but in the actual output we got an empty string, "".

8.3. Prefer internal tests to external tests

The `go` tool supports writing your `testing` package tests in two places. Assuming your package is called `http2`, you can write a `http2_test.go` file and use the `package http2` declaration. Doing so will compile the code in `http2_test.go` *as if* it were part of the `http2` package. This is known colloquially as an *internal* test.

The `go` tool also supports a special package declaration, ending in `test`, *ie.*, `package http_test`. *This allows your test files to live alongside your code in the same package, however when those tests are compiled they are not part of your package's code, they live in their own package. This allows you to write your tests as if you were another package calling into your code. This is known as an _external* test.

I recommend using internal tests when writing unit tests for your package. This allows you to test each function or method directly, avoiding the bureaucracy of external testing.

However, you *should* place your `Example` test functions in an external test file. This ensures that when viewed in godoc, the examples have the appropriate package prefix and can be easily copy pasted.

Tip	Use internal tests and avoid dot imports.

8.4. Tests give you the power to release any day of the week

Testing is more than just the hand to hand combat of software development. Tests are a critical component of

making sure you can always ship your master branch.

For all the La Croix and ping pong tables, at the end of the day, as a development team, you are judged on your ability to deliver working software to the business. Your super power is at any time anyone on the team should be confident that the master branch of your code is shippable—it may not have all the desired features, but what features are there should work as expected. This means at any time you can deliver a release of your software to the business and the business can recoup its investment in your development R&D.

I cannot emphasise this enough. If you want the non technical parts of the business to believe you are heros, you must never put yourself in the situation where you tell them "well, we can't release right now because we're in the middle of an important refactoring. It'll be a few weeks. We hope."

I'm not saying you cannot refactor, but at every stage your product has to be shippable. To give yourself the confidence that your product works as described, your tests have to pass.

8.5. Tests give you confidence to change someone else's code

Maybe the most important aspect of testing is good coverage gives you the confidence to change code that you didn't write. This is critical when working in a codebase that is shared with others, or more likely on a codebase that has evolved alongside a team.

Fundamentally solid test coverage gives you the confidence to change the codebase you're working in.

9. Concurrency

Often Go is chosen for a project because of its concurrency features. The Go team have gone to great lengths to make concurrency in Go cheap (in terms of hardware resources) and performant, however it is possible to use Go's concurrency features to write code which is neither performant or reliable. With the time I have left I want to leave you with some advice for avoid some of the pitfalls that come with Go's concurrency features.

Go features first class support for concurrency with channels, and the `select` and `go` statements. If you've learnt Go formally from a book or training course, you might have noticed that the concurrency section is always one of the last you'll cover. This workshop is no different, I have chosen to cover concurrency last, as if it is somehow additional to the regular the skills a Go programmer should master.

There is a dichotomy here; Go's headline feature is our simple, lightweight concurrency model. As a product, our language almost sells itself on this feature alone. On the other hand, there is a narrative that concurrency isn't actually that easy to use, otherwise authors wouldn't make it the last chapter in their book.

9.1. Channel Axioms

Go programmers quickly grasp the idea of a channel as a queue of values and are comfortable with the notion that channel operations may block when full or empty. In this section I want to explore several of the less

common properties of channels.

9.1.1. A send to a nil channel blocks forever

It can be surprising to newcomers that is a send on a `nil` channel blocks forever.

```
func main() {
        var c chan string
        c <- "let's get started" // (1)
}
```

deadlock

This example program will deadlock on line 5 because the zero value for an uninitialised channel is `nil`. You cannot send to a channel that has not been initialised.

9.1.2. A receive from a nil channel blocks forever

Similarly receiving from a nil channel blocks the receiver forever.

```
func main() {
        var c chan string
        fmt.Println(<-c) // (1)
}
```

deadlock

So why does this happen? Here is one possible explanation:

The size of a channel's buffer is not part of its type declaration, so it must be part of the channel's value.

If the channel is not initialised then its buffer size will be zero.

If the size of the channel's buffer is zero, then the channel is unbuffered.

If the channel is unbuffered, then a send will block until another goroutine is ready to receive.

If the channel is `nil` then the sender and receiver have no reference to each other; they are both blocked waiting on independent channels and will never unblock.

9.1.3. A send to a closed channel panics

The following program will likely panic as the first goroutine to reach 10 will close the channel before its siblings have time to finish sending their values.

```
func main() {
        var c = make(chan int, 100)
        for i := 0; i < 10; i++ {
                go func() {
                        for j := 0; j < 10; j++ {
                                c <- j
                        }
                        close(c)
                }()
        }
        for i := range c {
                fmt.Println(i)
        }
}
```

So why isn't there a version of `close` that lets you check if a channel is closed? Something like this:

```
if !isClosed(c) {
        // c isn't closed, send the value
        c <- v
}
```

But this function would have an inherent race. Someone may close the channel after we checked `isClosed(c)` but *before* the code gets to `c ← v`.

One way to think about how this is possible is to imagine that goroutines work in different universes. They cannot observe each other unless they communicate. Because they cannot observe each other except for these communication points, time moves differently for each goroutine (given we cannot prove the opposite; time moves at the same rate for each goroutine, we must admit that it is *possible*) hence you cannot make statements like "a small amount of time" when talking about the interactions of different goroutines. There is no *happens before* relationship with goroutines unless they explicitly communicate.

This is not just a theoretical bun fight, it is easily demonstrable that the operating system thread backing any goroutine may be rescheduled at any time by the operating system. A different thread hosting a different goroutine can move ahead, relative to the sleeping thread, in time easily able to execute the channel close operation before the original thread is revived to attempt to send on the now closed channel.

If you need to ensure that only one goroutine closes a channel you must create a point of coordination *before* the `close` operation.

```
func main() {
```

```
        var c = make(chan int, 100)
        var mu sync.Mutex
        var closed bool
        for i := 0; i < 10; i++ {
                go func() {
                        for j := 0; j < 10; j++ {
                                c <- j
                        }
                        mu.Lock()
                        if !closed {
                                close(c)
                                closed = true
                        }
                        mu.Unlock()
                }()
        }
        for i := range c {
                fmt.Println(i)
        }
}
```

9.1.4. A receive from a closed channel returns the zero value immediately

The final case is the inverse of the previous. Once a channel is closed *and* all values drained from its buffer, the channel will always return zero values immediately.

```
func main() {
        c := make(chan int, 3)
        c <- 1
        c <- 2
        c <- 3
        close(c)
        for i := 0; i < 4; i++ {
                fmt.Printf("%d ", <-c) // prints 1 2 3 0
        }
}
```

Note	When consuming values from a channel until it closes, the better solution is to use a `for range` style loop.

```
for v := range c {
        // do something with v
}
```

Which is just syntactic sugar over the more verbose

```
for v, ok := <- c; ok ; v, ok = <- c {
        // do something with v
}
```

These two statements are equivalent in function, and demonstrate what for range is doing under the hood.

9.2. Prefer channels with a size of zero or one

When dealing with an unknown producer or consumer choose a buffer size of zero or one.

A buffer size of zero is ideal for coordination. A buffer size of one is idea to permit the sender to deposit the value without blocking and move on.

A buffer size greater than one is useful in the case where you know that exact number of values that will be deposited in the channel *before* it is drained. The common case is multiple workers operating in parallel, and a coordinator waiting on that result.

The most reasonable channels sizes are usually zero and one. Most other sizes are *guesses*. When you guess incorrectly, the program is unreliable.

9.3. Keep yourself busy or do the work yourself

What is the problem with this program?

```
package main

import (
        "fmt"
        "log"
        "net/http"
)

func main() {
        http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request)
{
                fmt.Fprintln(w, "Hello, GopherCon SG")
        })
        go func() {
```

```
                if err := http.ListenAndServe(":8080", nil); err != nil {
                        log.Fatal(err)
                }
        }()

        for {
        }
}
```

The program does what we intended, it serves a simple web server. However it also does something else at the same time, it wastes CPU in an infinite loop. This is because the `for{}` on the last line of `main` is going to block the main goroutine because it doesn't do any IO, wait on a lock, send or receive on a channel, or otherwise communicate with the scheduler.

As the Go runtime is mostly cooperatively scheduled, this program is going to spin fruitlessly on a single CPU, and may eventually end up live-locked.

How could we fix this? Here's one suggestion.

```
package main

import (
        "fmt"
        "log"
        "net/http"
        "runtime"
)

func main() {
        http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request)
{
                fmt.Fprintln(w, "Hello, GopherCon SG")
        })
        go func() {
                if err := http.ListenAndServe(":8080", nil); err != nil {
                        log.Fatal(err)
                }
        }()
```

```
        for {
                runtime.Gosched()
        }
}
```

This might look silly, but it's a common common solution I see in the wild. It's symptomatic of not understanding the underlying problem.

Now, if you're a little more experienced with go, you might instead write something like this.

```
package main

import (
        "fmt"
        "log"
        "net/http"
)

func main() {
        http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request)
{
                fmt.Fprintln(w, "Hello, GopherCon SG")
        })
        go func() {
                if err := http.ListenAndServe(":8080", nil); err != nil {
                        log.Fatal(err)
                }
        }()

        select {}
}
```

An empty select statement will block forever. This is a useful property because now we're not spinning a whole CPU just to call `runtime.GoSched()`. However, we're only treating the symptom, not the cause.

I want to present to you another solution, one which has hopefully already occurred to you. Rather than run `http.ListenAndServe` in a goroutine, leaving us with the problem of what to do with the main goroutine, simply run `http.ListenAndServe` on the main goroutine itself.

Tip	If the `main.main` function of a Go program returns then the Go program will unconditionally exit no matter what other goroutines started by the program over time are doing.

```
package main

import (
        "fmt"
        "log"
        "net/http"
)

func main() {
        http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request)
{
                fmt.Fprintln(w, "Hello, GopherCon SG")
        })
        if err := http.ListenAndServe(":8080", nil); err != nil {
                log.Fatal(err)
        }
}
```

So this is my first piece of advice: if your goroutine cannot make progress until it gets the result from another, oftentimes it is simpler to just do the work yourself rather than to delegate it.

This often eliminates a lot of state tracking and channel manipulation required to plumb a result back from a goroutine to its initiator.

Tip	Many Go programmers overuse goroutines, especially when they are starting out. As with all things in life, moderation is the key to success.

9.4. Leave concurrency to the caller

What is the difference between these two APIs?

```
// ListDirectory returns the contents of dir.
func ListDirectory(dir string) ([]string, error)

// ListDirectory returns a channel over which
// directory entries will be published. When the list
// of entries is exhausted, the channel will be closed.
func ListDirectory(dir string) chan string
```

The obvious differences are the first example reads a directory into a slice then returns the whole slice, or an error if something went wrong. This happens synchronously, the caller of `ListDirectory` blocks until all

directory entries have been read. Depending on how large the directory, this could take a long time, and could potentially allocate a lot of memory building up the slide of directory entry names.

Lets look at the second example. This is a little more Go like, `ListDirectory` returns a channel over which directory entries will be passed. When the channel is closed, that is your indication that there are no more directory entries. As the population of the channel happens *after* `ListDirectory` returns, `ListDirectory` is probably starting a goroutine to populate the channel.

Note	It's not necessary for the second version to actually use a Go routine; it could allocate a channel sufficient to hold all the directory entries without blocking, fill the channel, close it, then return the channel to the caller. But this is unlikely, as this would have the same problems with consuming a large amount of memory to buffer all the results in a channel.

The channel version of `ListDirectory` has two further problems:

By using a closed channel as the signal that there are no more items to process there is no way for `ListDirectory` to tell the caller that the set of items returned over the channel is incomplete because an error was encountered partway through. There is no way for the caller to tell the difference between an *empty directory* and an *error* to read from the directory entirely. Both result in a channel returned from `ListDirectory` which appears to be closed immediately.

The caller *must* continue to read from the channel until it is closed because that is the only way the caller can know that the goroutine which was started to fill the channel has stopped. This is a serious limitation on the use of `ListDirectory`, the caller has to spend time reading from the channel even though it may have received the answer it wanted. It is probably more efficient in terms of memory usage for medium to large directories, but this method is no faster than the original slice based method.

The solution to the problems of both implementations is to use a callback, a function that is called in the context of each directory entry as it is executed.

```
func ListDirectory(dir string, fn func(string))
```

Not surprisingly this is how the `filepath.WalkDir` function works.

Tip	If your function starts a goroutine you must provide the caller with a way to explicitly stop that goroutine. It is often easier to leave decision to execute a function asynchronously to the caller of that function.

9.5. Never start a goroutine without knowing when it will stop

Perhaps fitting for the final topic in this presentation, we're going to talk about stopping.

A previous example showed using a goroutine when one wasn't really necessary. But one of the driving reasons for using Go is the first class concurrency features the language offers. Indeed there are many instances where you want to exploit the parallelism available in your hardware. To do so, you must use goroutines.

This simple application serves http traffic on two different ports, port 8080 for application traffic and port 8001 for access to the `/debug/pprof` endpoint.

```
package main

import (
        "fmt"
        "net/http"
        _ "net/http/pprof"
)

func main() {
        mux := http.NewServeMux()
        mux.HandleFunc("/", func(resp http.ResponseWriter, req
*http.Request) {
                fmt.Fprintln(resp, "Hello, QCon!")
        })
        go http.ListenAndServe("127.0.0.1:8001", http.DefaultServeMux) //
debug
        http.ListenAndServe("0.0.0.0:8080", mux)                       //
app traffic
}
```

Although this program isn't very complicated, it represents the basis of a real application.

There are a few problems with the application as it stands which will reveal themselves as the application grows, so lets address a few of them now.

```
func serveApp() {
        mux := http.NewServeMux()
        mux.HandleFunc("/", func(resp http.ResponseWriter, req
*http.Request) {
                fmt.Fprintln(resp, "Hello, QCon!")
        })
        http.ListenAndServe("0.0.0.0:8080", mux)
}

func serveDebug() {
        http.ListenAndServe("127.0.0.1:8001", http.DefaultServeMux)
```

```
}

func main() {
        go serveDebug()
        serveApp()
}
```

By breaking the `serveApp` and `serveDebug` handlers out into their own functions we've decoupled them from `main.main`. We've also followed the advice from above and make sure that `serveApp` and `serveDebug` leave their concurrency to the caller.

But there are some operability problems with this program. If `serveApp` returns then `main.main` will return causing the program to shutdown and be restarted by whatever process manager you're using.

Tip	Just as functions in Go leave concurrency to the caller, applications should leave the job of monitoring their status and restarting them if they fail to the program that invoked them. Do not make your applications responsible for restarting themselves, this is a procedure best handled from outside the application.

However, `serveDebug` is run in a separate goroutine and if it returns just that goroutine will exit while the rest of the program continues on. Your operations staff will not be happy to find that they cannot get the statistics out of your application when they want too because the `/debug` handler stopped working a long time ago.

What we want to ensure is that if *any* of the goroutines responsible for serving this application stop, we shut down the application.

```
func serveApp() {
        mux := http.NewServeMux()
        mux.HandleFunc("/", func(resp http.ResponseWriter, req
*http.Request) {
                fmt.Fprintln(resp, "Hello, QCon!")
        })
        if err := http.ListenAndServe("0.0.0.0:8080", mux); err != nil {
                log.Fatal(err)
        }
}

func serveDebug() {
        if err := http.ListenAndServe("127.0.0.1:8001",
http.DefaultServeMux); err != nil {
```

```
                log.Fatal(err)
        }
}

func main() {
        go serveDebug()
        go serveApp()
        select {}
}
```

Now `serverApp` and `serveDebug` check the error returned from `ListenAndServe` and call `log.Fatal` if required. Because both handlers are running in goroutines, we park the main goroutine in a `select{}`.

This approach has a number of problems:

If `ListenAndServer` returns with a `nil` error, `log.Fatal` won't be called and the HTTP service on that port will shut down without stopping the application.

`log.Fatal` calls `os.Exit` which will unconditionally exit the program; defers won't be called, other goroutines won't be notified to shut down, the program will just stop. This makes it difficult to write tests for those functions.

Tip	Only use `log.Fatal` from `main.main` or `init` functions.

What we'd really like is to pass any error that occurs back to the originator of the goroutine so that it can know *why* the goroutine stopped, can shut down the process cleanly.

```
func serveApp() error {
        mux := http.NewServeMux()
        mux.HandleFunc("/", func(resp http.ResponseWriter, req
*http.Request) {
                fmt.Fprintln(resp, "Hello, QCon!")
        })
        return http.ListenAndServe("0.0.0.0:8080", mux)
}

func serveDebug() error {
        return http.ListenAndServe("127.0.0.1:8001", http.DefaultServeMux)
}
```

```
func main() {
        done := make(chan error, 2)
        go func() {
                done <- serveDebug()
        }()
        go func() {
                done <- serveApp()
        }()

        for i := 0; i < cap(done); i++ {
                if err := <-done; err != nil {
                        fmt.Println("error: %v", err)
                }
        }
}
```

We can use a channel to collect the return status of the goroutine. The size of the channel is equal to the number of goroutines we want to manage so that sending to the `done` channel will not block, as this will block the shutdown the of goroutine, causing it to leak.

As there is no way to safely close the `done` channel we cannot use the `for range` idiom to loop of the channel until all goroutines have reported in, instead we loop for as many goroutines we started, which is equal to the capacity of the channel.

Now we have a way to wait for each goroutine to exit cleanly and log any error they encounter. All that is needed is a way to forward the shutdown signal from the first goroutine that exits to the others.

It turns out that asking a `http.Server` to shut down is a little involved, so I've spun that logic out into a helper function. The `serve` helper takes an address and `http.Handler`, similar to `http.ListenAndServe`, and also a `stop` channel which we use to trigger the `Shutdown` method.

```
func serve(addr string, handler http.Handler, stop <-chan struct{}) error
{
        s := http.Server{
                Addr:    addr,
                Handler: handler,
        }

        go func() {
                <-stop // wait for stop signal
```

```
                s.Shutdown(context.Background())
        }()

        return s.ListenAndServe()
}

func serveApp(stop <-chan struct{}) error {
        mux := http.NewServeMux()
        mux.HandleFunc("/", func(resp http.ResponseWriter, req
*http.Request) {
                fmt.Fprintln(resp, "Hello, QCon!")
        })
        return serve("0.0.0.0:8080", mux, stop)
}

func serveDebug(stop <-chan struct{}) error {
        return serve("127.0.0.1:8001", http.DefaultServeMux, stop)
}

func main() {
        done := make(chan error, 2)
        stop := make(chan struct{})
        go func() {
                done <- serveDebug(stop)
        }()
        go func() {
                done <- serveApp(stop)
        }()

        var stopped bool
        for i := 0; i < cap(done); i++ {
                if err := <-done; err != nil {
                        fmt.Println("error: %v", err)
                }
                if !stopped {
                        stopped = true
                        close(stop)
```

```
                }
        }
}
```

Now, each time we receive a value on the `done` channel, we close the `stop` channel which causes all the goroutines waiting on that channel to shut down their `http.Server`. This in turn will cause all the remaining `ListenAndServe` goroutines to return. Once all the goroutines we started have stopped, `main.main` returns and the process stops cleanly.