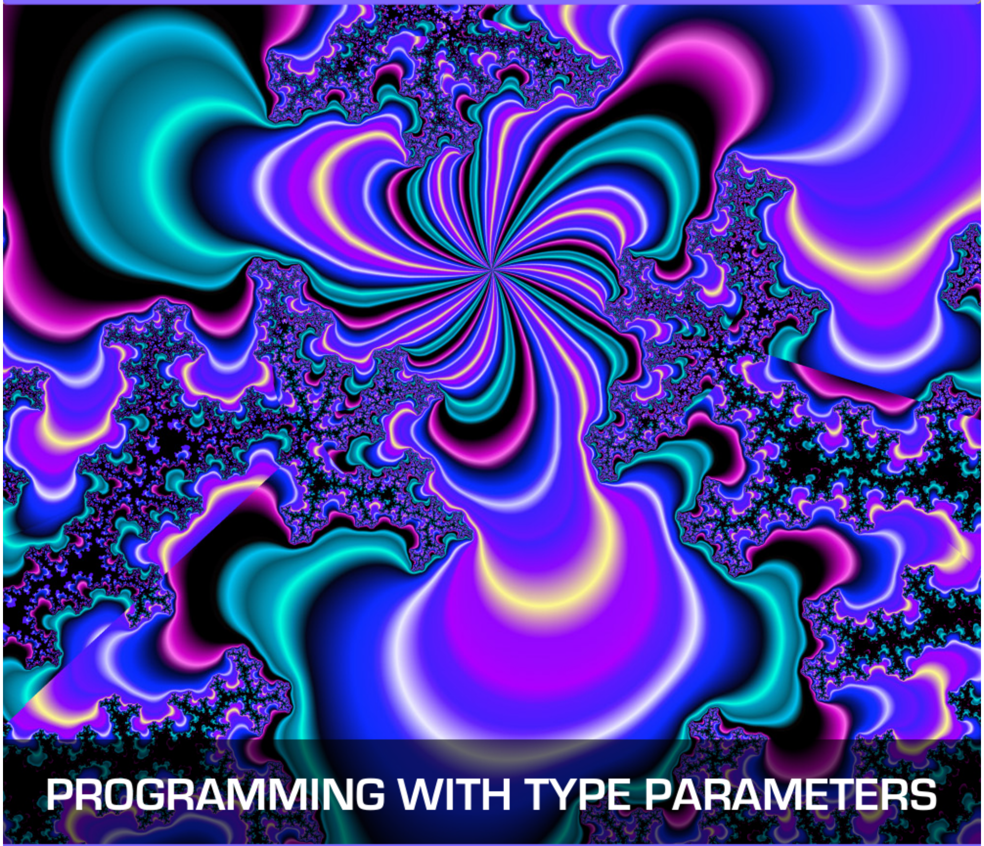
# Know Go - Generics

John Arundel

April 18, 2022

# KNOW GO GENERICS

"Beautifully explained"
—Pavel Anni



PROGRAMMING WITH TYPE PARAMETERS

JOHN ARUNDEL

# Contents

# Praise for ‘Know Go: Generics’

Everything I wanted to know about generics in Go, beautifully explained!
—Pavel Anni

I really loved reading this book. I found the idea of generics scary at first, but now I’m very comfortable with it thanks to John’s simple yet complete way of teaching.
—Shivdas Patil

It’s taken me from being apprehensive about learning something new to being excited about the possibilities that generics brings.
—Dan Macklin

Well written: the explanations and examples are clear and easy to understand.
—Pedro Sandoval

Nature, not content with denying to Mr — the faculty of thought, has endowed him with the faculty of writing.
—A. E. Housman

# Introduction

*The more I use Go, the more I think generics would be a useless misfeature. Why do so many people think they need them?*
—Aram Hăvărneanu

Hello, welcome to the book, and welcome to the world of generic programming in Go! I'm looking forward to exploring it with you.

## What is the book about?

This book is about *generics* in Go. We'll talk about exactly what that means in more detail later on, but, briefly, it's about defining (and using) generic functions and generic types.

First, what's a generic function? It's one that doesn't specify the types of all its parameters in advance. Instead, some types are represented by placeholders, called *type parameters*.

Generic *types* work the same way. For example, we might want to define a slice of elements of some unspecified type, or a struct with some field of a type that will be known later.

Like many things in programming, generics sounds complicated at first, but once you get your head round it, it's actually quite straightforward. In this book, we'll work together through the steps necessary to understand what generic functions and types are, why they're useful, how they work in Go, and what fun and interesting things we can do with them.

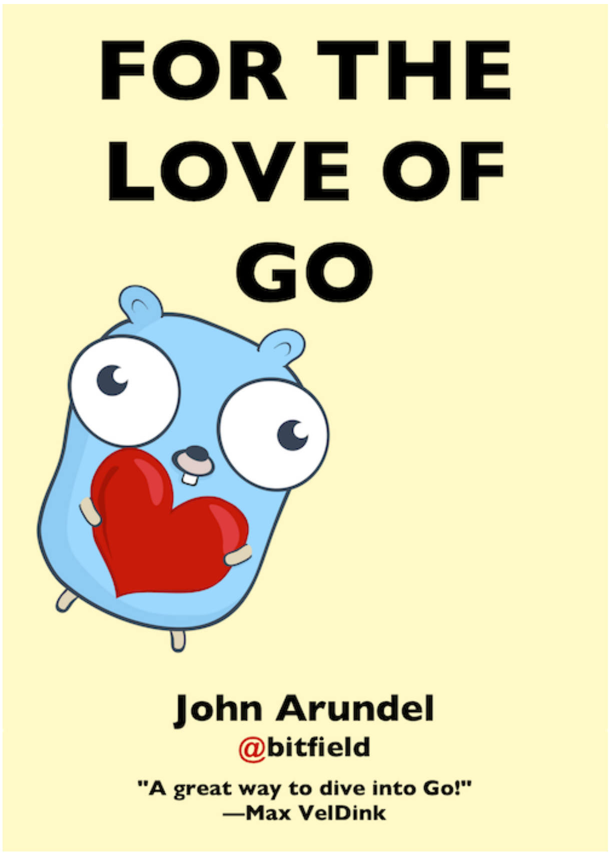
## Who is the book for?

This book is for people who are new to the generics features in Go and want to know what they are, how to use them, and what they should do differently now that Go has generics. If you have some experience using Go prior to the introduction of generics, and you just want to know what's new, you'll find everything you need to know right here.

If you're used to using generics in other languages, such as Java or C++, and you'd like to know how that experience will translate to Go, this book is also for you. If you've considered using Go in the past but decided against it for one reason or another, maybe the introduction of generics will tip the balance for you. This book will help you decide whether you'll be able to do what you want to do with Go's generics.

And whether you have any experience with Go or not, you may be worried that generics adds unnecessary complexity to the language and will make it harder for you to understand, or even write, programs. This book is for you too! I hope you'll find that generic programming in Go isn't as difficult or complicated as it might sound. In fact, it's extremely straightforward, when we approach it the right way.

If you're completely new to Go, or even to programming, I recommend you read my previous book, "For the Love of Go", first. It'll give you a good grounding in the basics of Go, which will help you understand the material in *this* book more easily:



- https://bitfieldconsulting.com/books/love

## What version of Go does it cover?

This book requires Go 1.18, released in March 2022, and all the code samples have been tested against at least that version. Although experimental (and strongly deprecated) support for generics was included in Go 1.17, you shouldn't use it. To use generics in production programs, you need to upgrade to at least Go 1.18.

If a suitable version of Go isn't yet available as a package in your operating system distribution, you can build it from source or download a suitable binary package from the Go website directly:

- https://go.dev/learn/

If you can't upgrade to Go 1.18 yet, for whatever reason, you can still play with generics and try out the examples in this book, using the Go Playground:

- https://go.dev/play/

## Where to find the code examples

There are lots of exercise for you to solve throughout the book, each designed to help you test your understanding of the concepts you've just learned. If you run into trouble, or just want to check your code, each exercise is accompanied by a complete sample solution, with tests.

All these solutions are also available in a public GitHub repo here:

- https://github.com/bitfield/kg-generics

Each exercise in the book is accompanied by a number (for example, Exercise 2.1), and you'll find the solution to that exercise in the numbered folder of the repo.

## What you'll learn

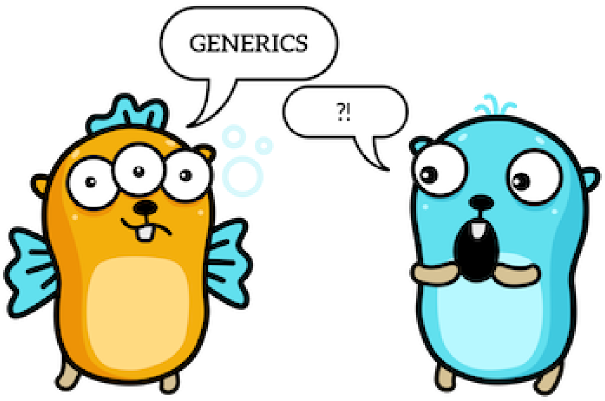
By reading through this book and completing the exercises, you'll learn:

- What we mean by *generic programming* in general, and specifically how that applies to Go
- What *type parameters* are, and how they differ from interfaces
- How to declare and write *generic functions*, and when that's necessary (and when it's not)
- How generic functions and types are implemented in Go, and how that affects the way we write programs
- How to define and use *constraints* on type parameters, and what constraints are provided in library packages and the Go language itself
- How to write *type element* constraints and *type approximations*
- What *operations* are allowed on parameterized types, and how to choose the right constraints for them
- How to define *generic types*, such as maps, slices, and structs, and how to write *methods* on such types.
- How to write *generic functions*, such as map, reduce, and filter operations on slices, and how to combine *first-class functions* with generics in a useful way.
- How generics support enables us to create some interesting *data structures* such as sets, trees, graphs, heaps, and queues.
- How the standard library is changing with the introduction of generics, and how some popular third-party packages will be affected.

- The answers to many common questions about generics, such as “How will I write code differently?”, and “How will using generics affect the performance of my programs?”

# 1. Generics

*I went to a general store, but they wouldn't let me buy anything specific.*
—Steven Wright



First, it'll be useful to clarify exactly what we mean by "generics", or "generic programming" in general. If you're already familiar with this concept from other languages, and you just want to find out how it works in Go, you can skip this chapter and go straight on to the next.

## Programming with types

Still with me? Great. Let's lay the groundwork a little by talking about *types*.

I'm sure you know that Go has data types: numbers, strings, and so on. Every variable and value in Go has some type, whether it's a built-in type such as `int`, or a user-defined type such as a struct.

The compiler keeps track of these types, and you'll be well aware that it won't let you get away with any type mismatches, such as trying to assign an `int` to a `float64`.

And you've probably written functions, for example, that take some specific type of parameter, such as a string. If we tried to pass a value of a different type, the compiler would complain.

A function that declares a `string` parameter, for example, can accept only `string` values:

```go
func PrintString(s string) {
    fmt.Println(s)
}
```

So that's *specific programming*, if you like: writing functions that take parameters of some specific type. And that's the kind of programming you're probably used to doing in Go.

## Generic programming

What would *generic programming* be, by contrast? It would have to be writing functions that can take either *any* type of parameter, or, more usefully, a *range* of possible types.

The generic equivalent of our `PrintString` function, for example, would be able to print not just a string, but *anything*. What would it look like? What kind of parameter type would we declare?

It's tricky, because something has to go inside the function's parameter list, and we simply don't know what to write there yet:

```go
// not this
func PrintAnything(???) {
// invalid character U+003F '?'
```

We'll find out how to write the real signature for the generic `PrintAnything` function in the next chapter, but first, let's see if there are any other ways we can achieve the same kind of thing in Go.

## Interface types

Go has always had a limited kind of support for functions that can take an argument of more than one specific type, using *interfaces*.

You might have encountered interface types like `io.Writer`, for example:

```go
func PrintTo(w io.Writer, msg string) {
    fmt.Fprintln(w, msg)
}
```

Here we don't know what the precise type of `w` will be at run time (its *dynamic* type, we say), but we at least know something about it. We know that it must *implement* the interface `io.Writer`.

What does it mean to implement an interface? Well, we can look at the interface definition for clues:

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

—io.go

What this is saying is that to be an `io.Writer`—to *implement* `io.Writer`—is to have a particular set of methods. In this case, just one method, `Write`, with a particular signature (it must take a `[]byte` parameter and return `int` and `error`).

This means that more than one specific type can implement `io.Writer`. In fact, any type that has a suitable `Write` method implements it automatically.

You don't need to explicitly declare that your type implements a certain interface. If you have the right set of methods, you implement any interface that specifies those methods.

For example, we can define some struct type of our own, and give it a `Write` method that does nothing at all:

```
type MyWriter struct {}

func (MyWriter) Write([]byte) (int, error) {
    return 0, nil
}
```

We now know that the presence of this `Write` method implicitly makes our struct type an `io.Writer`. So we could pass an instance of `MyWriter` to any function that expects an `io.Writer` parameter, for example.

## Interface parameters

The `MyWriter` type may not be very useful in practice, since it doesn't do anything. Nonetheless, any value of type `MyWriter` is a valid `io.Writer`, because it has the required `Write` method.

It can have *other* methods, too, but the compiler doesn't care about that when it's deciding whether or not a `MyWriter` is an `io.Writer`. It just needs to see a `Write` method with the correct signature.

This means that we can pass an instance of `MyWriter` to `PrintTo`, for example:

```
PrintTo(MyWriter{}, "Hello, world!")
```

If we tried to pass a value of some other type that *doesn't* satisfy the interface, we feel like it shouldn't work:

```
type BogusWriter struct{}
```

```
PrintTo(BogusWriter{}, "This won't compile!")
```

And indeed, we get this error:

```
cannot use BogusWriter{} (type BogusWriter) as type io.Writer
in argument to PrintTo:
    BogusWriter does not implement io.Writer
    (missing Write method)
```

That's fair enough. A function wouldn't take a parameter of type `io.Writer` unless it needed to *call* `Write` on that value.

The compiler can tell in advance that this won't work on a `BogusWriter`, because it doesn't have any such method. So it won't let us pass a `BogusWriter` where an `io.Writer` is expected.

## Polymorphism

What's the point of all this, though? Why not just define the `PrintTo` function to take a `MyWriter` parameter, for example? That is to say, some *concrete* (non-interface) type?

Well, you already know the answer to that: because *more than one* concrete type can be an `io.Writer`. There are many examples in the standard library. For example, all these types implement `io.Writer`:

- `*os.File`
- `*bytes.Buffer`
- `*bufio.Writer`
- `*strings.Builder`
- ...

Now we can see why interfaces are so useful: they let us write very flexible functions. We don't have to write multiple versions of the function, like `PrintToFile`, `PrintToBuffer`, `PrintToBuilder`, and so on.

Instead, we can write one function that takes an interface parameter, `io.Writer`, and it'll work with any type that implements this interface. Indeed, it works with types that don't even exist yet! As long as it has a `Write` method, it'll be acceptable to our function.

The fancy computer science term for this is *polymorphism* ("many forms"). But it just means we can take "many types" of value as a parameter, providing they implement some interface (that is, some set of methods) that we specify.

## Constraining parameters with interfaces

Interfaces in Go are a neat way of introducing some degree of polymorphism into our programs. When we don't care what type our parameter is, so long as we can call certain methods on it, we can use an interface to express that requirement.

It doesn't have to be a standard library interface, such as `io.Writer`; we can define any interface we want.

For example, suppose we're writing some function that takes a value and turns it into a string, by calling a `String` method on it. What sort of interface parameter could we take?

Well, we know we'll be calling `String` on the value, so it *must* have at least a `String` method. How can we express that requirement as an interface? Like this:

```
type Stringer interface {
    String() string
}
```

In other words, any type can be a `Stringer` so long as it has a `String` method. Then we can define our `Stringify` function to take a parameter of this interface type:

```
func Stringify(s Stringer) string {
    return s.String()
}
```

In fact, this interface already exists in the standard library (it's called `fmt.Stringer`), but you get the point. By declaring a function parameter of interface type, we can use the same code to handle multiple dynamic types.

Note that all we can require about a method using an interface is its name and signature (that is, what types it takes and returns). We can't specify anything about what that method actually *does*.

Indeed, it might do nothing at all, as we saw with the `MyWriter` type, and that's okay: it still implements the interface.

## Limitations of method sets

This "method set" approach to constraining parameters is useful, but fairly limited. Suppose we want to write a function that adds two numbers. We might write something like this:

```
func AddNumbers(x, y int) int {
    return x + y
}
```

That's great for `int` values, but what about `float64`? Well, we'd have to write essentially the same function again, but this time with a different parameter and result type:

```
func AddFloats(x, y float64) float64 {
    return x + y
}
```

The actual logic (`x + y`) is exactly the same in both cases, so the type system is hurting us more than it's helping us here.

Indeed, we'd also have to write `AddInt64s`, `AddInt32s`, `AddUints`, and so on, and they'd all consist of the same code. This is boring, and it's not the kind of thing that we became programmers to do.

So we need to think of something else. Maybe interfaces can come to our rescue?

Let's try. Suppose we change `AddNumbers` to take a pair of parameters of some interface type, instead of a concrete type like `int` or `float64`.

What interface could we use? In other words, what methods would we need to specify that the parameter type must implement?

Well, here's where we run into a problem. Actually, `int` *has* no methods, and nor do any of the other built-in types. So there's no method set we can specify that would be implemented by `int`, `float64`, and friends.

We could still define *some* interface: for example, we could require an `Add` method, and then we could define struct types with such a method, and pass them to `AddNumber`. Great. But it wouldn't allow us to use any of Go's *built-in* number types, and that would be a most inconvenient limitation.

## The empty interface

Here's another idea. What about the *empty* interface, `interface{}`? That specifies no methods at all, so literally every concrete type implements it.

Could we use `interface{}` as the type of our parameters? Since that would allow us to pass arguments of any type at all, we might rename our function `AddAnything`:

```
// invalid
func AddAnything(x, y interface{}) interface{} {
    return x + y
}
```

Unfortunately, this doesn't compile:

```
invalid operation: x + y (operator + not defined on interface)
```

The problem here is what we tried to *do* with the parameters: that is, add them. To do that, we used the + operator, and that's not allowed here.

Why not? Because we said, in effect, that x and y can be *any* type, and not every type works with the + operator.

If x and y were instances of some kind of struct, for example, what would it even *mean* to add them? There's no way to know, so the compiler plays it safe by disallowing the + operation altogether.

And there's another, subtler problem here: we presumably need x and y to be the *same* concrete type, whatever it is. But because they're both declared as `interface{}`, we can call this function with *different* concrete types for x and y, which wouldn't make much sense.

## Type assertions and switches

You probably know that we can write a *type assertion* in Go, to detect what concrete type an interface value contains. And there's a more elaborate construct called a *type switch*, which lets us detect a whole set of possible types, like this:

```go
switch v := x.(type) {
case int:
    return v + y
case float64:
    return v + y
case ...
```

Using a `switch` statement like this, we can list all the concrete types that we know *do* support the + operator, and use it with each of them.

This seems promising at first, but really we're just right back where we started! We wanted to avoid writing a separate, essentially identical version of the `Add` function for each concrete type, but here we are doing basically just that. So an interface is no use here.

In practice, we don't often need to write functions like `AddAnything`, which is just as well. But this is an awkward limitation of Go: it makes it difficult for us to write general-purpose library packages, among other things.

Look at the `math` package in the standard library, for example. It provides lots of useful utility functions such as `Pow`, `Abs`, and `Max`... but only on `float64` values.

If you want to use those functions with some other type, you'll have to explicitly convert it to `float64` on the way in, and back to your preferred type on the way out. That's just lame.

## Introducing generics to Go

This isn't full generic programming, then, in the way that we now understand the term. We can't write the equivalent of an `AddAnything` function using just method-based interfaces, even the empty interface.

Or rather, we can write functions that *take* values of any type: we just can't *do* anything useful with those values, like add them together.

At least, that was true until very recently, but now there *is* a way to do this kind of thing. We can use Go generics!

We'll see in the next chapter what that actually involves, but let's take a very brief trip through the history of Go to see how we got where we are today.

> *Go was released on November 10, 2009. Less than 24 hours later we saw the first comment about generics.*
> —https://go.dev/blog/why-generics

Go was deliberately designed to be a very simple language, and specifically to make it easy to compile and build Go programs very fast, without using a lot of resources. As such, the first versions of Go omitted a lot of features programmers have come to expect from modern languages, including generics.

Why didn't they just add generics later, then? Well, there are a couple of compelling reasons.

Go was intended to be quick to learn, without a lot of syntax and keywords to master before you can be productive with the language. Every new thing you add to it is something else that beginners will have to learn.

The Go team also puts a great value on *backwards compatibility*: that is to say, no breaking changes can be introduced to the language. So if you introduce some new syntax, it has to be done in a way that doesn't conflict with any possible existing programs. That's hard!

Various proposals for generics in Go have been made over the years, in fact, but most of them fell at one or another of these hurdles. Some involved an unacceptable hit to compiler performance, or to runtime performance; others introduced too much complexity or weren't backwards compatible with existing code.

## The "contracts" proposal

> *This document is long.*
> —Type Parameters Proposal

In 2019, Go core developers Ian Lance Taylor and Robert Griesemer submitted a draft design proposal for a kind of generics called *type parameters*, introducing a new keyword `contract`. This generated a lot of discussion and criticism, which of course was the point.

By mid-2020, the Go team had incorporated much of this feedback into the draft design, and the proposed syntax had evolved to more or less what we see in Go today.

The biggest change in the 2020 version of the generics proposal was the removal of the `contract` keyword. Instead, it was decided to use the existing keyword `interface` to define contracts, now known as *constraints*, and to extend the syntax of interfaces to allow for this.

A formal theoretical basis for the generics design was written up by Robert Griesemer and others in a 2020 paper titled Featherweight Go, which essentially proved that the idea would work.

Once all the pesky details had been worked out, the draft design became a formal language change proposal, which was eventually accepted:

- https://go.googlesource.com/proposal/+/refs/heads/master/design/43651-type-parameters.md

The job of a document like this is to specify not only the straightforward ways to use generics, but also to nail down the required behaviour of all the weird oddities and edge cases that Go's ingenious users might come up with. It's not exactly light reading, then, but it is at least definitive.

With the basic syntax settled, an experimental tool called go2go was released that could translate generic Go code to "standard Go" for compilation. This allowed Go users to play with type parameters, shake out any bugs or issues, and give feedback on how it worked with their programs.

Go 1.17, released in August 2021, included some experimental support for type parameters, hidden behind a special compiler flag. This was rather buggy and incomplete, though, and shouldn't be used now.

The first Go release with full support for generics was Go 1.18, released in March 2022.

In the next chapter, we'll talk about exactly what it is that's been added to Go, and start writing some generic programs of our own!

## Takeaways

- Support for *generics* in a programming language means being able to write *polymorphic* functions: that is, functions that take parameters of some unspecified arbitrary type.
- Go *interfaces* have always provided a lightweight form of polymorphism, or generics, in that we can write functions that take some set of allowed types based on their methods.
- But interface values introduce an extra layer of indirection, or "boxing": to get at the concrete value inside, you need to use type assertions or type switches.
- Also, interfaces can only specify methods, and not all types *have* methods (none of the built-in types do, for example).
- This also limits what we can *do* with interface values, since Go operators don't call methods; there's no way to express an interface requirement like "works with +", for example.

- Interfaces also don't allow us to specify *relationships* between parameters, such as that two parameters must be of the same type, or that a function's result is the same type as its parameter.
- It's been a long road to getting "real" generics in Go: it took ten years to decide how it should work, and another couple of years to hammer the proposed design into an acceptable shape.
- So if Go's generics support is new to you and you're feeling a bit confused, don't worry—it's new to all of us!

## Review questions

Test your understanding of what you've just read (and improve your retention of it) by briefly answering these review questions. I suggest you write the answers down somewhere: the act of putting your knowledge into words will help to cement it more firmly in your memory.

If you find yourself stuck on any of the questions, it's a good idea to put the book aside for a while, then come back later and read through the chapter again, trying to find the answer. Don't give up!

1. What is a "generic function", in the sense we've used in this discussion? How is it essentially different from an ordinary Go function?
2. What is an interface type in Go? How is it different from a concrete type, such as `int`? Explain the relationship between a value of an interface type and its underlying dynamic type.
3. How does the Go compiler determine whether a given type satisfies a certain interface? To put it another way, what does it *mean* to "satisfy", or synonymously, to "implement" an interface?
4. What are some limitations of polymorphic functions using interface types, as opposed to generic functions? What are some types that *can't* satisfy any method-set interface? What kind of things can't you do with *empty interface* values?
5. Why don't type assertions and type switches, when used with interface values, make generics unnecessary? Conversely, now that we *have* generics, why do we still need interfaces and type switches?

# 2. Type parameters

*I learned very early the difference between knowing the name of something, and knowing something.*
—Richard Feynman

Now that we've described what generic programming is, and how it came to Go, it's time to look at exactly what Go's generics support involves, and what we can do with it.

And now that's easier to explain: it means we can write functions like `PrintAnything` and `AddAnything`!

To be precise, we can write *generic functions* that take parameters not of some specific named type, but of some arbitrary type that we don't have to specify in advance. Let's call it T, for "type".

When the function is actually *called* in our program, T will be some specific type, such as `int`. But we don't want to have to specify that in advance when we're writing the function, so we're just going to use T as a placeholder for whatever the type ends up being.

In fact, there might be more than one T in our finished program. For example, we might call our generic function with a `float64` value, in which case T will be `float64`. But elsewhere we might call the same function with a `string` value, in which case T will be `string`.

This T placeholder is called a *type parameter*. A generic function in Go, then, is a function that takes a type parameter. So what does that look like?

## Generic functions

Let's take the simplest imaginable case first. We'll write a function that takes a type parameter T, where T is a placeholder for any type.

Here's the signature of our `PrintString` function from the previous chapter:

```
func PrintString(s string) {
```

We can now use the type parameter syntax to write a new version called `PrintAnything`, whose argument is of type T, where T is any type:

```
func PrintAnything[T any](v T) {
```

This might look confusing at first, especially if you're new to Go, so let's break it down. We still have the familiar function name and the usual list of parameters in parentheses, but we've inserted something new in the middle:

```
[T any]
```

## Type parameters

What does this mean? We can read this function signature as saying something like:

> For any type T, `PrintAnything[T]` takes a T parameter (that is, a parameter whose type is T), and returns nothing.

While v is just an ordinary parameter, of some unspecified type, T is different. T is a new kind of parameter in Go: a *type parameter*.

We say that `PrintAnything` is a *parameterized* function, that is, a generic function *on* some type T. For short, we usually just talk about `PrintAnything[T]`, pronounced "`PrintAnything` *of* T".

## Instantiation

What type *is* T, specifically? It depends what we decide to pass to the function when it's called.

Suppose we call it with an `int` argument, for example. We give the name of the function, as usual, and we select which particular T we want by putting it inside square brackets after the name:

```
var x int = 5
PrintAnything[int](x)
// 5
```

This is called *instantiating* the function. In effect, the generic PrintAnything function is like a kind of template, and when we call it with some specific type, we create a specific *instance* of the function that takes that type.

We can imagine the compiler seeing this call to PrintAnything[int](x) and thinking "Aha! Now I know what T is: it's int. So I'll compile a version of PrintAnything that takes an int parameter". And that's what happens.

In fact, we usually don't need to explicitly instantiate the function by giving the type name in square brackets. Where the compiler can *infer* this type from context, we can leave it out, so this looks just like an ordinary non-generic function call:

```
var x int = 5
PrintAnything(x)
// 5
```

What if we have another call to PrintAnything[T] somewhere else in the program, and this time T is a different type, such as string? Well, that's okay. The compiler will produce another version of PrintAnything, this time one that takes a string argument.

For every distinct instantiation of a generic function in a particular program, then, the compiler will produce a distinct implementation of the function, that takes the required concrete type as its parameter.

## Stencilling

This approach to implementing generics is called *stencilling*, which is a rather apt name. You can imagine the compiler spray-painting a bunch of similar versions of the function, all using the same "stencil", and differing only in the type of the parameter they take.

We could have done the same thing ourselves using the existing *code generation* machinery in Go, and indeed many people did do exactly that before the introduction of generics.

This makes for efficient machine code, because there's no indirection (unlike with interface values). We don't need type assertions, because each different implementation of PrintAnything knows exactly what concrete type it's getting.

This isn't a particularly compelling example of generics in action, though, because it was already possible to write PrintAnything using an empty interface parameter. We can just pass it straight to fmt.Println, which also takes interface{}.

## An `Identity` function

Let's look at a slightly more interesting, though still rather contrived, example.

Suppose we want to write a function called `Identity` that simply returns whatever value you pass it. How could we write that?

This is where we start to go beyond the limits of interfaces. Using `interface{}`, for example, we'd have to write something like:

```
func Identity(v interface{}) interface{} {
    return v
}
```

This works, but it isn't really satisfactory. As we saw with `AddAnything` in the previous chapter, we don't have any way to tell the compiler that the function's parameter and its result must be the *same* concrete type, whatever it is.

Now we know how to do that, using a type parameter:

```
func Identity[T any](v T) T {
    return v
}
```

Remember how to read this?

> *For any type T, `Identity[T]` takes a T parameter, and returns a T result.*

## Instantiating `Identity`

Suppose we call this function somewhere in our program with a string argument:

```
fmt.Println(Identity("Hello"))
// Hello
```

You now know how this works. Under the hood, the compiler generates an instantiated version of `Identity` that takes a string parameter and returns a string result. This is just a plain, ordinary Go function that we could have written ourselves, or generated mechanically.

The point is, of course, that we don't need to supply a separate version of `Identity` for each concrete type that we want to use. Instead, we just write it once for some arbitrary type T, and the compiler will automatically generate a version of `Identity` for each type that's actually used in our program.

So much for concrete types: could we pass `Identity` a value of some interface type instead, like `io.Reader`? Let's try:

```
buf := bytes.NewBufferString("Hello")
fmt.Println(Identity(io.Reader(buf)))
// Hello
```

What about other interfaces, such as `error`?

```
err := errors.New("oh no")
fmt.Println(Identity(err))
// oh no
```

This makes sense, since `T any` means any type is allowed, including interface types.

## Exercise 2.1: Hello, generics

Now it's over to you to write your first generic function in Go! Let's work through it together, step by step.

First of all, make sure you've checked out a copy of the GitHub repo for this book:

- https://github.com/bitfield/kg-generics

Open the `exercises/2.1` folder in your code editor and take a look at the `print_test.go` file.

You'll find this test:

```
t.Parallel()
buf := &bytes.Buffer{}
print.PrintAnythingTo[string](buf, "Hello, world")
want := "Hello, world\n"
got := buf.String()
if !cmp.Equal(want, got) {
    t.Error(cmp.Diff(want, got))
}
```

First, run the `go mod tidy` command to download the necessary modules for this test. Once that's completed, you're ready to start the exercise.

Run the test using your editor, or the `go test` command. You'll find right now the test doesn't compile, because the required function doesn't exist:

```
undefined: print.PrintAnythingTo
```

Remember, you'll need at least Go version 1.18 to be able to use generics, including running this test and implementing the function to make it pass. That's because we're using some new syntax that doesn't exist in Go version 1.17 and earlier.

If you try to compile this code with a version of Go that doesn't support generics, you'll get a rather confusing additional error message:

```
type string is not an expression
```

So if you see this, you need to upgrade your Go. Follow the instructions in the introduction to this book to do that. Now read on!

To make this compile, you'll need to define a generic function in the `print` package named `PrintAnythingTo` that takes one parameter of type `io.Writer`, and another value that's of some unspecified type.

In other words, `PrintAnythingTo` will have a type parameter `T`, which can be any type, and it takes a parameter of this type T, just like `Identity`. But unlike `Identity`, it also takes another parameter, which is of type `io.Writer`.

To make the test *pass*, your function will need to write the supplied value to the supplied writer. It's up to you how to do this, but you might like to use `fmt.Fprintln`, like our `PrintTo` example in the previous chapter.

The necessary `go.mod` and `print.go` files are already set up for you. All you need to do is edit `print.go` and add the `PrintAnythingTo` function, then run the test again.

The main point of this exercise is to get you up and running writing generics in Go, and to get some practice declaring functions that take type parameters. When the test passes, you're done, so you can move on to the next section.

If you get stuck, you can take a look at my suggested answer in Solution 2.1.

## Composite types

So far, we've figured out how to define a generic function that, for some T, takes a parameter of type T:

```
func Identity[T any](v T) {
```

So is that it? Are we restricted to declaring only parameters of type T itself, or could we also take some *composite type*? That is, some type *involving* T, not just T itself? For example, a *slice* of T?

We could indeed. Suppose we wanted to write a function `Len` that returns the length of a given slice. And its parameter will always be a slice of *something*: that is, of some arbitrary element type. Let's call it `E`, for "element".

The signature of `Len`, then, might look something like this:

```
func Len[E any](s []E) int {
```

See if you can read the signature of this function out loud. We might say, for example:

> *For any type E, `Len[E]` takes a slice of E, and returns `int`.*

By a "slice of E" here, we mean a slice whose elements are all of type E.

But we can also use a type parameter in other kinds of composite type. For example, we can write a generic function on a *channel* of some element type E:

```go
func Drain[E any](ch <-chan E) {}
```

For any type E, `Drain[E]` takes a receive-only channel of E, and returns nothing.

We could even write a *variadic* function: that is, a function that takes a variable number of arguments. In this case, it would take a variable number of channels of E:

```go
func Merge[E any](chs ...<-chan E) <-chan E {
```

For any type E, `Merge[E]` takes any number of receive-only channels of E, and returns a receive-only channel of E.

## Generic types

Generic functions are great, but we can do more. We can also write generic *types*. What does that mean?

We often deal with *collections* of values in Go. For example, consider this slice type:

```go
type SliceOfInt []int
```

You already know that, just as an `int` variable can only hold `int` values, a `[]int` slice can only hold `int` elements. We wouldn't want to have to also define `SliceOfFloat`, `SliceOfString`, and so on.

## A generic slice type

Could we write a *generic* type definition that takes a type parameter, just like a generic function? For example, could we make a slice of any type?

Yes, we could:

```go
type Bunch[E any] []E
```

In other words:

> *For any type E, a `Bunch[E]` is a slice of E.*

Just as we could define a function such as Len that takes a slice of an arbitrary element type E, we can also define a new *type* that's a slice of E.

And just as with generic functions, a generic type is always *instantiated* on some specific type when it's used in a program. That is to say, a `Bunch[int]` will be a slice of `int`, a `Bunch[string]` will be a slice of strings, and so on.

Each of these is a distinct concrete type, as you'd expect, and we can write a `Bunch` literal by giving the type we want in square brackets:

```
b := Bunch[int]{1, 2, 3}
```

## There are no generic types

It's very important to understand that, even though a Bunch is defined as a slice of E for any type E, any *particular* Bunch can't contain values of *different* types.

For example, we can't create a Bunch of one type, and then try to append a value of a different type:

```
b := Bunch[int]{1, 2, 3}
b = append(b, "hello")
// cannot use "hello" (untyped string
// constant) as int value in argument
// to append
```

We can have a `Bunch[int]` *or* a `Bunch[string]` *or* a Bunch of elements of any other specific type. What we can't have is a Bunch of *mixed-type* elements.

It's easy to hear a term like "generic slice" and jump to the conclusion that it means "slice containing values of different types". But that's actually not the case.

There are, in fact, *no generic types in Go*. I know that sounds crazy, but stick with me.

That is, you can *define* generic types like `Bunch[E]`, but to actually use them in your program, you need to *instantiate* them on some specific type, like `int`.

At that point, what you have is an ordinary Go slice of `int`, and it stands to reason that such a slice can only contain `int` elements.

So just because we used a type parameter, it doesn't mean we can create a single slice that contains elements of different types. What we can create are different *slice types*, such as `[]int`, or `[]string`, without having to specify their element type in advance.

One way to express this is to say that, while we can define generic types at compile time, there are no generic types *at run time*.

There's one partial exception to this, which you're already familiar with: interface types. We could always create a slice of `interface{}` in Go, and populate it with elements of different dynamic types. But, for the reasons we've discussed, this isn't the same thing as a truly generic type.

## Exercise 2.2: Group therapy

Over to you again now, to try your hand at Exercise 2.2. This time, you'll need to define a generic slice type `Group[E]` to pass the following test:

```
t.Parallel()
got := group.Group[string]{}
got = append(got, "hello")
got = append(got, "world")
want := group.Group[string]{"hello", "world"}
if !cmp.Equal(want, got) {
    t.Error(cmp.Diff(want, got))
}
```

If you get stuck, have a look at Solution 2.2.

## Generic function types

You probably know that *functions are values* in Go. That is, you can pass a function as an argument to another function, you can *return* a function from a function, and you can declare variables or struct fields of a certain function type.

So what if that function were generic? For example, the `Identity[T]` function in our earlier example. How would we declare a variable to which we could assign the function `Identity[T]`? What would be the type of such a variable?

Well, there's actually no such function as `Identity[T]`, only one or more *instantiations* of that function on a specific type, such as `string`. As we've seen, the `Identity[T]` function definition is just a kind of "stencil" that the compiler uses to produce *actual* functions.

So there *is* such a function as `Identity[string]`, and accordingly we can use it as a value. For example, we can assign it to a variable:

```
f := Identity[string]
```

What is the type of the variable `f`, then? Well, it's whatever the type of `Identity[string]` is. Here's the signature of the generic `Identity` function again:

```
func Identity[T any](v T) T {
```

So if T is `string` in this case, then we feel the type of `Identity[string]` should be:

```
func(string) string
```

Let's ask Go itself. Conveniently, the `fmt` library can report the type of a value, using the `%T` verb. So we'll see what type it thinks `f` is in this case:

```
f := Identity[string]
fmt.Printf("%T\n", f)
// func(string) string
```

Just as with generic types, then, *there are no generic functions in Go*. Any generic functions you may write will in fact be instantiated on some specific type at compile time, and they will then just be plain old functions, like `func(string) string`.

Couldn't we define a *generic function type*, though? For example, could we write something like:

```
type idFunc func[T any](T) T
// syntax error: function type must have no type parameters
```

Nope. If you think about it, how could the compiler compile this? It couldn't, because *all type arguments must be known at compile time*.

What we *can* write is this:

```
type idFunc[T any] func(T) T
```

This is fine, because it's a generic type, just like the ones we've already seen. For some T, we're saying, an `idFunc[T]` is a function that takes a T and returns a T.

If this is ever instantiated in our program, it will become some specific type like `func(int) int`. If not, the compiler can safely ignore it. Either way, no unknown types are involved.

## Generic types as function parameters

An intriguing possibility may have already occurred to you. Could we write a generic function that takes a *parameter* of a generic type? Absolutely.

For example, suppose we want to write a function that takes a `Bunch[E]` as a parameter. We can write, straightforwardly:

```
func PrintBunch[E any](v Bunch[E]) {
    fmt.Println(v)
}
```

We're saying:

*For any type E, `PrintBunch[E]` takes a `Bunch[E]` and returns nothing.*

This would be a parameterized *composite* type, just like when we wrote functions like Len that took a slice of E, for example. And it works as expected:

```
b := Bunch[int]{1, 2, 3}
PrintBunch(b)
// [1, 2, 3]
```

This is pretty exciting stuff! But there's a limit to what we can do with generic functions that take literally *any* type. And we'll run into that limit pretty soon.

## Exercise 2.3: Lengthy proceedings

Now it's your turn to write a generic function on a generic type, to solve Exercise 2.3.

Your task is to implement a generic function `Len[E]` that returns the length of a given slice. As usual, the supplied test will tell you when you've got it right!

Have fun, and if you run into difficulties, check out Solution 2.3.

## Constraining type parameters

We saw in the previous chapter that one of the limitations of interface types in Go is that we can't use them with operators such as `+`. Remember our `AddAnything` example?

You might be wondering if the same kind of limitation applies to functions on `T any`, and you'd be right. If we try to write a generic `AddAnything[T any]`, that doesn't work:

```
func AddAnything[T any](x, y T) T {
    return x + y
}
// invalid operation: operator + not defined on x (variable
// of type T constrained by any)
```

The compiler is always right, but it can sometimes express itself in a rather terse way, so let's unpack that error message a little.

It's complaining about this line:

```
return x + y
```

And it's saying:

```
operator + not defined on x
```

To put it another way, the compiler has no way to guarantee that whatever specific type `T` happens to be when the function is instantiated, that type will work with the `+` operator.

It *might*, but then again, it might not, because:

```
variable of type T constrained by any
```

In other words, because `x` can be *any* type, there's no way to know if it's one of the types that supports the `+` operator. So this is just the same kind of problem as when we tried to add together two `interface{}` values.

If `x` and `y` were some struct type, for example, that certainly wouldn't work: structs don't support `+`, because it's not meaningful to add two structs together. The `+` operator isn't *defined* on structs, we say.

The compiler is telling us that we can't write the expression `x + y` with values of literally *any* type T: that's too broad a range of possible types.

It might be that, in a given program, we only ever instantiate `AddAnything` on types that *do* happen to support `+`, such as `int` or `string`. But that's not good enough for the compiler: we need to *guarantee* that it can't be instantiated on some inappropriate type.

To do this, we need to *constrain* T somehow. That is, to restrict the allowed possibilities for T to only those types that support the operator we want to use. And we'll see how to do that in the next chapter.

## Takeaways

- A generic function in Go is one where we don't specify the precise type of one or more of its parameters.
- Instead, we give a *type parameter* (conventionally called T, but we can use any name) that acts like a placeholder for the type.
- When we call the function with some specific type, such as `int`, the generic function is thus *instantiated* on that type: a version of it is generated that takes an `int` parameter, for example.
- For every distinct instantiation of the function on some specific type, the compiler will produce a distinct implementation of it that will be called at run time.
- Generic functions in Go, then, only exist at *compile time*, not at run time: indeed, experimental support for generics in earlier versions of Go was provided by a source-code translation tool.
- Generic functions can be instantiated on interface types (for example, `error`) as well as concrete types.
- We can also define generic *types*, using a type parameter: for example, the generic slice type `[]E`, which is a slice of some unspecified element type E.
- Type parameters must have a *constraint*, specifying the range of possible types that can be substituted for them: for example, the constraint any allows any type at all.
- But the any constraint on a type parameter doesn't allow us to use Go operators, such as `+`, with values of that type.

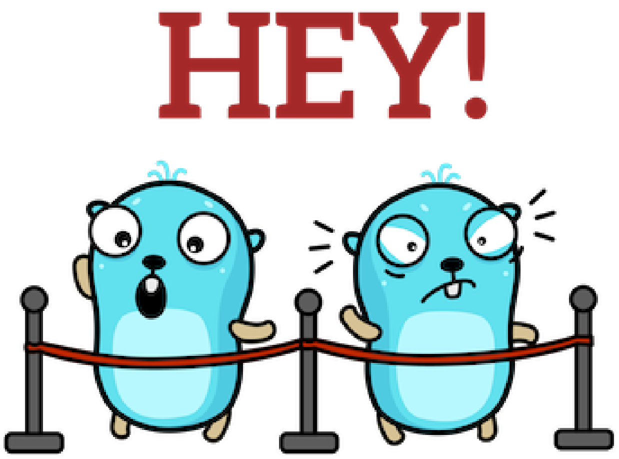
## Review questions

1. What is a type parameter? Give an example of both a parameterized function and a parameterized type.

2. Explain what's meant by "instantiating" a generic function. Give an example.
3. When the author says, in a book on generic types and functions in Go, that "there are no generic types or functions in Go", what does he mean? Is he drunk, or crazy? If not, what is he talking about?
4. Explain the term "type inference". Give examples of when it's possible, and when it's not.
5. Given the parameterized function `Either[T any](p, q T) T`, what would be the function type of `Either[int]`? In other words, if you assigned `Either[int]` to some variable, what would be the type of that variable, as reported by `fmt.Printf("%T", f)`? What about `Either[string]`?

# 3. Constraints

*Design is the beauty of turning constraints into advantages.*
—Aza Raskin



We saw in the previous chapter that when we're writing generic functions that take any type, the range of things we can *do* with values of that type is rather limited. For example, we can't add them together. For that, we'd need to be able to prove to the compiler that they're one of the types that support the + operator.

## Limitations of the `any` constraint

It's the same with interfaces, as we discussed in the first chapter. The empty interface, `interface{}`, is implemented by every type, and so knowing that something implements `interface{}` tells you nothing distinctive about it.

Similarly, in a generic function parameterized by some type T, constraining T to `any` doesn't give the compiler any information about it. So it has no way to guarantee that a given operator, such as +, will work with values of T.

Indeed, `any` is an *alias* for `interface{}` in Go. They both describe the same constraint: that is, no constraint at all.

A Go proverb says:

*The bigger the interface, the weaker the abstraction.*

—https://go-proverbs.github.io/

And the same is true of constraints. The broader the constraint, and thus the more types it allows, the less we can guarantee about what operations we can do on them.

There *are* a few things we can do with literally any type, as you already know, because we've done them. For example, we can declare variables of that type, we can assign values to them, we can return them from functions, and so on.

But we can't really do a whole lot of *computation* with them, because we can't use any operators. So in order to be able to do something useful with values of T, such as adding them, we need more restrictive constraints.

What *kinds* of constraints could there be on T? Let's examine the possibilities.

## Method sets

One kind of constraint that we're already familiar with in Go is an *interface*. In fact, all constraints are interfaces, but let's use the term *classic interface* here to avoid confusion. A classic interface, we'll say, is one that contains only method elements.

For example, the `fmt.Stringer` interface we saw in the opening chapter:

```go
type Stringer interface {
    String() string
}
```

And you're probably familiar with using interfaces like this in pre-generics Go, but it makes perfect sense that we can use classic interfaces as type constraints, too.

For example, we could write a generic function parameterized by some type T that implements the `fmt.Stringer` interface:

```go
func Stringify[T fmt.Stringer](s T) string {
    return s.String()
}
```

This is straightforward, and it works the same way as the generic functions we've already written. The only new thing is that we used the constraint `Stringer` instead of `any`. Instead of the empty interface, in other words, we constrained T to an interface that *isn't* empty!

What would happen, then, if we tried to call `Stringify` with an argument that *doesn't* implement `Stringer`? We feel instinctively that this shouldn't work, and it doesn't:

```go
fmt.Println(Stringify(1))
// int does not implement Stringer (missing method String)
```

That makes sense. It's just the same as if we wrote an ordinary, non-generic function that took a parameter of type `Stringer`, as we did in the first chapter.

There's no advantage to writing a generic function in this case, since we can use this interface type directly in an ordinary function. All the same, a method set interface is a valid constraint for type parameters.

## Exercise 3.1: Stringy beans

Flex your generics muscles a little now, by writing a generic function constrained by `fmt.Stringer` to solve Exercise 3.1.

Your job here is to write a generic function `StringifyTo[T]` that takes an `io.Writer` and a value of some arbitrary type constrained by `fmt.Stringer`, and prints the value to the writer.

If you're not sure what to do, peek at Solution 3.1 for hints.

## Named types

We've seen that one way an interface can specify an allowed range of types is by including a *method element*, such as `String() string`.

Another way to constrain type parameters is with an interface containing a *type element*. This also specifies an allowed range of types, and one way to do that is by simply *naming* those types!

For example, suppose we wanted to write some generic function `Double` that multiplies a number by two, and we want a type constraint that allows only values of type `int`.

How can we write that constraint? Here's what it looks like:

```
type OnlyInt interface {
    int
}
```

Very straightforward! It looks just like a regular interface definition, except that instead of method element, it contains a single type element, consisting of a named type. In this case, the named type is `int`.

How would we use a constraint like this? Let's write `Double`:

```
func Double[T OnlyInt](v T) T {
    return v * 2
}
```

In other words, for some `T` that satisfies the constraint `OnlyInt`, `Double` takes a `T` parameter and returns a `T` result.

Note that we now have one answer to the problem of how to enable the + operator in a parameterized function. Since T can only be `int` (thanks to the `OnlyInt` constraint), the compiler can guarantee that the + operator will work with T values.

It's not the *complete* answer, since there are other types that support + that wouldn't be allowed by this constraint. And in any case we could have written an ordinary function that took an `int` parameter, and got the same result.

So we'll need to be able to expand the range of types allowed by our constraint a little, but not beyond the types that support `+`. How can we do that?

## Unions

What types *can* satisfy the constraint `OnlyInt`? Well, only `int`! To broaden this range, we can create a constraint specifying more than one named type:

```go
type Integer interface {
    int | int8 | int16 | int32 | int64
}
```

The types are separated by the pipe character, `|`. You can think of this as representing "or".

This kind of interface element is called a *union*. A union can include any Go types, including interface types.

It can even contain other constraints. In other words, we can *compose* new constraints from existing ones, like this:

```go
type Float interface {
    float32 | float64
}

type Complex interface {
    complex64 | complex128
}

type Number interface {
    Integer | Float | Complex
}
```

## Type sets

The *type set* of a constraint is the set of all types that satisfy it. The type set of the empty interface (any) is the set of all types, as you'd expect.

The type set of a union element (such as `Float` in the previous example) is the union of the type sets of all its terms.

In the `Float` example, which is the union of `float32 | float64`, its type set contains `float32`, `float64`, and no other types.

## Composite type literals

A *composite* type is one that's built up from other types. We saw some composite types in the previous chapter, such as `[]E`—a slice of some element type `E`.

But we're not restricted to defined types with names. We can also construct new types on the fly, using a *type literal*: that is, literally writing out the type definition as part of the interface.

For example, this interface specifies a *struct* type literal:

```
type Pointish interface {
    struct{ X, Y int }
}
```

A type parameter with this constraint would allow any instance of such a struct. In other words, its type set contains exactly one type: `struct{ X, Y int }`.

## Access to struct fields is not allowed

It's worth knowing, by the way, that even though we can *define* a constraint based on some struct type, we can't do something like this:

```
func GetX[T Pointish](p T) int {
    return p.X
}
// p.X undefined (type T has no field or method X)
```

In other words, we can't refer to a field on `p`, even though the function's constraint explicitly says that any `p` is guaranteed to be a struct with a field `X`. This is a little puzzling at first, since it seems like something we *ought* to be able to do.

However, there are some ambiguities for the compiler that make this problematic. To avoid these issues, access to fields of a struct type parameter is not allowed (in Go 1.18, though this could change in the future). If you're interested in the details, you can read about exactly why this feature was removed in Issue 51576.

## Constraints versus (classic) interfaces

An interface containing type elements can *only* be used as a constraint on a type parameter. It can't be used like a classic interface (that is, one containing only method elements), as the type of a variable or parameter declaration, for example.

But what stops us from doing that? We already know that we can write functions that take ordinary parameters of some classic interface type such as `Stringer`.

So what happens if we try to do the same with an interface containing type elements, such as `Number`?

```
func Double(p Number) Number {
// interface contains type constraints
```

This doesn't compile, for the reasons we've discussed. There is an intriguing hint in the generics proposal that future versions of Go might resolve this anomaly, and allow type constraints to be used as ordinary interfaces. But for now, that's not the case.

Some potential confusion arises from the fact that a classic interface can be used as both a regular interface type *and* a constraint on type parameters. But interfaces that contain type elements can only be used as constraints.

We might express this by saying, whimsically, that all interfaces are constraints, but not all constraints are interfaces. That's not *strictly* accurate, given the newly-expanded meaning of the term "interface", but you get the idea, I hope.

## Constraints are not classes

If you have some experience with languages that have *classes* (hierarchies of types), then there's another thing that might trip you up with Go generics: constraints are not classes, and you can't instantiate a generic function or type on a constraint interface.

To illustrate, suppose we have some concrete types `Cow` and `Chicken`:

```
type Cow struct{ moo string }

type Chicken struct{ cluck string }
```

And suppose we define some interface `Animal` whose type set consists of `Cow` and `Chicken`:

```
type Animal interface {
    Cow | Chicken
}
```

So far, so good, and suppose we now define a generic type `Farm` as a slice of `T Animal`:

```
type Farm[T Animal] []T
```

Since we know the type set of `Animal` contains exactly `Cow` and `Chicken`, then either of those types can be used to instantiate `Farm`:

```
dairy := Farm[Cow]{}
poultry := Farm[Chicken]{}
```

What about `Animal` itself? Could we create a `Farm[Animal]`? No, because there's no such type as `Animal`. It's a type *constraint*, not a type, so this gives an error:

```
mixed := Farm[Animal]{}
// interface contains type constraints
```

And, as we've seen, we also couldn't use `Animal` as the type of some variable, or ordinary function parameter. Only classic interfaces can be used this way, not interfaces containing type elements.

## Limitations of named types

Suppose we've defined some constraint such as `Integer` that consists of a union of named types. Specifically, the built-in signed integer types:

```
type Integer interface {
    int | int8 | int16 | int32 | int64
}
```

We know that the type set of this interface contains all the types we've named. But what about defined types whose *underlying* type is one of the built-in types?

For example:

```
type MyInt int
```

Is `MyInt` also in the type set of `Integer`? Let's find out. Suppose we write a generic function that uses this constraint:

```
func Double[T Integer](v T) T {
    return v * 2
}
```

Can we pass it a `MyInt` value? We'll soon know:

```
fmt.Println(Double(MyInt(1)))
// MyInt does not implement Integer
```

No. That makes sense, because `Integer` is a list of named types, and we can see that `MyInt` isn't one of them.

How can we write an interface that allows not only a set of specific named types, but also any other types *derived* from them?

## Type approximations

We need a new kind of type element: a *type approximation*. We write it using the tilde (~) character:

```
type ApproximatelyInt interface {
    ~int
}
```

The type set of ~int includes int itself, but also any type whose underlying type is int (for example, MyInt).

If we rewrite Double to use this constraint, we can pass it a MyInt, which is good. Even better, it will accept *any* type, now or in the future, whose underlying type is int.

Approximations are especially useful with struct type elements. Remember our Pointish interface?

```
type Pointish interface {
    struct{ x, y int }
}
```

Let's write a generic function with this constraint:

```
func Plot[T Pointish](p T) {
```

We can pass it values of type struct{ x, y int }, as you'd expect:

```
p := struct{ x, y int }{1, 2}
Plot(p)
```

But there's a very serious limitation. We can't pass values of any *named* struct type, even if the struct definition itself matches the constraint perfectly:

```
type Point struct {
    x, y int
}
p := Point{1, 2}
Plot(p)
// Point does not implement Pointish (possibly missing ~ for
// struct{x int; y int} in constraint Pointish)
```

What's the problem here? Our constraint allows struct{ x, y int }, but Point is *not that type*. It's a type *derived* from it. And, just as with MyInt, a derived type is distinct from its underlying type.

You know now how to solve this problem: use a type approximation! And the compiler is telling us the same thing: "Hint, hint: I think you meant to write a ~ in your constraint."

If we add that approximation, the type set of our interface expands to encompass all types derived from the specified struct, including Point:

```
type Pointish interface {
    ~struct{ x, y int }
}
```

## Exercise 3.2: A first approximation

Can you use what you've just learned to solve Exercise 3.2?

Here you're provided with a function IsPositive, which determines whether a given value is greater than zero, and a set of accompanying tests.

The IsPositive function is constrained by some interface Intish, and the tests call it with values of the following type:

```
type MyInt int
```

Your task here is to define the Intish interface. A method set won't work, because int has no methods. On the other hand, the type literal int won't work either, because My-Int is not int. What could you use instead?

If you'd like some clues, have a look at Solution 3.2.

## Intersections

You probably know that with a classic interface, a type must have *all* of the methods listed in order to implement the interface. And if the interface contains other interfaces, a type must implement *all* of those interfaces, not just one of them.

For example:

```
type ReaderStringer interface {
    io.Reader
    fmt.Stringer
}
```

If we were to write this as an *interface literal*, we would separate the methods with a semicolon instead of a newline, but the meaning is the same:

```
interface { io.Reader; fmt.Stringer }
```

To implement this interface, a type has to implement *both* io.Reader *and* fmt.Stringer. Just one or the other isn't good enough.

Could we write an interface with *type* elements arranged in a similar way? That is, separated by newlines (or, equivalently, semicolons)?

Let's try:

```
type IntStringer interface {
    ~int
    fmt.Stringer
}
```

Or, equivalently:

```
interface { ~int; fmt.Stringer }
```

To implement this interface, a type must implement *both* `~int` and `Stringer`. And that's possible, isn't it?

The built-in `int` type doesn't have a `String` method, so `int` itself couldn't satisfy this constraint. But because we used the approximation token `~`, types *derived* from `int` would be okay, as long as they have a `String` method.

Each line of an interface definition like this, then, is treated as a distinct type element. The type set of the interface as a whole is the *intersection* of the type sets of all its elements. That is, only those types that all the elements have in common.

So putting interface elements on different lines has the effect of requiring a type to implement *all* those elements. We don't need this kind of interface very often, but we can imagine cases where it might be necessary.

On the other hand, intersections also make it perfectly possible to define *unsatisfiable* constraints, which aren't useful. For example:

```
type Unpossible interface {
    int
    string
}
```

Clearly no type can be both `int` and `string` at the same time! Or, to put it another way, this interface's type set is empty.

If we try to instantiate a function constrained by `Unpossible`, we'll find, naturally enough, that it can't be done:

```
cannot implement Unpossible (empty type set)
```

## The `constraints` package

In order to write generic functions using the + operator, for example, we now know how to specify appropriate constraints. For example, we could include all the built-in signed and unsigned integer types, and types derived from them:

```
type Integerish interface {
    ~int | ~int8 | ~int16 | ~int32 | ~int64 | ~uint | ~uint8 |
        ~uint16 | ~uint32 | ~uint64
}
```

There are a couple of problems with this, though. One is that we wouldn't want to have to include it in every program using a generic function on integers.

Another is that it's not future-proof. If a new integer type were introduced to Go (such as `int128`, which has been proposed), our existing constraints wouldn't allow it. People wouldn't be able to use the new type with our library unless we explicitly updated it to include `int128`.

Fortunately, there's a new official package called `constraints` which solves this problem. It's not (yet) in the Go standard library, so it currently lives in the official "experimental" repo, called `golang.org/x/exp`. We'll have more to say about this repo and its contents later in the book, in the chapter on libraries.

For example, `constraints.Integer` specifies the same type set as our hand-rolled `Integerish`. So we don't have to write that constraint ourselves, which is nice, and if any new integer types are ever added to Go, they'll be included in `constraints.Integer` too.

There are some other handy constraints defined in the `constraints` package, including some of those we've already come up with:

- `Signed` / `Unsigned`
- `Integer` / `Float`
- `Complex` / `Ordered`

If you're wondering what `Ordered` is, you'll find out in the next chapter. Before that, though, let's look at a few other interesting ways of writing constraints.

## Constraint literals

Up to now, we've always used type parameters with a *named* constraint, such as `Integer` (or even just any). And we know that those constraints are defined as interfaces. So could we use an *interface literal* as a type constraint?

An interface literal, as you probably know, consists of the keyword `interface` followed by curly braces containing (optionally) some interface elements. For example, the simplest interface literal is the empty interface, `interface{}`.

As we saw earlier, any is an alias for `interface{}`, so we should be able to use an empty interface literal wherever any is allowed:

```
func Identity[T interface{}](v T) T {
```

And so we can. But we can do more. We're not restricted to *empty* interface literals. We could write an interface literal containing a method element, for example:

```
func Stringify[T interface{ String() string }](s T) string {
    return s.String()
}
```

This is a little hard to read at first, perhaps. But we've already seen this exact function before, only in that case it had a *named* constraint `Stringer`. We've simply replaced that name with the corresponding interface literal:

```
interface{ String() string }
```

## Omitting `interface`

And we're not limited to just method elements in interface literals used as constraints. We can use type elements too:

```
[T interface{ ~int }]
```

Conveniently, in this case we can omit the enclosing `interface { ... }`, and write simply `~int` as the constraint:

```
[T ~int]
```

For example, we could write some function `Increment` constrained to types derived from `int`:

```
func Increment[T ~int](v T) T {
    return v + 1
}
```

However, we can only omit the `interface` keyword when the constraint contains exactly one type element. Multiple elements wouldn't be allowed, so this doesn't work:

```
func Increment[T ~int; ~float64](v T) T {
// syntax error: unexpected semicolon, expecting comma or ]
```

And we can't omit `interface` with method elements either:

```
func Increment[T String() string](v T) T {
// syntax error: unexpected (, expecting comma or ]
```

And we can only omit `interface` in a constraint *literal*. We can't omit it when defining a named constraint. So this doesn't work, for example:

```
type Intish ~int
// syntax error: unexpected ~ in type declaration
```

## Self-referential constraints

If we can write a constraint on some type parameter T as an interface literal, you might be wondering: can we refer to T in the interface itself?

To see why we might need to do that, suppose we wanted to write a generic function `Contains[T]`, that takes a slice of T and tells you whether or not it contains a given value.

And suppose that we'll determine this, for any particular element of the slice, by calling some `Equal` method on the element.

So the constraint for T is going to be an interface containing the method `Equal(T) bool`, let's say.

Can we do this? Let's try:

```
func Contains[T interface{ Equal(T) bool }](s []T,  v T) bool {
```

Yes, this is fine. In fact, using an interface literal is the *only* way to write this constraint. We couldn't have created some *named* interface type to do the same thing. Why not?

Let's see what happens if we try:

```
type Equaler interface {
    Equal(???) bool // we can't say 'T' here
}
```

Because the type parameter T is part of the `Equal` method signature, and we don't *have* T here. The only way to refer to T is in an interface literal inside a type constraint:

```
[T interface{ Equal(T) bool }]
```

At least, we can't write a *specific* interface that mentions T in its method set. What we'd need here, in fact, is a *generic* interface, and we'll see how to define that in the next chapter.

## Exercise 3.3: Greater love

Your turn now to see if you can solve Exercise 3.3.

You've been given the following (incomplete) function:

```
func IsGreater[T /* Your constraint here! */](x, y T) bool {
    return x.Greater(y)
}
```

This takes two values of some arbitrary type, and compares them by calling the `Greater` method on the first value, passing it the second value.

The tests exercise this function by calling it with two values of a defined type `MyInt`, which has the required `Greater` method. So to make these tests pass, you'll need to write an appropriate type constraint for `IsGreater`.

Can you see what to do?

You can check your answer against Solution 3.3.

## Takeaways

- There's very little we can do with values of literally *any* type, so to do more, we need to be able to express more restrictive constraints than the predeclared `any`.
- One way we can constrain type parameters is with an interface containing *method elements*, such as `String() string`.
- We can also express a range of allowed types by giving a *type element*: for example, a single named type, such as `int`.
- We can include a *union* of types in a constraint, with each possibility separated by the vertical bar (`|`) character.
- Interfaces can have multiple elements, separated by newlines or semicolons.
- The *type set* of an element is the set of all the types it describes.
- The type set of an interface is the intersection of the type sets of all its elements.
- As a result, some interfaces are *unsatisfiable*: if an interface contains intersecting elements whose type sets are disjoint, then that's another way of saying that its type set is empty.
- An interface containing only method elements can act as *either* a classic interface *or* a type constraint, but an interface containing type elements can *only* be used as a constraint.
- A *type approximation* element, using the tilde (`~`) character, specifies not just the named type, but also all types *derived* from it.
- The new `constraints` package defines a number of useful and common constraints, including `Signed`, `Unsigned`, `Integer`, `Float`, `Complex`, and `Ordered`.

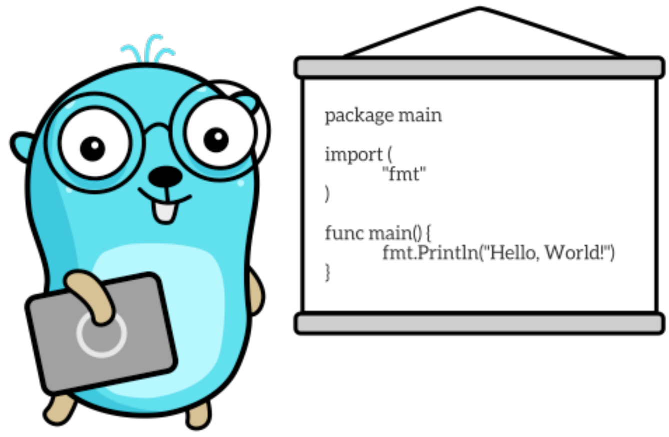
## Review questions

1. "All interfaces can be used as constraints, but not all constraints can be used as interfaces." Explain this statement. What is it about some interfaces that means they can only be used to constrain type parameters, and not as a classic interface?
2. What is a union element of a type constraint, and how do we write it? What is the type set of the union `int | string`?

3. Given a type constraint containing two intersecting elements, such as `interface { io.Reader; fmt.Stringer }`, is it possible for any type to satisfy this constraint? If so, how? What about `int; float64`?
4. What is a type approximation? How do we write it? What does it let us do that we couldn't do simply by writing a named type, or even a union of named types?
5. Using an interface literal, give the signature for a function `Less` parameterized by some `T`, where `T` is constrained to have a method `Less(T) bool`.

# 4. Operations

*It is not only the programmer's responsibility to produce a correct program but also to demonstrate its correctness in a convincing manner.*
—Edsger Dijkstra



```
package main

import (
        "fmt"
)

func main() {
        fmt.Println("Hello, World!")
}
```

In the previous chapter we looked at different ways to constrain type parameters: method sets, named types, and approximations. And you'll recall that the reason we needed to constrain these types in the first place was so that we could do useful *operations* with values, like adding them together.

So what other operations might we want to do with values of parameterized types, and what constraints would make this possible? Let's take the addition example first.

## Arithmetic

Remember the `constraints.Integer` interface, which we met in the previous chapter? Its type set includes all built-in Go integer types, signed or unsigned, and all types derived from them.

We should be able to use this constraint to write the generic `AddAnything` function we discussed in the first chapter, shouldn't we?

First, we'll need to import the `constraints` package. As we saw in the previous chapter, it's not yet part of the Go standard library, so we have to import it from the "experimental" repository:

```
import "golang.org/x/exp/constraints"
```

Now we can use the various constraints defined in this package, including the one we think will be suitable for `AddAnything`, which is `constraints.Integer`.

Here's one way to write the function, for example:

```
func AddAnything[T constraints.Integer](x, y T) T {
    return x + y
}
```

We're saying that `AddAnything`, for some integer type T, takes two T parameters, and returns a T result. Let's see if it works:

```
fmt.Println(AddAnything(1, 2))
// 3
```

That's reassuring. Of course, it's possible to add together floating-point and complex numbers too, and right now our function can't do that.

Let's write a new constraint composing `Integer`, `Float`, and `Complex`, so that we can use all three type sets:

```
type Number interface {
    constraints.Integer | constraints.Float |
        constraints.Complex
}
```

And let's update `AddAnything` to take this new constraint:

```
func AddAnything[T Number](x, y T) T {
    return x + y
}
```

This looks promising, and the compiler seems happy. Let's try adding two complex numbers, just for fun:

```
fmt.Println(AddAnything(1 + 2i, 3 + 4i))
// (4+6i)
```

## Exercise 4.1: Product placement

Here's another challenge for you: solve Exercise 4.1.

Your job is to define a function `Product`, which will return the result of multiplying its two inputs. For example:

```
fmt.Println(Product(2, 3))
// 6
```

You'll also need to define a suitable type constraint for this function. Make sure it's broad enough to allow all the different numeric types supplied by the tests, but not *so* broad that it doesn't allow you to implement the function!

If you have any trouble, refer to Solution 4.1 for one possible answer.

## Relative magnitude

So much for addition. What else could we do? How about finding the greater of two numbers, for example?

```
func Greater[T Number](x, y T) T {
    if x > y {
        return x
    }
    return y
}
```

This *looks* reasonable, but doesn't compile:

```
invalid operation: cannot compare x > y (operator > not
defined on T)
```

What's going on? We know that at least some of the types in `Number`'s type set support the > operator. For example, integers certainly do, and so do floats.

But if we're going to use the > operator with values of T, then *all* possible types for T must support that operator, and clearly not all of them do. By elimination, it must be `Complex` that's the problem.

Unlike the *real* (non-complex) numbers, it's a mathematical convention that there's no natural ordering of complex numbers. Therefore, they can't be compared with the > operator to see which is greater.

So while the `Number` constraint works fine for addition, we'll need a more restrictive constraint for `Greater`. We'll need one that includes the real numbers (integer and floating-point), but not the complex numbers.

Well, we can write that explicitly:

```
type Real interface {
    constraints.Integer | constraints.Float
}
```

This looks promising, so let's update our `Greater` function to use it:

```
func Greater[T Real](x, y T) T {
```

Let's try it with the integers 1 and 2, to find out which one is bigger. Look away now if you don't want to see the result:

```
fmt.Println(Greater(1, 2))
// 2
```

## Ordering

The > operator, and its friends, isn't limited to just numbers; we can also use it with strings. For example, the string `"b"` is greater than the string `"a"`, from Go's point of view.

So we can make our `Greater` function more widely useful by accepting not only real numbers, but also strings and types derived from `string`. In other words, all types that support the > operator.

Let's call such types `Ordered`. That is to say, there exists some well-defined way to arrange values of each type in order, from smallest to largest.

Here's one way we can write an interface whose type set is the ordered types:

```
type Ordered interface {
    constraints.Integer | constraints.Float | ~string
}
```

Recall that the ~ specifies a type approximation: we're allowing not just `string`, but any type whose underlying type is `string`, too.

Let's update our `Greater` function to use the `Ordered` constraint, then:

```
func Greater[T Ordered](x, y T) T {
    if x > y {
        return x
    }
    return y
}
```

This compiles okay, so we'll try it out with a couple of strings:

```
fmt.Println(Greater("a", "b"))
// b
```

It's becoming clearer why we might want to constrain type parameters in certain ways, isn't it?

We need to restrict the allowed types enough that we can guarantee that they'll support the operator we're using (such as + or >).

On the other hand, we don't want to restrict them *too* much. To make our function as widely usable as possible, we should include *all* the types that support the relevant operator.

Our `Ordered` constraint, then, expresses precisely the set of types that can be used with > and its related operators <, >=, and <=. Very useful!

In fact, it's so useful that it's available as `constraints.Ordered`. So now we know what that's for, and we needn't bother writing our own interface for it every time we use the > operator.

## Multiple type parameters

So far we've seen generic functions with a single type parameter, and you might be wondering if it's possible to write functions with *multiple* type parameters.

Indeed it is, and it works just the way you'd expect. Let's write a version of our `Identity` function that takes values of *two* arbitrary types, and returns them both:

```
func Identity[T, U any](x T, y U) (T, U) {
    return x, y
}
```

This hurts the eyes a little at first, until you get used to reading generic code. Let's parse it carefully:

> *For any types T and U, `Identity[T, U]` takes T and U, and returns T and U.*

That's not actually as complicated as it sounds. Let's see what it looks like in use:

```
fmt.Println(Identity(1, "hello"))
// 1 hello
```

This is fine, but we're rather limited in what we can do with T and U values *together*, because we don't have any idea of the *relationship* between the two types T and U.

For example, we can't find which value is greater, using the > operator. Why not? Because T and U are different types, and that operator can only work on values of the *same* type. For the same reason, we couldn't add them together, or use any other operators.

In functions with multiple type parameters, then, each of those abstract types is distinct to the compiler. Even if they have the same *constraint*, that doesn't guarantee they'll be the same *type*.

That might happen to be the case, by coincidence, but the compiler has no way to guarantee this. It's much more likely that T and U will be two different types, each of which satisfies the constraint in a different way.

For example, `int` and `float64` both satisfy `Ordered`, but they're not the same type, and we couldn't compare them with `>`. So if we want to use an operator like that, we'd better take two values of a *single* arbitrary type, rather than two type parameters.

But there are quite sensible uses for multiple type parameters. For example, consider a function that operates on an arbitrary map type. We could specify one type parameter `K` for the map's keys, and another type parameter `V` for the values.

We'll see how to do that in the next chapter, but first let's look at another very important kind of operation.

## Equality

Probably the most common operation we do with two values in Go programs is checking whether they're *equal*. How could we do that in a generic function?

We might start by trying to use the `any` constraint. Let's try:

```
func Equal[T any](x, y T) bool {
    return x == y
}
// invalid operation: cannot compare x == y (operator == not
// defined on T)
```

Oh dear. It's the same kind of problem we've run into before, isn't it? One does not simply compare two values of any type with the `==` operator.

So `any` is too broad a constraint to allow us to use the `==` operator. We need to specify a smaller set of types.

Well, what about `constraints.Ordered`? Surely every type that's `Ordered` can be used with `==`?

```
func Equal[T constraints.Ordered](x, y T) bool {
    return x == y
}
```

Indeed, and this works:

```
fmt.Println(Equal(1, 2))
// false
```

But wait a minute: have we been *too* restrictive? The point of generic functions is to accept as wide a range of types as possible. The type set of `Ordered` might not include everything we can compare with the `==` operator.

So are there any types that we're missing out with this constraint? That is, types that don't support `>` but *do* support `==`?

## Comparable types

Actually, yes, there are. `bool` is one example. We can compare two `bool` values for equality with `==`, but we can't order them with `>`.

Structs are another example of such *comparable* types. Provided all its fields are comparable, a struct type is also comparable. And there are many more such "comparable, but unordered" types: complex numbers, pointers, channels, arrays, and interfaces.

So for a function like `Equal` that uses the `==` operator, we need a constraint that specifies the full set of comparable types. How could we write that as an interface?

Perhaps surprisingly, it turns out that we can't!

Yes, we could list all the built-in comparable types by name, but what about user-defined structs, interfaces, channels, and so on?

We can't use a method set, because the `==` operator doesn't *call* methods; no operator does. We can't use type elements either, because we can't name every possible struct type, for example.

## The `comparable` constraint

Yet it's clearly very important to be able to compare values using `==`. We do this all the time in Go programs, so not being able to do it in generic functions would be a severe limitation, to say the least.

This is why Go provides a predeclared constraint named `comparable`. It specifies exactly the set of comparable types, which as we've seen we wouldn't be able to define using a Go interface.

It sounds like `comparable` is exactly the type constraint we need for our `Equal` function:

```
func Equal[T comparable](x, y T) bool {
    return x == y
}
```

No errors, so let's try it out:

```
fmt.Println(Equal(1, 1))
// true
```

What a relief!

You might be wondering why `comparable` is a predeclared identifier in Go, rather than being provided in some package. Why isn't it `constraints.Comparable`, for example?

Well, because that package is written in Go, and we already know that we can't *write* `comparable` in Go. It has to be implemented in the compiler, which can already type-check expressions using the == operator to make sure they're allowed. This works the same way.

## Exercise 4.2: Duplicate keys

See if you can use this idea to tackle Exercise 4.2.

The task here is to determine if a given slice contains duplicated elements. For example, if a slice of numbers contains any given number more than once, then the slice contains duplicates. The duplicate elements need not be consecutive.

To do this, you'll need to write a function `Dupes`, with an appropriate type constraint, that returns `true` if the given slice contains duplicates, or `false` otherwise.

For example:

```
fmt.Println(Dupes([]int{1, 2, 3}]
// false
fmt.Println(Dupes([]int{1, 2, 1}]
// true
```

See Solution 4.2 if you'd like something to compare with your answer.

## A `Max` function

Armed with `comparable` and the `constraints` package, we feel we should now be able to write some interesting non-trivial generic functions. Let's try.

We already have a function to find the greater of two values, so what about one to find the greatest element of a *slice* of values?

Let's call this function `Max` (it's a nice name). How should we constrain the element type of the slice it operates on? To answer that question, we need to ask, first, what are we going to *do* with those values?

Finding the biggest element of a slice is a lot like finding the bigger of two values; in fact, that's exactly how we're going to do it.

We'll compare each element in turn with the biggest value we've seen so far, and to do that we'll use the > operator.

So we know straight away what constraint to use, don't we? That's right:

```
func Max[E constraints.Ordered](s []E) E {
    var max E
    for _, e := range s {
        if e > max {
            max = e
        }
    }
    return max
}
```

Let's read this back in English to make sure we got it right:

> *For any ordered type E, `Max[E]` takes a slice of E, and returns E.*

We could try this out with a slice of numbers, but that's a little uninspiring. We know that `Ordered` also includes strings, so we should be able to do something like this:

```
s := []string{"faith", "hope", "love"}
fmt.Println(Max(s))
// love
```

Another important question answered!

## Abstract types

Here's something interesting we haven't seen before. We didn't just use the type parameter E in the *signature* of the `Max` function. We used it in the body too, to declare our result variable:

```
var max E
```

This is the first time we've used a type parameter within the body of a function like this, but it's absolutely fine. We already know we can refer to E in the function's parameter and result lists, and this is no different.

In fact, the *scope* of a type parameter extends as far as the end of the function body. When it's used in the body like this, we call E an *abstract type*.

There's nothing special about an abstract type. When the function is instantiated, it'll just turn out to be some ordinary specific type. In our example, it's `string`.

## The zero value

Sometimes in a Go function we want to return the *zero value* of whatever type we're dealing with (in the error case, for example). How can we do that with an abstract type like E?

Well, we already know that all types in Go have a *default* value, which is what a variable will have if we don't assign anything to it.

That's just another name for the zero value, so using a `var` statement is one way to get a zero E:

```go
var max E
return max
```

This isn't always convenient, so another possible way is to explicitly *convert* the constant 0 to E:

```go
return E(0)
```

Provided that 0 is *representable* in E, this is okay. In other words, if we could convert the constant 0 to every possible type E allowed by the constraint, then we can do this.

That's not always the case, but when it is, `E(0)` is a convenient way to get the zero value.

## Takeaways

- When we need to do arithmetic operations, such as addition, on parameterized types, we can use the handy interfaces defined in the `constraints` package: `Integer`, `Float`, or `Complex`, or some combination of these, as appropriate.
- If we need to find the relative magnitude of two numbers, our constraint must exclude Go's complex number types, because they don't support the > operator: they're not *ordered*, in other words.
- However, strings *are* ordered, which means a function that uses > can meaningfully accept types based on `string` as well as *real* (non-complex) numbers.
- `constraints.Ordered` specifies precisely these ordered types, so that's what we need when using the > operator.
- The set of types with which we can use the == and `!=` operators—the *comparable* types—is larger than `Ordered`, because it includes complex numbers, booleans, pointers, channels, arrays, interfaces, and some structs.
- Indeed, the comparable types can't be represented in Go using interface elements at all, so we use the predeclared identifier `comparable` instead.
- We can use an abstract type T in the body of a function, as well as in its signature; for example, we can declare variables of T.
- And to get the zero value of T, we can use an explicit type conversion: `T(0)`, provided 0 is representable in T.

## Review questions

1. If you want to use the == operator in a generic function, what constraint would you need on the function's type parameter? What about if you wanted to use the >

operator?

2. Which constraint's type set is larger, `comparable` or `constraints.Ordered`? Why? Name some types that are in both sets, some that are in one set but not the other, and some that aren't in either. Which of the two sets, if either, is `string` a member of? What about `complex128`?

3. Why can't the `comparable` constraint be adequately defined using Go, in the `constraints` library? And if it can't, how or where *is* it defined?

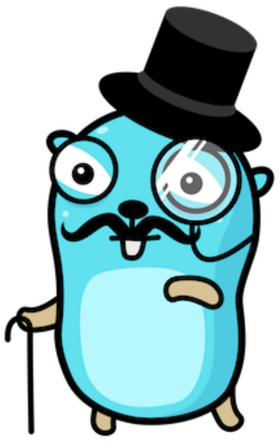
4. Distinguish an abstract type from a concrete type or an interface type.

5. What constraint would you need on the type parameter for the following functions? Write and test the functions to see if you're right.

   - A function that returns `true` if a given value is in a given slice
   - A function that appends a given element to a slice
   - A function that returns `true` if a given value is positive, and `false` if it's zero or negative

# 5. Types

*Just because everything is different doesn't mean anything has changed.*
—Irene Peter



So far in this book we've talked mostly about generic functions. Let's talk a little now about generic *types*.

## Type definitions

You probably know that in Go we can define a new type by using the `type` keyword, like this:

```go
type MyInt int
```

This creates a new type named `MyInt`, which is entirely distinct from any other Go type, including `int`. In other words, if we declared a variable of type `MyInt`, we couldn't then assign an `int` value to it. That would be a compile error.

What use is this, if all we've done is give the type a new name? Well, it turns out giving things names is quite useful! For example, port numbers for web services are integers, so we could imagine creating a type just for port numbers:

```go
type Port int
```

That would mean we can write some method that takes a `Port`, for example, and the compiler can help us check that we're not just passing it random integer values, but only integers that are intended to represent port numbers.

Can we do the same using type parameters? In other words, could we write, for example:

```
type MyT[T any] T
// cannot use a type parameter as RHS in type declaration
```

No, this isn't allowed. It's not that it isn't useful to give new names to existing types, as we saw earlier. It's just that it isn't really useful to give new names to *generic* types such as T. They're generic, so they can be anything. It's not clear how creating a new name for "anything" would help us.

## Slices

So if we can't give new names to a generic type, what can we do? Well, almost everything else. We can, for example, define a slice of elements of some arbitrary type, as we did in an earlier chapter:

```
type Bunch[E any] []E
```

It's conventional, by the way, though not required, to capitalise the names of type parameters. And when we're talking about arbitrary types in general, we conventionally name them T (for "type", in case you hadn't guessed), but when referring to elements of slices, for example, the usual name is E (for "element").

We've also seen that we need to *instantiate* such types when we use them:

```
b := Bunch[int]{1, 2, 3}
```

And this leads into a really important point to remember about generic types. They're always instantiated, in your compiled program, on *some specific type*.

That means that a generic slice type like `Bunch`, for example, *cannot contain elements of different types*. This might seem obvious, or perhaps it doesn't, but the type parameter syntax can easily mislead people into thinking something like:

> *Well, if it's a slice that can contain elements of any type, then I can put a string, an `int`, and a struct into it, right?*
> *…Right?*

Wrong:

```
b := Bunch[int]{1, 2, 3}
b = append(b, "hello")
// cannot use "hello" (untyped string constant) as int value
// in argument to append
```

And *of course* this doesn't work: it's the same as trying to append a string to a slice of `int`. Which is, in fact, exactly what we're doing!

Because there's no such type as a `Bunch`. It looks like there is, but there isn't. What there *is* is a `Bunch[int]`. Or a `Bunch[string]`. Or a `Bunch` *of* any specific type.

## There are no generic types

Remember how we talked about a confusing overlap between interfaces and type constraints? This especially applies to generic types, because we *can* perfectly well write some definition like:

```
type Stuff []interface{}
```

This really *is* a generic slice, if you like: we can put values of any type into it, including values of *different* concrete types. For example:

```
s := Stuff{1, 2, 3}
s = append(s, "hello")
fmt.Println(s)
// [1 2 3 hello]
```

Exactly what we can't do with a so-called "generic" slice type like `Bunch`! So it's critical to be really clear in the way we think and talk about generic types, especially collection types.

Generic types are *not* like collections of empty interface values. Once they're instantiated, they are just perfectly ordinary collections of elements, all of the same specific type.

We can talk about "generic types", then, when we're speaking loosely, but to be absolutely precise, *there are no generic types* in Go. At compile time, all parameterized types will be instantiated on some specific type. At run time, therefore, there are only specific types.

## Instantiating generic types with interfaces

Just to add to the potential confusion, generic types *can* be instantiated with interface types, provided always that the interface in question matches the necessary constraint.

For example, we've instantiated `Bunch` on concrete types like `string` or `int`. But we're not restricted to only concrete types. We could perfectly well instantiate `Bunch` on a slice of values of some interface type, such as `error`:

```
var b Bunch[error]
b = append(b, errors.New("oh no"))
```

```
fmt.Println(b)
// [oh no]
```

So even though `interface{}` is technically a synonym for `any`, it's important to understand why the following two things are fundamentally different:

1. A slice of elements of type `interface{}`
2. A slice of elements of type `E`, where `E` is `any`.

This is a good example of where clarity of terminology matters, because people often talk loosely about "generic container types", for example, meaning something like a slice of `interface{}`, which can contain elements of different concrete types.

That's not what a `Bunch` is, as we've seen. Rather, a `Bunch` is a slice of elements that all have the *same*, arbitrary type.

Which type that *is* will be decided when we actually use the type somewhere in our program: for example, it might be `int`, if we use a `Bunch[int]`.

## Maps

Can we define *maps* involving some arbitrary type, then? For example, a map of `string` to an arbitrary value type `V`?

Certainly we can:

```
type Catalog[V any] map[string]V
```

For any type `V`, a `Catalog[V]` is a map of `string` to `V`.

Let's try it out:

```
cat := Catalog[int]{}
fmt.Println(cat["bogus"])
// 0
```

We've instantiated `Catalog[V]` with the type `int`, creating a map of `string` to `int`.

The result of this is exactly the same as if we'd written something like:

```
type CatalogInt map[string]int
cat := CatalogInt{}
fmt.Println(cat["bogus"])
```

## Multiple type parameters

So far in this book we've used only generic functions and types with just *one* type parameter. We saw in the previous chapter, though, that we can use more type parameters if we need to: for example, when defining a generic map type.

Let's try that now:

```
type Index[K, V any] map[K]V
// invalid map key type K (missing comparable constraint)
```

Hmm, what's gone wrong here? It doesn't like us constraining K by `any`. Actually, that makes sense. We can't define even an ordinary, non-generic Go map with literally *any* key type. Only certain types are allowed.

Why? Think about how maps work. If we store a value in the map with a certain key, of some specific type, we expect to be able to retrieve it again later using the same key. But how can we tell that it *is* the same key unless we can use the == operator with that type?

We can't, so that restricts map keys to only those types that support the == operator. As we've seen, that constraint is described by the predeclared identifier `comparable`, and that's why the compiler complains:

```
missing comparable constraint
```

Whatever constraint we impose on K, it can't allow any types which are not `comparable`. It could be even *more* restrictive, if we want (allowing only integer types, for example), but it doesn't need to be. `comparable` is sufficient for map keys.

If we add that constraint (for K only), then it should work:

```
type Index[K comparable, V any] map[K]V
// this is fine
```

How do we instantiate an `Index` on some specific pair of types, then? We already know how to instantiate generic types with one type parameter: the specific type we want goes in square brackets after the type name.

And it's just the same when we have two type parameters: we give them both inside the square brackets, separated by a comma:

```
age := Index[string, int]{}
```

And, because `age` is just a perfectly ordinary map of `string` to `int`, we can use it in the familiar way:

```
age["John"] = 29
fmt.Println(age["John"])
// 29
```

Reassuring!

## Structs

What about struct types? Could we, for example, define some struct type with a field of arbitrary type T?

```
type NamedThing[T any] struct {
    Name  string
    Thing T
}
```

This seems straightforward, and we can instantiate and use it without any difficulty:

```
n := NamedThing[float64]{
    Name:  "Latitude",
    Thing: 50.406,
}
fmt.Println(n)
// {Latitude 50.406}
```

What if we tried to do something silly here? For example, instantiate a NamedThing where the Thing type itself is NamedThing? Would we disappear into a never-ending recursive hall of mirrors?

That sounds fun, so let's try:

```
var n NamedThing[NamedThing]
// *Inception horn*
```

No, because, as we saw earlier, there's really *no such type as `NamedThing`*. There are only specific *instantiations* of it: for example, `NamedThing[float64]`.

To instantiate the outer `NameThing`, we need to give a specific type in square brackets, so we can't just say `NamedThing`. That's a *generic* type, which is no good:

```
cannot use generic type NamedThing[T any] without instantiation
```

So we have to instantiate the inner `NamedThing`, and that eliminates the recursion:

```
p := NamedThing[NamedThing[float64]]{
    Name: "Position",
    Thing: NamedThing[float64]{
        Name:  "Latitude",
        Thing: 50.406,
    },
}
fmt.Println(p)
// {Position {Latitude 50.406}}
```

Maybe this particular example is a little silly, or maybe not, but it's a good way of making the point, and understanding the rules of generic types. It's quite possible that real applications might want to nest generic types in this way.

## Methods

Now that we know how to define generic types, what about methods on those types? Take the Bunch type, for example, based on a slice. Could we write a `First` method that returns the first element of the bunch? Let's try:

```
type Bunch[E any] []E

func (b Bunch[E]) First() E {
    return b[0]
}
```

It's not hard to *write* the `First` method, but the win for generics is that now we won't have to write it N times for N types. We can write it once, and use it on any kind of Bunch that we choose to instantiate:

```
b := Bunch[string]{"a", "b", "c"}
fmt.Println(b.First())
// a
```

## Exercise 5.1: Empty promises

Have a look at Exercise 5.1 and see if you can solve it, based on what you just learned.

Your job is to create a Sequence type that can hold multiple values, and an `Empty` method on it that will report whether or not the sequence is empty.

For example:

```
s := empty.Sequence[string]{"a", "b", "c"}
fmt.Println(s.Empty())
// false
```

See Solution 5.1 if you'd like something to compare with your answer.

## Zero values

And now that we can use generics as a force multiplier, we can put a bit more thought into the things we multiply with it.

For example, we might want `First` to not panic if the slice is empty.

```
func (b Bunch[E]) First() E {
    if len(b) == 0 {
        return ... // what?
```

What indeed? Perhaps zero would make sense, whatever the zero value of E happens to be.

Let's try that, using the trick we already learned with the Max function in the previous chapter: explicitly converting the constant 0 to E.

```
return E(0)
// cannot convert 0 (untyped int value) to
// type E: E does not contain specific types
```

Say what now? Actually, this *does* make sense. This is a method on a Bunch of some type E. And E is constrained only by any: that is, it can be any type.

The type set of any doesn't "contain specific types", in the compiler's words. So we can't promise that it'll always be possible to convert the constant 0 to type E.

That problem only exists because we're using the any constraint, though. Suppose we used some constraint that *does* contain specific types, such as `constraints.Ordered`. That would be absolutely fine. Zero can be explicitly converted to all `Ordered` types.

But we don't really want to restrict our Bunch to ordered element types, just to be able to do this zero conversion. Let's just replace it with a `var` statement:

```
func (b Bunch[E]) First() E {
    var zero E
    if len(b) == 0 {
        return zero
    }
    return b[0]
}
```

## Methods with type parameters

If we can write methods on generic types, and we already know we can write generic *functions*, then could we write a generic *method* on a generic type?

That is, could we write a method on a type such as `Bunch[E]` that takes a parameter of some arbitrary type, unrelated to E? Let's try an example.

Suppose we wanted to create a method `PrintWith` that prints the contents of the `Bunch[E]` along with some other value of an arbitrary type T. How would we write that?

It turns out that we can't:

```
func (b Bunch[E]) PrintWith[T any](v T) {
    fmt.Println(v, b)
}
// methods cannot have type parameters
```

Well, there's our answer! Methods cannot have type parameters. Sorry about that.

The reasons for this are somewhat technical, and needn't concern us here, but you can read about them in the type parameters proposal.

It's not the fact that `Bunch[E]` is a generic type that's the problem: we can't write parameterized methods on *any* type.

So, to be clear, while we *can* write ordinary methods on parameterized types, we can't write a parameterized method of any kind.

It's possible that this might change in the future, but for now, this is where we are. And there are some other limitations of Go generics which we'll talk about in the final chapter of this book.

## Interfaces

We've looked at generic slices, maps, and structs, but those aren't the only kind of types available in Go. What about interfaces, for example?

Remember our `Equaler` example from the previous chapter, when we had to write our constraint as an interface literal, because it referred to T?

```
func Contains[T interface{ Equal(T) bool }](s []T,v T) bool {
```

We couldn't define a named interface `Equaler`, because it would have to refer to T in its method set, and we don't *have* T at that point!

But it seems completely logical that we should be able to define some *generic* interface type `Equaler[T]`, doesn't it?

In other words, we should be able to write this:

```
type Equaler[T any] interface {
    Equal(T) bool
}
```

Absolutely! For any type T, `Equaler[T]` is an interface specifying a method `Equal` that takes a T parameter and returns `bool`. And that's just what we wanted.

## Channels

Everybody loves a channel, and it would be disappointing if we couldn't create a channel of some (instantiated) generic type.

Let's try:

```go
type myChan[E any] chan E
ch := make(myChan[error])
go func() {
    ch <- errors.New("oh no")
}()
fmt.Println(<-ch)
// oh no
```

Of course we can! And since there are many operations we can do on channels without caring about the element type, this is very useful. For example, the `Drain` and `Merge` examples we saw in an earlier chapter.

## Takeaways

- We can use type parameters in type definitions (provided the type isn't trivially defined as itself, so `type MyT[T any] T` isn't allowed).
- We can, for example, define a type as a slice of elements of some arbitrary type E.
- Importantly, all the elements of such a slice will have the *same* type, whatever it turns out to be; it's not like a slice of `interface{}`, which can contain elements of different (dynamic) types.
- Indeed, there are no "generic types" (except in the sense of `interface{}`): any parameterized type we define has to be instantiated before we can use it.
- However, just to make things complicated, we can instantiate generic types on interface types (for example, a slice of `error`).
- We can use *multiple* type parameters in a single definition: for example, a map of K to V.
- But, since map keys must be comparable, K must be constrained by at least `comparable`.
- We can also define struct types parameterized by some T, where the struct has fields of type T (or some derivative of it, such as `[]T`).
- We can write methods on a generic type (for example, a `First` or even a `Max` method on a slice type).
- But we can't write a generic method (that is, a method that takes a type parameter), because `reasons`.
- It can sometimes be tricky to express the zero value of the T we're dealing with: when the explicit conversion `T(0)` doesn't work, we can fall back on using a `var` declaration or (equivalently) a named result parameter.

## Review questions

1. Can a generic slice type such as `Bunch[E any]` contain elements of mixed types? That is, could a `Bunch` variable contain both an `int` and a `float64`? If not, why not?
2. Can a generic type like `Bunch` contain *interface* types, such as `any`, `error`, or `io.Reader`? If not, why not?
3. Clearly distinguish between a slice of elements of interface type `any`, and a generic slice of elements of type `E`, where `E` is constrained by `any`.
4. Can we define a generic map type such as `Map[K, V any] map[K]V`? If not, why not, and what should we write instead?
5. Which of the following constructs are legal in Go?
   - A method on a specific type
   - A method on a generic type
   - A generic method on a specific type
   - A generic method on a generic type

# 6. Functions

*Most of the arguments for adding generics to Go are to support collection types, sets, tries, etc. That isn't very interesting to me, I'm sure it's interesting to others, but not to me; maps and slices work just fine. What is interesting to me is writing generic functions.*
—Dave Cheney

Now that we've trotted patiently through several chapters explaining what generic functions and types even *mean*, and how they work in Go, let's break into a gentle canter through some *applications* of generics.

After all this fuss, what real-world code can we actually write with type parameters? Are they of any practical use in programs?

## Functions on container types

What did the people who so loudly and persistently campaigned for generics in Go *want*, actually? What was it they needed to write that they couldn't do (easily) without generics?

Well, as we already discussed, it was difficult to write general-purpose functions on Go's built-in *collection*, or *container* types, slices and maps, without generics. Why? Well, because there's no such type as `slice`, for example: every slice type is a slice *of* some element type, such as `int`.

Any non-generic function that takes a slice, then, must take a slice of some *specific* type, which means that we have to duplicate it for every distinct type that we want to handle.

Worse, we can't define some new type (such as `MyInt`) and pass a slice of it to one of these functions. We'd have to implement the function all over again for every such type.

## Contains

One important consequence of generics, then, is the ability to write *libraries* of functions on arbitrary container types, such as slices. For example, without generics it wasn't possible to provide a standard library of common operations on a slice, such as `Contains`: a function that checks if the slice contains a certain element.

Instead, everybody who wanted to *do* that operation in their programs just had to write it anew each time, for the specific slice type they were using.

It's not hard to *implement* `Contains`, as we'll see. It's just that, without generics, it's hard to write a single version of `Contains` that works with multiple slice types.

Happily, that gap is now filled, and we can write a generic `Contains`! Let's try.

```
func Contains[E comparable](s []E, v E) bool {
    for _, vs := range s {
        if v == vs {
            return true
        }
    }
    return false
}
```

For any comparable type `E`, `Contains[E]` takes a slice of `E` and an `E` value, and returns `bool`.

We know that the element type must be constrained by at least `comparable`, because we'll be comparing each element against `v` to see if it's the one we're looking for.

## Reverse

What about other operations on generic slices? We've already written a `Max` function in a previous chapter, and we feel `Min` should also be possible along the same lines.

Other common operations on slices are `Sort` and `Reverse`. Let's tackle `Reverse` first, because it's easier.

```
func Reverse[E any](s []E) []E {
    result := make([]E, 0, len(s))
    for i := len(s) - 1; i >= 0; i-- {
        result = append(result, s[i])
    }
    return result
}
```

For any type E, `Reverse[E]` takes a slice of E and returns a slice of E.

This time, we don't need `comparable`, because we won't be doing any comparisons. All we need to do is loop over the input slice backwards, appending each element to the result slice.

The specific implementation doesn't matter, and this one may not be the most efficient or even the easiest to understand. The point is that this is something we simply couldn't have written at all without generics, without repeating the whole function for every specific element type.

## Sort

Sorting a slice is another example of something which can be a bit laborious without generics.

The standard library's `sort.Slice` API requires users to pass in their own function to determine which of two elements is the lesser:

```
sort.Slice(s, func(i, j int) bool {
    return s[i] < s[j]
})
```

But it seems silly to have to pass in such a function when we're sorting a slice of something perfectly straightforward like `int`, and that's usually the case.

Go knows perfectly well how to compare two integers with <, so why should we have to write a function to do that? Well, you know why, but that doesn't make it any less annoying.

Now we can write a generic Sort function which doesn't have such an awkward API:

```
func Sort[E constraints.Ordered](s []E) []E {
    result := make([]E, len(s))
    copy(result, s)
    sort.Slice(result, func(i, j int) bool {
        return result[i] < result[j]
    })
    return result
}
```

For any ordered type `E`, `Sort[E]` takes a slice of `E` and returns a slice of `E`.

Again, this certainly isn't a model implementation—it uses `sort.Slice` under the hood, which is slow because it uses reflection. That's not necessary with generics, and we could implement some efficient sort algorithm such as Quicksort directly.

The important thing about this `Sort` function is not whether the code is any good—it isn't—but that we can write it straightforwardly for any `Ordered` type. So that unordered types needn't feel left out, we could always provide a version where the user can pass in their own `Less` function, as before.

## First-class functions

As you probably know, *functions are values* in Go. In other words, we can pass a function as an argument to another function, or return one as its result. Indeed, this is a common pattern, and one of the most powerful features of Go.

The technical term for this feature of a language is that its functions are *first-class citizens*: they're on the same level as things like variables and literals, and can be used in the same way.

However, before generics, first-class functions in Go were rather limited by the need to always specify the type of their parameters and results. That meant that it was difficult or impossible to use some of the most desirable features of first-class functions available in other languages.

An example of this kind of feature is *mapping* some function over a collection of elements. That is, applying the function to each element, with the final result being the transformed collection.

For example, if we wanted to double every integer in a slice, we could write some function `double` that doubles its input, and "map" that function over the slice. The result would be a slice where each element is twice the value of the corresponding element in the original slice.

This is nothing to do with Go's `map` type, but it conveys the same idea of mapping one set of values to another; in this case, by applying some arbitrary function to each value.

## Map

A `Map` function is difficult to write in Go without generics, because the signature of that arbitrary function is a problem.

What type of parameter should it declare? We'd have to write a new version of the the function for each element type we needed.

Worse, we'd have to write a separate version of the `Map` function as well, because *it* has to declare the arbitrary function as a parameter.

This is all much easier now. Let's first of all work out what the type of the arbitrary function needs to be.

Well, because it transforms a slice of E to another slice of E, it makes sense that it has to take E and return E:

```
type mapFunc[E any] func(E) E
```

For any type E, a `mapFunc[E]` is a function that takes an E parameter and returns an E result.

Now we can write a generic function that takes a `mapFunc[E]` and applies it to every element of a slice of E:

```
func Map[E any](s []E, f mapFunc[E]) []E {
    result := make([]E, len(s))
    for i := range s {
        result[i] = f(s[i])
    }
    return result
}
```

For any type E, `Map[E]` takes a slice of E and a `mapFunc[E]`, and returns a slice of E.

As usual, the implementation is not the interesting part. What's interesting is that now we can easily map any suitable function over a slice.

For example, given a slice of `string`, we could map some function that takes a string and returns a string. Here's one example from the standard library:

```
s := []string{"a", "b", "c"}
fmt.Println(Map(s, strings.ToUpper))
// [A B C]
```

## Type inference

It might not be immediately obvious what stops us trying to apply an *unsuitable* function here.

For example, suppose we tried to map `strings.ToUpper` over a slice of `int`. Clearly that's insane, but let's try:

```
s := []int{1, 2, 3}
fmt.Println(Map(s, strings.ToUpper))
// type func(s string) string of strings.ToUpper does not
// match inferred type mapFunc[int] for mapFunc[E]
```

We knew this wouldn't work, but the error message is worth parsing carefully, because it shows how the compiler does *type inference*: working out what E is in this instance, without us having to explicitly specify it.

Recall that `Map` declares a parameter of type `[]E`, and in this case that's `[]int`. Fine. The compiler can correctly infer that E is `int` here.

But now there's a problem: the second parameter is a `mapFunc[E]`, for whatever E is. And since we know E is `int`, then the type of the function argument must be `func(int) int`.

The signature of the one we actually *passed*, though, is `func(string) string`. That's a straightforward type mismatch, hence the error.

## Exercise 6.1: Func to funky

For this exercise, you'll be building a simple *dynamic dispatch* system in Go. What's required here is a way of storing a number of different functions by name, and then applying the one you want.

Your job is to create a `FuncMap` type which can store a number of named functions. That is, it's a map of `string` to some arbitrary function type.

For example:

```go
fm := FuncMap[int, int]{
    "double": func(i int) int {
        return i * 2
    },
    "addOne": func(i int) int {
        return i + 1
    },
}
```

You'll also need to implement an `Apply` method on `FuncMap` which will apply the specified function to some value and return the result. For example:

```go
fmt.Println(fm.Apply("double", 2))
// 4
```

If it stops being fun, take a look at one possible answer in Solution 6.1.

## Filter

Let's look at another popular functional pattern on slices. `Filter` is the conventional name for a function that, like `Map`, takes an arbitrary function and applies it to each element of a slice.

However, where Map uses the supplied function to *tranform* each element, `Filter` uses it to decide which elements to keep and which to discard.

The result of `Filter` is a slice containing only the elements that satisfied the `keep` function.

For example, suppose we have a slice of integers and we want to extract only the *even* values from it. We need some generic function `Filter` that takes an arbitrary slice, and an arbitrary function that decides which elements to keep and which to discard.

What can we deduce about the signature of this `keep` function, then?

```
type keepFunc[E any] func(E) bool
```

It needs to take take E and return `bool`, doesn't it? We'll say that if it returns true, then we'll keep the element.

Fine. So now we can write `Filter`:

```
func Filter[E any](s []E, f keepFunc[E]) []E {
    result := []E{}
    for _, v := range s {
        if f(v) {
            result = append(result, v)
        }
    }
    return result
}
```

For any type E, `Filter[E]` takes a slice of E and a `keepFunc[E]`, and returns a slice of E.

Again, there's nothing very complicated about the implementation here. We range over the input slice checking whether `f` is true for each element. If so, we append it to the result slice. At the end, we have the slice containing only the elements of `s` for which `f` is true.

What kind of `f` would be useful? Suppose we want to filter a slice of integers to find only the even values.

We could write some function matching the `keepFunc` signature that returns true if its argument is even; that is, if dividing it by 2 gives a remainder of zero. Here it is:

```
func(v int) bool {
    return v%2 == 0
}
```

Let's try it out, by passing this function literal to `Filter`:

```
s := []int{1, 2, 3, 4}
fmt.Println(Filter(s, func(v int) bool {
    return v%2 == 0
}))
// [2 4]
```

Reassuring! This looks more or less the same as when we used `Map`, except that instead of passing an existing function `strings.ToUpper`, we supplied an anonymous *function literal* instead, which is quite idiomatic in Go.

## Passing a generic function

We're not limited to passing only function literals to `Filter`, of course. We could certainly pass some existing *named* function, as we did with `strings.ToUpper`, too.

But there's another interesting option. What if we wanted to pass a *generic* `keepFunc`? That is, a function parameterized on some T?

Could we do that? We might start by trying something like this:

```
func IsEven[T any](v T) bool {
    return v%2 == 0
}
```

Stop, stop! We know this won't work. Why? Because we can't use the == operator with `any`. We can only use it with `comparable` types.

And the % (remainder) operator only works with integers, which is an even smaller type set. So that's the constraint we'll need to use:

```
func IsEven[T constraints.Integer](v T) bool {
    return v%2 == 0
}
```

Let's try it out:

```
fmt.Println(Filter(s, IsEven[int]))
// [2 4]
```

Wonderful!

## Reduce

`Map` and `Filter` make perfect partners, but they're even better when united with their best friend, `Reduce`. What does that do?

If Map uses a function to transform each element of a slice, and Filter uses a function to pick only the desired elements, then Reduce uses a function to *combine* the elements of a slice.

This operation is sometimes also called Fold, Inject, or a variety of other names depending on the programming language. Essentially, it's about turning a sequence of values into a *single* value by some kind of computation.

For example, a simple way to reduce a slice of numbers would be to *sum* them. We could do this by calling Reduce with some function that adds each element to the current running total.

In general, then, this arbitrary reduction function needs to take two things: a value representing the "current result" of the reduction (so far), and the next slice element to combine with it.

Let's try to work out the signature of such a function first:

```
type reduceFunc[E any] func(cur, next E) E
```

For any type E, a reduceFunc[E] is a function that takes two E parameters and returns E.

## Implementing Reduce

This makes sense, so now let's try to write Reduce:

```
func Reduce[E any](s []E, init E, f reduceFunc[E]) E {
    cur := init
    for _, v := range s {
        cur = f(cur, v)
    }
    return cur
}
```

For any type E, Reduce[E] takes a slice of E, a value of E, and a reduceFunc[E], and returns an E result.

The first parameter is the slice to operate on. The second is some starting value for the reduction (for example, zero). And the third is the reduceFunc to apply.

The implementation is pretty straightforward. We set cur equal to the initialising value init. Then, for each slice element, we call the reduceFunc with the current value of cur, and the element. The result of the reduceFunc becomes the next cur.

Let's try summing a slice of integers, for example. Our reduceFunc is easy to write. It needs to take a cur and next value and return their sum:

```
func(cur, next int) int {
    return cur + next
}
```

So let's pass this function literal to Reduce and see what it does:

```
s := []int{1, 2, 3, 4}
sum := Reduce(s, 0, func(cur, next int) int {
    return cur + next
})
fmt.Println(sum)
// 10
```

## Other reduction operations

Let's try another reduction operation. What about multiplication, for example?

```
s := []int{1, 2, 3, 4}
p := Reduce(s, 1, func(cur, next int) int {
    return cur * next
})
fmt.Println(p)
// 24
```

We're not restricted to just numerical operations, of course. Here's a reduction involving strings:

```
s := []string{"a", "b", "c"}
j := Reduce(s, "", func(c, n string) string {
    return c + n
})
fmt.Println(j)
// abc
```

The effect of this `reduceFunc` is to *join* all the elements of `s` together into a single string.

## Constraints

You might be wondering why it's okay to use the `any` constraint here. Doesn't that mean we can't use any operators in our `reduceFunc[E]`?

For example, when summing integers, shouldn't we have to constrain the `Reduce` function to only the set of types that support the + operator?

Well, let's think about what would happen if we tried to `Reduce` a slice of things that *don't* support that operator: a struct, for example.

What kind of `reduceFunc` would we need to pass in that case? Something like this, perhaps:

```go
func sumStructs(cur, next struct{}) struct{} {
    return cur + next
}
// invalid operation: operator + not defined on cur (variable
// of type struct{})
```

Well, we can't write that function! Structs don't support the + operator. So the problem doesn't arise.

## Concurrency

If we had large slices to deal with, and if the map, reduce, or filter operations on them were relatively expensive, it might be helpful to do these operations *concurrently*.

For example, suppose we had a slice of strings representing web URLs, and we wanted to filter them to extract only those URLs which respond with OK status.

We can imagine using our existing `Filter` implementation on this slice with some function that uses `http.Get` to call the URL and check the response status. But each request would take a fair amount of time (by computer standards): a few hundred milliseconds, perhaps.

Suppose each request takes about 250ms to complete, and there are 1,000 URLs to check. Filtering them sequentially would thus take more than two minutes.

Since we can make many HTTP requests at once, it seems silly not to start *all* these requests concurrently, and then wait for the responses to trickle in. The total filter time would then be a little more than the time of the slowest response, so of the order of 250ms.

That's a pretty big speedup, and since all the tasks in this workload are independent of each other (making it what's known as *embarrassingly parallel*), the speedup is linear with N. In other words, concurrent filtering of slice elements always takes about the same total time, no matter how large the slice.

Naturally, adding concurrency to something like `Filter` also adds a good deal of complication and extra code, but that's okay: if it can be generic, then we only need to write it once! Indeed, some third-party libraries already provide a concurrent `Filter` and related functions.

## Exercise 6.2: Compose yourself

*Composing* functions means applying them in a chain, so that each function operates on the result of the previous one.

For example, if we had two arbitrary functions `outer` and `inner`, then we could compose them directly by writing:

```
outer(inner())
```

Your job in this exercise is to write a `Compose` function that will do just this. Given two functions, `outer` and `inner`, and a value of arbitrary type, it should first apply `inner` to the value, then apply `outer` to the result returned by `inner`, then return the result returned by `outer`.

Got all that? Here's an example. Suppose we had a function `double` that returns double its input, and another function `addOne` that returns the result of adding 1 to its input.

And suppose our input value is 1. If we first apply `addOne` to 1, we should get 2. If we then apply `double` to 2, we should get 4. So:

```
fmt.Println(Compose(double, addOne, 1))
// 4
```

What if we composed the functions in reverse order, so that we applied `double` first, then `addOne` to the result? Well:

```
fmt.Println(Compose(addOne, double, 1))
// 3
```

If you're having trouble with this, remember that the `inner` function will need to take a parameter of the same type as the value, and its result type is arbitrary. The `outer` function will need to take a parameter of whatever type `inner` returns, but *its* result type is also arbitrary.

So, all told, there are *three* type parameters involved here.

If you're still stuck, have a look at one possible way of doing it in Solution 6.2.

## Takeaways

- One of the most important applications of generics in Go is writing general-purpose functions on container types, such as maps and slices.
- For example, we can now write a function that takes a slice of any (comparable) type, and tells us whether it contains a given element.
- We can also reverse a slice, which is something that's extremely awkward to do without generics.

- The existing `sort` API for slices is a little weird, and with generics we can write a much more pleasing `Sort` function on any ordered type.
- Apart from these common and useful operations on generic slices, we can also now apply *arbitrary* operations to slices, thanks to *first-class functions*.
- For example, we can *map* a given function over a slice; that is, we can apply an arbitrary function to each element of the slice, yielding the transformed slice.
- We can also write a `Filter` function that picks only certain elements from a slice, again using an arbitrary function supplied by the user, perhaps even a standard library function.
- Usefully, we can also pass a generic function to `Filter`, such as `isEven[T]`, for some type T.
- Similarly, we can write a `Reduce` function that applies some arbitrary operation to slice elements to *combine* them into a single result: for example, finding the sum or product of a slice of numbers.
- Another exciting application of generics is writing *concurrent* versions of things like `Map`, `Reduce`, and `Filter`, which can greatly improve performance on suitable workloads.

## Review questions

1. What constraint would be appropriate on the type parameter to a function such as `Max` or `Min`? What about `Sort`? What about `Reverse`? What about `Contains`? Justify your answers.
2. Write the signature of a generic function `IsPrime` that would be suitable for passing to a generic `Filter` function on some slice type.
3. Write an anonymous function literal that returns the product of its two inputs, suitable for passing as the function argument of `Reduce` on a slice of numbers. Use it to implement a generic `Factorial` function on such a slice.
4. If we constrain the type parameter of `Map`, `Reduce`, or `Filter` to `any`, which seems like a good idea, what would happen if we wanted to apply some function that requires more tightly-constrained parameters? For example, filtering a slice by `IsEven`, which works only with integers. Why doesn't this cause a run-time error?
5. Briefly explain what's meant by "first-class functions", and give some examples of where they might be useful.

# 7. Containers

*In order to be irreplaceable, one must always be different.*
—Coco Chanel

In the previous chapter we looked at some interesting generic functions that we can write on slices (and, by extension, maps). Maps and slices are so useful and ubiquitous in Go precisely because they can be used to hold elements of any type.

Before the introduction of generics, though, we were more or less limited to just maps and slices, because it wasn't possible to write user-defined containers of arbitrary types.

Now that we can, what container types, or other generic data structures, would it be interesting to write?

## Sets

A *set* is like a kind of compromise between a slice and a map. Like a slice, a set is a collection of elements, all of the same type. Like a map, there's no defined order, and elements must be unique.

We very often need something like a set in Go, but there's no built-in or standard library set type, so we tend to roll our own using a map. For example:

```
var validCategory = map[string]bool{
    "Autobiography":       true,
    "Large Print Romance": true,
    "Particle Physics":    true,
}
```

—For the Love of Go

The *value* type of the map doesn't matter here, since we're only really interested in the keys. But it's convenient to use `bool` values, since it means we can use them in conditional expressions:

```
if validCategory[category] {
```

As you probably know, a map index expression like this returns the zero value if the given key is not in the map. Since the zero value for `bool` is `false`, this means the `if` condition is false if `category` is not in the "set".

As neat as this is, it feels a little hacky, so it's nice that we can now write a proper `Set` type using generics. Let's try.

## Operations on sets

What kind of things would we like to be able to do with a set? There are many possibilities, but perhaps the most basic are adding an element, and checking whether the set contains a given element.

In other words, we'd like to be able to write something like this:

```
s := NewSet[int]()
s.Add(1)
fmt.Println(s.Contains(1))
// true
```

Let's get to work!

## Designing the Set type

We need something to keep the actual data in. A map would make sense, since it gives us the "unique key" property for free.

And we know the map keys will need to be of arbitrary (but comparable) type, since they represent the elements of our Set. What about the value type?

We could use `bool` again, but maybe `struct{}` is a better choice in this case, since it takes up no space.

So suppose we wrote some type definition like this:

```go
type Set[E comparable] map[E]struct{}
```

For some comparable type E, a Set is a map of E to empty struct.

And since we can't use a map until it's initialised, let's provide a constructor function NewSet:

```go
func NewSet[E comparable]() Set[E] {
    return Set[E]{}
}
```

## The Add method

Let's write an Add method on this type. It should take some value and store it in the set. We can do that by assigning to the map:

```go
func (s Set[E]) Add(v E) {
    s[v] = struct{}{}
}
```

## The Contains method

The next job is to write Contains. Given some value, it should return true if the value is in the set, or false otherwise.

You probably know that a map index expression in Go can return two values. The second, conventionally named ok, tells us whether the key was in fact in the map. And that's the information we want here:

```go
func (s Set[E]) Contains(v E) bool {
    _, ok := s[v]
    return ok
}
```

Let's try out some simple set operations:

```go
s := NewSet[string]()
s.Add("hello")
fmt.Println(s.Contains("hello"))
// true
```

Great! This is already useful. But let's see what we can do to make it even better.

## Initialising with multiple values

It seems as though it would be convenient to initialise a set with a bunch of elements supplied to NewSet, rather than having to create the set first and *then* add each element one by one. In other words, we'd like to write:

```
s := NewSet(1, 2, 3)
```

Not only does this save time, but now we don't need to instantiate NewSet, because the compiler can infer the type of E from the values we pass to it.

Let's modify NewSet to take any number of E values:

```
func NewSet[E comparable](vals ...E) Set[E] {
    s := Set[E]{}
    for _, v := range vals {
        s[v] = struct{}{}
    }
    return s
}
```

While we're at it, we'll make the same change to Add, so that we can add any number of new members in one go:

```
func (s *Set[E]) Add(vals ...E) {
    for _, v := range vals {
        s[v] = struct{}{}
    }
}
```

We're ready to roll:

```
s := NewSet(true, true, true)
s.Add(true, true)
fmt.Println(s.Contains(false))
// false
```

What else could we do?

## Getting a set's members

It'll almost certainly be convenient to get all the members of the set as a slice. One of the more annoying limitations of using maps as sets in Go is that it's not straightforward to do this.

Let's make it straightforward!

```
func (s Set[E]) Members() []E {
    result := make([]E, 0, len(s))
    for v := range s {
        result = append(result, v)
    }
    return result
}
```

Now we can write:

```
s := NewSet(1, 2, 3)
fmt.Println(s.Members())
// [1 2 3]
```

## A `String` method

In fact, let's add a `String` method so that we can print the set in a nice way:

```
func (s Set[E]) String() string {
    return fmt.Sprintf("%v", s.Members())
}
```

Now we don't need to call `Members` to print out the set, and we can write simply:

```
fmt.Println(s)
// [2 3 1]
```

Notice that the members came out in a different order this time, and that's okay!

It's part of the definition of a set that its members *aren't* in any special order, just like a map. And because we're using a map to store the elements, we get that behaviour for free.

## The union of two sets

Let's do something more fun. A common operation on sets is to ask for the *union* of two sets: that is, the set that contains all the elements of both sets. How could we implement that?

One way would be to create a new set, using the members of set 1, then add the members of set 2. Here's what that looks like:

```
func (s Set[E]) Union(s2 Set[E]) Set[E] {
    result := NewSet(s.Members()...)
    result.Add(s2.Members()...)
    return result
}
```

Making `NewSet` and `Add` variadic has paid off already, since it makes `Union` much easier to write, with no looping.

If this works, then we should be able to find the union of two small sets of integers:

```
s1 := NewSet(1, 2, 3)
s2 := NewSet(3, 4, 5)
fmt.Println(s1.Union(s2))
// [1 2 3 4 5]
```

Excellent! As we'd expect, `Union` has not just added together the two sets, it's also removed any duplicates. We didn't have to write any code to do this: again, it came free with the map.

## An `Intersection` method

Let's write `Intersection`, too, just for completeness. The intersection of two sets is the set of members that they have in common.

This is a little more difficult to write, but I think we can do it:

```
func (s Set[E]) Intersection(s2 Set[E]) Set[E] {
    result := NewSet[E]()
    for _, v := range s.Members() {
        if s2.Contains(v) {
            result.Add(v)
        }
    }
    return result
}
```

In other words, for each member of `s`, we check whether it is also present in `s2`. If so, we add it to the result set.

Does it work? Here goes:

```
s1 := NewSet(1, 2, 3)
s2 := NewSet(3, 4, 5)
fmt.Println(s1.Intersection(s2))
// [3]
```

Encouraging!

## Logic on sets

Let's try out our new Set type in a semi-realistic application: checking job applications against the required set of skills for a vacancy.

Since we've just written `Intersection`, this should be easy. We just need to define the set of required skills, and find its intersection with the *candidate's* skills.

If there are any members in this set, then the candidate has at least one of the required skills, so we can hire them.

For example, suppose there's a job available which requires either Go or Java skills, but I happen to know Go and Ruby. Would I qualify?

Let's find out:

```go
jobSkills := NewSet("go", "java")
mySkills := NewSet("go", "ruby")
matches := jobSkills.Intersection(mySkills)
if len(matches) > 0 {
    fmt.Println("You're hired!")
}
// You're hired!
```

If only it were that easy.

## Exercise 7.1: Stack overflow

A stack is an abstract data type which stores items in a last-in-first-out way. The only thing you can do with a stack is "push" a new item onto it, or "pop" the top item off it.

Your task here is to define a generic `Stack` type that holds an ordered collection of values of as broad a range of types as possible.

You'll need to write a Push method that can append any number of items to the stack, in order, and a Pop method that retrieves (and removes) the last item from the stack.

Pop should also return a second ok value indicating whether an item was actually popped. If the stack is empty, then Pop should return `false` for this second value, but `true` otherwise.

Also, you should provide a Len method that returns the number of items in the stack.

For example:

```go
s := Stack[int]{}
fmt.Println(s.Pop())
// 0 false
```

```
s.Push(5, 6)
fmt.Println(s.Len())
// 2
v, ok := s.Pop()
fmt.Println(v)
// 6
fmt.Println(ok)
// true
```

If you'd like some hints, check out Solution 7.1 for one possible way to solve this.

## And more

When we had to write, or generate, lots of duplicated code to implement data structures on multiple specific types, there was an obvious disincentive to put too much effort into it. Since it was impractical to provide general-purpose libraries of such data structures, we tended to hack together our own versions as needed.

That being so, they generally weren't very sophisticated: no one wants to maintain N different versions of the same sophisticated data structure. So we often went without neat features like concurrency safety, high performance, state-of-the-art algorithms, and so on.

That's no longer the case, and we can write generic abstract data types that are as sophisticated as we like. Let's explore this a little in the next chapter, by trying to add a relatively simple feature to our new `Set` type: concurrency safety.

## Takeaways

- It was awkward to write general-purpose container types before generics, so we usually didn't bother; now we can.
- Most generic containers will probably use maps or slices for their underlying storage, since they're the fundamental building blocks of other kinds of type.
- But they can add useful new features to them, such as concurrency safety (or even concurrent operations).

## Review questions

1. Why was it so difficult (or impossible) to write general-purpose container libraries in Go before generics? What is awkward or sub-optimal about the pre-generic standard library types such as `sync.Map` and `container.List`? How does generics help with these problems?

2. Why must most container types be constrained by `comparable`? Can you think of any useful generic containers that could be constrained by `any` instead?

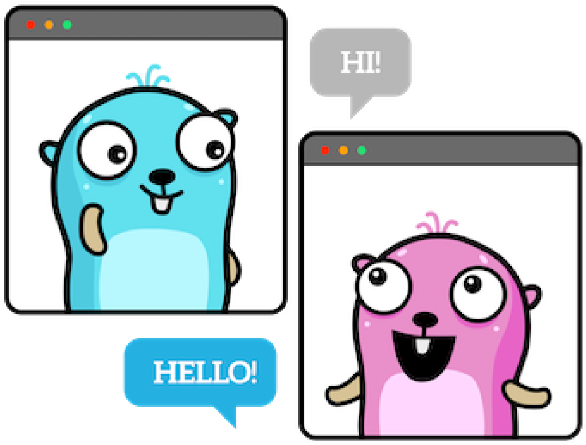
3. What can you deduce about a method on some container type that takes a pointer receiver? By implication, what can you deduce about a method that *doesn't* take a pointer? Say whether (and why) a pointer receiver would be appropriate for methods named (A) `Contains`, (B) `Len`, (C) `Append`, and (D) `Delete`.

4. Briefly explain the "union" and "intersection" set operations. What is the union of a set with itself? What is the union of a set with the empty set? What is the intersection of a set with itself? What is the intersection of a set with the empty set?

5. Why might a Go program that operates on sets produce different results each time we run it? What issues might this cause? How could we solve them?

# 8. Concurrency

*Any sufficiently complicated concurrent program in another language contains an ad hoc informally-specified bug-ridden slow implementation of half of Erlang.*
—Robert Virding



Something that the built-in collection types in Go don't have is *concurrency safety*. That is to say, if one goroutine writes to a map, for example, and another goroutine reads it concurrently, then a *data race* exists, and something bad will happen.

At best, the program will terminate unexpectedly. At worst, it will continue to run, but with corrupted or incorrect data. It's up to you to avoid this situation.

The best way to prevent data races on shared data is *not to have any mutable shared data*, but this isn't always possible.

The second best way is *not to share the data*. That is, to allow only one *guard* goroutine to access it, and require other goroutines to issue read or write instructions to it via a *channel*, which is inherently concurrency-safe.

## Mutexes

If neither of these options are available, and they certainly wouldn't be in a general-purpose set library, for example, then another possibility is to use a *mutex*.

A mutex (as in “mutual exclusion”) is a concurrency-safe “lock” variable. Goroutines can use it to obtain “permission” before reading or writing the shared data, and release it afterwards.

Indeed, the standard library includes a `sync.Map` type, which is a concurrency-safe map with an internal mutex. Because it pre-dates generics, though, its API is restricted to `interface{}`, which is annoying, so it’s not much used.

There are several drawbacks to using mutexes in general. First, if goroutines often need concurrent access to the data, there will be *contention* for the mutex. In other words, goroutines will waste time waiting for the lock when they could have been doing useful work.

Second, when there is more than one mutex, a *deadlock* can occur, bringing the program to a halt. For example, goroutine A holds mutex 1, and is waiting for mutex 2, while in turn goroutine B holds mutex 2, and tries to get mutex 1.

Like two drivers at a road junction, each waiting for the other to go first, deadlocked goroutines will block each other forever. At best, a deadlock will cause your program to crash. At worst, it will *look* like it’s okay, but stop doing any useful work, which can be hard to detect.

Third, since mutexes are “opt-in”, people can use them wrongly. For example, they might forget to release the lock, which would be inefficient. Worse, they might forget to *obtain* it in the first place, which would be disastrous.

As a library writer, you can make a mutex available, and add all the necessary locking logic to your code, but you can’t actually ensure that people remember to use it.

Unless, of course, you force them to call your *accessor methods* to read or write the object, and that’s the usual solution.

Let’s see if we can build on the Set type we developed in the previous chapter to add concurrency safety using accessor methods in this way.

## A concurrency-safe set type

We’ll define a new type `SetC` which incorporates the same Set functionality we already have, but we’ll add a mutex to it to make it safe for concurrent access.

We’ll use the standard library `sync.RWMutex` type, which allows us to get either a *read lock* or a *write lock* on the data, depending what we need to do with it.

We need to keep both the mutex and the data map in the same value. That sounds like some kind of struct, doesn’t it?

Let’s start by writing something like this:

```
type SetC[E comparable] struct {
    mutex *sync.RWMutex
    data  map[E]struct{}
}
```

For some comparable type E, a SetC[E] is a struct containing a *sync.RWMutex field, and a map[E]struct{} field.

We need to store a *pointer* to the mutex, not the mutex value itself. Otherwise, if the struct is copied, we'll get two copies of the mutex, which would mean it no longer protects the data.

It never makes sense to copy a mutex, so when we store them in a struct, we use a pointer field to prevent this.

## The constructor

What would NewSetC need to do? Well, initialise the map and store any supplied values, just as before. It also needs to initialise the mutex:

```
func NewSetC[E comparable](vals ...E) *SetC[E] {
    s := SetC[E]{
        mutex: &sync.RWMutex{},
        data: map[E]struct{}{},
    }
    for _, v := range vals {
        s.data[v] = struct{}{}
    }
    return &s
}
```

## A locking Add method

The behaviour of Add on our SetC type will be pretty much the same as before, with one important difference. It needs to lock the mutex before updating the map.

Specifically, it needs a write lock, to prevent any other goroutine from even *reading* the map while the update is happening.

Here's what that looks like:

```
func (s *SetC[E]) Add(vals ...E) {
    s.mutex.Lock()
    defer s.mutex.Unlock()
    for _, v := range vals {
        s.data[v] = struct{}{}
    }
}
```

## The read-locking methods

What about `Members`? That also needs to be lock-aware, but does it need a write lock? No, because it doesn't modify the data: it just reads it.

So a read lock is good enough here:

```
func (s SetC[E]) Members() []E {
    s.mutex.RLock()
    defer s.mutex.RUnlock()
    result := make([]E, 0, len(s.data))
    for v := range s.data {
        result = append(result, v)
    }
    return result
}
```

`Contains` is also read-only, so we can use the same strategy:

```
func (s SetC[E]) Contains(v E) bool {
    s.mutex.RLock()
    defer s.mutex.RUnlock()
    _, ok := s.data[v]
    return ok
}
```

If we've done everything right so far, then we should be able to use a SetC in the ordinary, non-concurrent way that we did before. Let's try:

```
s := NewSetC('a', 'b')
s.Add('c')
fmt.Println(s.Contains('d'))
// false
```

Great, but now we need to find out if the concurrency safety works. How could we do that?

## Smoke testing

We need to *use* our new type concurrently. That is to say, we need at least two goroutines.

And to expose a data race if it exists, at least one of those goroutines will need to *write* to the SetC, while the other reads it.

Just a single read and write might not be enough to trigger a crash; we might get lucky. So to increase the odds against us, let's read and write the SetC many times. Say, a thousand times.

If our locking works correctly, there should be no problem:

```
s := NewSetC(1, 2, 3)
var wg sync.WaitGroup
wg.Add(1)
go func() {
    for i := 0; i < 1000; i++ {
        s.Add(i)
    }
    wg.Done()
}()
for i := 0; i < 1000; i++ {
    _ = s.Members()
}
wg.Wait()
fmt.Println("We made it!")
// We made it!
```

This looks promising.

## A fatal error

Great! But *was* our mutex really necessary, though? We can't tell from this result.

Maybe we would have been just fine without it. We wouldn't, but I don't expect you to take my word for that.

Let's try commenting out the locking code in Add. If we're right, this should cause a crash:

```
func (s *SetC[E]) Add(vals ...E) {
    // s.mutex.Lock()
```

```
        // defer s.mutex.Unlock()
        ...
```

Here we go:

```
fatal error: concurrent map iteration and map write

goroutine 1 [running]:
... [stack frames omitted]
main.(*SetC[...]).Members(0xc0000b6000?)
        main.go:44 +0x145 fp=0xc0000c3e40
        sp=0xc0000c3d30 pc=0x108ba25
... [stack frames omitted]

goroutine 18 [runnable]:
main.(*SetC[...]).Add(...)
        main.go:36
... [stack frames omitted]
```

Yikes. We can see from this stack trace that the Go runtime detected a *read* on the map in Members and a concurrent *write* in Add.

Since it can no longer guarantee the integrity of the data, it terminates the program immediately.

And that's exactly what we expected to happen when we disabled the mutex. Turns out the mutex was doing something useful after all!

## The race detector

In fact, Go's built-in data race detector would have caught this problem for us, if we'd only thought to invoke it by adding the -race flag to our go run command. Let's see what it says now:

```
go run -race main.go
```

```
==================
WARNING: DATA RACE
Read at 0x00c000124180 by main goroutine:
  main.(*SetC[...]).Members()
      main.go:43 +0xe4
  main.main()
      main.go:86 +0x5b2

Previous write at 0x00c000124180 by goroutine 7:
  runtime.mapassign_fast64()
      /usr/local/go/src/runtime/map_fast64.go:93 +0x0
  main.(*SetC[...]).Add()
```

```
    main.go:36 +0xc8
main.main.func1()
    main.go:81 +0x4d
```

This underlines the importance of using the `-race` flag when testing any code involving concurrency, doesn't it? It's not infallible, but it goes a long way towards automatically catching data race situations like this.

If there *is* a data race in your program, it may not blow up today, and maybe not tomorrow. But it *will* blow up eventually, and most likely at a time of peak load, which is bound to make you somewhat unpopular.

## And the rest

How should we update the other `Set` methods to work with `SetC`, such as `String`, `Union`, and `Intersection`?

We can copy the existing `String` method as is, because it only calls `Members`, and that's already lock-aware.

The same goes for `Union`, which only calls `Members` and `Add`.

We have a choice, though, when it comes to implementing `Intersection`. The old code will also be concurrency-safe without modification, because it also uses the lock-aware methods.

But let's take a closer look at the code for Set:

```go
func (s Set[E]) Intersection(s2 Set[E]) Set[E] {
    result := NewSet[E]()
    for _, v := range s.Members() {
        if s2.Contains(v) {
            result.Add(v)
        }
    }
    return result
}
```

There might be an efficiency issue here. Can you see what it is?

Since we call `s2.Contains` (which locks) for *every* member of `s`, there's potentially a lot of locking and unlocking going on.

Mutexes are fast, but they aren't instantaneous, so too much mutex *churn* hurts performance. To make `Intersection` efficient on large sets, it might be a good idea to eliminate the repeated calls to `Contains`.

Instead, we could get the read lock just once, execute the whole loop, and then release the lock.

Here's the result:

```
func (s SetC[E]) Intersection(s2 SetC[E]) *SetC[E] {
    result := NewSetC[E]()
    s2.mutex.RLock()
    defer s2.mutex.RUnlock()
    for _, v := range s.Members() {
        _, ok := s2.data[v]
        if ok {
            result.Add(v)
        }
    }
    return result
}
```

It's a trade-off, though, isn't it? Locking the whole set while we read all the members avoids thrashing the mutex, but for large sets that might mean holding the lock for a long time.

So for some applications we might prefer more *granular* locking: that is, locking and unlocking the whole set for each individual read operation. It just depends.

## Exercise 8.1: Channelling frustration

Can you put together everything you've learned in this chapter to solve Exercise 8.1?

It seems people have been using channels in Go a little too freely, and a scheme is now proposed to bill users by activity.

Accordingly, you'll need to define a generic `Channel` type with suitable instrumentation to track the number of sends and receives.

You have the following tasks to complete:

1. Define the `Channel` type.
2. Define a `NewChannel` function that returns an initialised `Channel` with a specified capacity, ready for use.
3. Define `Send` and `Receive` methods on the `Channel` to allow users to send and receive values of the appropriate type.
4. Define `Sends` and `Receives` methods that will report the number of sends and receives on the channel, for billing purposes.

For example:

```
c := NewChannel[string](1)
c.Send("hello")
fmt.Println(c.Sends())
// 1
fmt.Println(c.Receive())
// hello
fmt.Println(c.Receives())
// 1
```

You'll need to ensure that your `Channel` is concurrency-safe, so make sure your solution passes the tests when the race detector is enabled:

```
go test -race
```

You can use an ordinary Go channel internally, of course: no need to re-implement that behaviour. But you'll need to wrap the channel with some methods that track its send and receive activity, in a concurrency-safe way.

See Solution 8.1 if you'd like something to compare with your answer.

## Extending the Set type

Alas, space prevents us exploring all the possible enhancements we could make to the `Set` and `SetC` types, but here are a few ideas you might like to think about.

For one thing, it's not really very convenient to have two separate set types, each with exactly the same API but with different semantics. Ideally, we'd just have one `Set` type, but we could opt in to concurrency safety if we needed to. The set could use a flag internally to tell it whether or not it should use the lock.

How would that work? Perhaps we could have two constructors: `NewSet` and `NewSafeSet`, for example. Or, since it's always best to err on the side of safety, maybe `NewSet` and `NewUnsafeSet`. This way, users will get a safe set by default, and if they need an *unsafe* one for better performance, they can request it explicitly.

## More set operations

There are lots more fun and useful things we could implement for our Set types. You might like to have a go at some of these yourself:

- A `Len` method
- A `Delete` method
- An `Equals` method
- An `IsSubset` method
- A `Subtract` method that removes all members that belong to some other set
- A `Difference` function to find the members that two sets don't have in common

- A `Make` function to pre-allocate memory
- `Map`, `Reduce`, and `Filter` operations on set elements

Also, the set types we've developed are *mutable*. That is, when you call `Add` on a set, for example, it updates the set in place.

For a more functional style of programming, we might like to have an *immutable* set type. In this case, methods like `Add` would *return* the modified set instead of changing the original.

## Key-value stores and caches

Here's another idea. Right now, we can only store values in a set, but it would be nice to store *key-value pairs* instead. This would be a great way to *cache* data that's expensive to generate.

We already have almost all the machinery we need. Instead of always storing `struct{}` in the map, we could store some arbitrary value instead, couldn't we?

And since we tend to want to cache things in high-concurrency applications like network servers, we'll probably also want concurrency safety. So we could build a key-value store on top of a `SetC`.

Cached data usually has a certain *lifetime*, or period of validity, after which it *expires* and needs to be refreshed. So perhaps we could extend the key-value store to handle deleting, or refreshing, expired data.

## Other container types

Sets are just the beginning. There are many other useful data structures we can now write generically in Go: trees, heaps, queues, rings, linked lists, graphs, vectors, matrices, and so on.

Very likely we'll see lots of new libraries providing high-quality implementations of these things for use in Go programs. That's great, and even if we don't often need to write generic code ourselves, we can still benefit from the general-purpose libraries provided by others.

And if you feel like writing such a library yourself, go right ahead!

## Takeaways

- While we always want to avoid mutable shared data if possible, it isn't always possible, so concurrency-safe container types are a good fall-back option.
- We can make a generic type concurrency-safe by wrapping a mutex inside it, and ensuring that all our methods use an appropriate lock.

- The struct must contain a *pointer* to the mutex, since copying a mutex makes no sense.
- The race detector, though not infallible, is a vital tool for testing concurrent Go programs.
- No matter how carefully we write programs, it's still alarmingly easy to unwittingly introduce data races, deadlocks, and other concurrency bugs.
- So it's important to *smoke-test* your concurrent code under as much contention as possible, in the hope of flushing out all such bugs.

## Review questions

1. What are some ways of making programs concurrency-safe without using mutexes? What are some drawbacks of using mutexes? When *can't* we avoid using a mutex?
2. What are the conditions necessary for a data race to exist? What are the possible consequences of a data race? How can the Go race detector help, and how do we invoke it when running tests or programs?
3. What are some design considerations for locking behaviour on collection types? When does it make sense to hold the mutex for as short a time as possible? When does it make sense to minimise the number of lock-unlock cycles?
4. How could a Set type like the one in this chapter be extended to store arbitrary key-value data? What would be the type constraints on such data?
5. What additional feature might it be useful to have if the key-value store were being used to cache data that's expensive to generate, but also time-sensitive?

# 9. Libraries

*If you have a garden and a library, you have everything you need.*
—Cicero

In the previous chapters, we've seen that the introduction of generics means that we can write library packages in a new way.

Even more significantly, we can write new *kinds* of library: general-purpose data structures, and general-purpose functions on arbitrary collections of data.

Great, but what if we're not really interested in writing libraries? What if we just want to sit back and enjoy using other people's libraries? How will generics affect us in that case?

In this chapter, we'll discuss the new packages that accompany generics, and the changes that generics brings to the Go ecosystem in general.

## The `constraints` package

Generics brings a new official Go package with it: `constraints`. We've already talked about it a bit in previous chapters, but here's a brief review.

`constraints` defines five constraints for numeric types, `Signed`, `Unsigned`, `Integer`, `Float`, and `Complex`. These are guaranteed to include all the appropriate built-in types, now and in the future.

Additionally, `constraints.Ordered` includes all built-in types that can be compared for relative magnitude using the > operator and friends. These are the real number type sets `Integer` and `Float`, but also any types derived from `string`, because strings can be ordered alphabetically.

## Slices and maps

And the changes don't stop there. Since, as we've seen, generics lets us write useful general-purpose functions on slices and maps, such as `Contains`, it makes perfect sense to provide some of these as part of the Go standard library.

Accordingly, two new packages will be introduced: `slices` and `maps`. We'll look at these in more detail in a moment, but first a word about how they'll get to us.

Initially, the idea was to introduce the new packages directly into the standard library in Go 1.18, the first release to officially support generics.

However, a wise person proposed holding off the introduction of these packages for at least one release:

> *It's too much to do all at once, and we might get it wrong. One thing Go has taught us is that it grows its own ways of doing things.*
>
> *For generics, we don't know what those new ways are yet. Also, the compatibility promise makes the cost of getting any detail wrong quite high. We should wait, watch, and learn.*
> —Rob Pike

## The `x/exp` repo

As we discussed in the chapter on constraints, there's an official "experimental" Go package repository called golang.org/x/exp. Packages here are excused from the Go 1 compatibility promise, so they can change in ways that might break existing code.

The `x/exp` repo is a kind of proving ground for packages which may (or may not) make it into the standard library eventually. It will let us *use* the new library packages for a while, try out their APIs, and see what patterns emerge.

Because of the compatibility promise, the APIs of *standard* library packages can never be changed, though they can be *extended*, and often are. By contrast, packages in `x/exp` can (and likely will) change, or even disappear altogether should they not work out.

As a programmer yourself, you'll know that the first API you write for a package usually needs tweaking once you actually come to *use* it.

This is one great reason to write packages test-first, by the way. In writing a test, you're forced to use your own API! You can usually see right away if it's awkward or laborious, and improve it before it reaches your users.

The Go team are adopting the same approach with `slices`, `maps`, `constraints`, and perhaps any other such packages that come along (`channels` has been suggested).

They'll be introduced first in `x/exp`, so that they're available for Gophers everywhere to use in their programs. If it becomes clear, after people have used them for a while, that changes are needed, there'll still be time to make them.

Once the APIs have stabilised, perhaps over the course of one or two minor releases, the new packages will likely migrate into the standard library.

## Design constraints

Standard library packages are hard to get right. They need to provide significant functionality: a package with just one or two functions isn't really worth it. On the other hand, they shouldn't try to do too much: if a library has dozens of functions, it's hard to find the one you want.

There's also no point providing very simple or trivial functions: users can write those for themselves, without needing to waste valuable standard library real estate. Conversely, very complicated or niche-specific functions won't be useful to enough people to justify their inclusion.

The ideal standard library package, then, contains perhaps a dozen or two functions that are widely useful, but not necessarily trivial to write.

## The `slices` package

Accordingly, then, the `slices` package doesn't contain everything you could ever want to do with slices.

It doesn't contain `Max`, for example (or `Min`). We already know how to write those (see the chapter on operations for details). Let's see what it does contain.

First, to import the package from the `x/exp` repo, we can write:

```
import "golang.org/x/exp/slices"
```

## Comparing slices

As you know, slice types in Go don't support the == operator, so `slices.Equal` now provides a standard way of checking whether two slices are equal:

```
s1 := []int{1, 2, 3}
s2 := []int{1, 2, 3}
fmt.Println(slices.Equal(s1, s2))
// true
```

This is convenient when the element type is `comparable`, but what if it's not? Or what if we want to compare elements in some special way?

In that case, we can use `slices.EqualFunc`, which lets us supply a suitable function to compare any two elements:

```
s1 := []string{"a", "b", "c"}
s2 := []string{"A", "B", "C"}
fmt.Println(slices.EqualFunc(s1, s2, strings.EqualFold))
// true
```

An interesting property of `EqualFunc` is that it's defined on *two* slice types, not one, so we can use it to compare two different kinds of slices.

All we need to do is write our equality function to take *both* element types:

```
s1 := []int{0, 1, 2}
s2 := []rune{'a', 'b', 'c'}
equal := func(n int, r rune) bool {
    return n == int(r-'a')
}
fmt.Println(slices.EqualFunc(s1, s2, equal))
// true
```

Sometimes it's not quite enough just to know whether two slices are *equal*: we might want to ask whether one is greater or lesser than the other, for example.

That's where `Compare` comes in. Given two slices, it returns 0 if they're equal, -1 if the first slice is lesser, or 1 if the first slice is greater.

```
smaller := []int{1, 2, 3}
bigger := []int{1, 2, 3, 4}
fmt.Println(slices.Compare(bigger, bigger))
// 0
fmt.Println(slices.Compare(smaller, bigger))
// -1
fmt.Println(slices.Compare(bigger, smaller))
// 1
```

As you can see, a shorter slice is considered lesser by `Compare`, but what if the slices are the same length? In that case, the comparison depends on the first non-matching element:

```
s1 := []int{1, 2, 3}
s2 := []int{1, 2, 4}
fmt.Println(slices.Compare(s1, s2))
```

```
// -1
```

And this is fine for element types that obey `constraints.Ordered` (that is, they work with the > operator).

But what about other types, or what if finding their relative magnitude isn't so straightforward? For those, we can supply our own comparison function, using `CompareFunc`.

For example, let's compare two slices of integers in a way that ignores their *sign*, and looks only at their absolute magnitude. In other words, these two slices should compare equal according to our scheme:

```
s1 := []int{-1, 2, -3}
s2 := []int{1, -2, 3}
```

We can write a suitable comparison function as follows:

```
func cmp(x, y int) int {
    absX := math.Abs(float64(x))
    absY := math.Abs(float64(y))
    if absX < absY {
        return -1
    }
    if absX > absY {
        return 1
    }
    return 0
}
```

Let's try it out:

```
fmt.Println(slices.CompareFunc(s1, s2, cmp))
// 0
```

And just like `EqualFunc`, we can use `CompareFunc` with two different slice types, given a suitable comparison function.

## Finding elements

Another common operation on slices is *finding* a specified element by index, or simply asking whether it's contained in the slice at all.

`slices.Index` will return the index of the first occurrence of a given element, or -1 if it's not found:

```
s := []int{1, 2, 3}
fmt.Println(slices.Index(s, 2))
```

```
// 1
fmt.Println(slices.Index(s, 9))
// -1
```

Maybe we don't want to search for a specific value, but instead we just want to know the first element for which some arbitrary function returns true. In that case, we can use `IndexFunc` instead of `Index`.

If we don't care about the index, but just want to know whether the element is anywhere in the slice, we can use `Contains`:

```
s := []int{1, 2, 3}
fmt.Println(slices.Contains(s, 1))
// true
```

You might remember that we wrote this very function in a previous chapter. Well, now we don't have to!

## Inserting and deleting

It's also perfectly natural to want to insert or delete elements of a slice, and accordingly the `slices` package provides `Insert` and `Delete` functions.

`Insert` inserts any number of elements at the specified index, returning the modified slice:

```
s := []string{"a", "c"}
s = slices.Insert(s, 1, "b")
fmt.Println(s)
// [a b c]
```

Conversely, `Delete` deletes elements from the first index up to, but not including, the second index:

```
s := []string{"a", "b", "c"}
s = slices.Delete(s, 0, 2)
fmt.Println(s)
// [c]
```

## Cloning and compacting

To create a shallow copy of a slice, we can use `slices.Clone`:

```
s := []int{1, 2, 3}
fmt.Println(slices.Clone(s))
```

```
// [1 2 3]
```

Or if we want to delete duplicate elements from the slice, like the Unix `uniq` command, we can use `slices.Compact`:

```
s := []int{1, 1, 1, 2, 3}
fmt.Println(slices.Compact(s))
// [1 2 3]
```

Note that `Compact` only removes *successive* duplicate elements (just like `uniq`). If the duplicate elements aren't next to each other, they won't be removed:

```
s := []int{1, 2, 1, 3, 1}
fmt.Println(slices.Compact(s))
// [1 2 1 3 1]
```

Again, if we want to override the default == comparison for identical elements, we can supply our own function to `CompactFunc`:

```
s := []string{"a", "A"}
fmt.Println(slices.CompactFunc(s, strings.EqualFold))
// [a]
```

## Growing and shrinking

As you probably know, slices in Go have some *length*, reported by the built-in `len` function. The length of a slice is the number of elements it contains.

A slice also has a *capacity*, which is reported by the `cap` function. You can think of the capacity of a slice as being the number of "slots" it has, even if not all those slots are currently occupied by elements.

A slice will automatically grow its capacity as needed when you add elements to it, which is convenient. But this process involves copying the slice and allocating a new chunk of memory every time it happens, which can result in a lot of unnecessary work.

For efficiency, we sometimes want to increase the capacity of a slice by a large amount in one shot. We can use `slices.Grow` to do this:

```
s := []int{1, 2}
fmt.Println(cap(s))
// 2
s = slices.Grow(s, 10)
fmt.Println(cap(s))
// 12
```

Conversely, if a slice is now a lot smaller than it used to be, and we'd like to reclaim that unused memory, we can use `slices.Clip` to reduce its capacity:

```
s := make([]int, 100)
fmt.Println(cap(s))
// 100
s = slices.Delete(s, 0, 100)
s = slices.Clip(s)
fmt.Println(cap(s))
// 0
```

## Sorting

As we saw in the chapter on functions, the introduction of generics means we're no longer required to use the `sort` package to sort slices. Instead, we get a new family of sorting functions in the `slices` package.

For straightforward, in-place sorting of ordered elements, we can use `slices.Sort`:

```
s := []int{3, 1, 2}
slices.Sort(s)
fmt.Println(s)
// [1 2 3]
```

Maybe the elements are not one of the ordered types, though, or maybe we just want to customise the sorting for some other reason.

In this case, we can pass our own function to `SortFunc`:

```
less := func(x, y int) bool {
    return x < y
}
s := []int{3, 1, 2}
slices.SortFunc(s, less)
fmt.Println(s)
// [1 2 3]
```

For a *stable* sort, where identical elements are guaranteed to remain in their original order, we can use `SortStableFunc`.

To detect if a slice is *already* sorted, use `IsSorted` or `IsSortedFunc`:

```
s := []int{1, 2, 3}
fmt.Println(slices.IsSorted(s))
// true
```

## Searching

One benefit of having a sorted slice is that we can look for elements using a *binary search*. This is a common-sense way of looking for things that we all use with sorted data.

For example, imagine you're looking for a certain word in a dictionary. If you open the dictionary roughly in the middle, at some random word, then you immediately know which half of the dictionary your target word is in, don't you?

It either comes before the word you landed on, or after it. So you just cut your search space in half, with a single lookup. That's the "binary" part.

Let's say you now know that your target word is in the first half. So you can open the dictionary halfway through *that* section and see what word you land on. Repeat this "binary-chopping" process until you hit the target word.

This is obviously much faster than starting at page 1 and looking at every page in turn, especially if you're curious about the word "zyzzyva" (a kind of South American weevil, since you ask).

Given a sorted slice, then, we can use `slices.BinarySearch` to quickly find the index of a certain element:

```
s := []int{1, 2, 3}
fmt.Println(slices.BinarySearch(s, 2))
// 1
```

Alternatively, we can use `BinarySearchFunc` to find the index of the first element for which some given function returns true.

If the element is *not* in the slice, then `BinarySearch` and `BinarySearchFunc` return the index where the element *would* be, if it were inserted at the correct place. For example:

```
s := []string{"a", "c"}
fmt.Println(slices.BinarySearch(s, "b"))
// 1
```

This is a little tricky, as it means you can't tell from the return value whether or not the element actually *was* found.

To do that, you have to compare the target with the element at the returned index:

```
s := []string{"a", "c"}
i := slices.BinarySearch(s, "b")
if s[i] == "b" {
    fmt.Println("found it!")
} else {
    fmt.Println("no luck")
}
// no luck
```

## Exercise 9.1: Contain your excitement

Use your knowledge of the `slices` library now to solve Exercise 9.1.

Your job here is to write a generic function `ContainsFunc` that operates on slices of arbitrary type. Given a function argument, `ContainsFunc` should return true if the function returns true for any element in the slice, or false otherwise.

For example:

```
s := []rune{'a', 'B', 'c'}
fmt.Println(ContainsFunc(s, unicode.IsUpper))
// true
```

See Solution 9.1 if you'd like something to compare with your answer.

## The `maps` package

Like the `slices` package, `maps` is small but purposeful. To import it, we can write:

```
import "golang.org/x/exp/maps"
```

Let's run through what's available in `maps`, with a few examples.

## Comparing maps

We can now compare two maps for equality, using `maps.Equal`:

```
m1 := map[string]bool{
    "generics": true,
}
m2 := map[string]bool{
    "generics": true,
}
fmt.Println(maps.Equal(m1, m2))
// true
```

And to supply our own definition of "equality", or when using value types that don't work with ==, we can call `EqualFunc` instead.

Since map keys must be comparable anyway, there's no need to provide your own function to test *keys* for equality. They can just be compared with ==.

## Enumerating keys and values

One very common operation on a map is to get either its keys or its values as a slice, and we can now do this easily using Keys and Values:

```
m := map[string]bool{
    "generics": true,
    "classes":  false,
}
fmt.Println(maps.Keys(m))
// [generics classes]
fmt.Println(maps.Values(m))
// [false true]
```

As you can see, and as you'd expect if you're familiar with the behaviour of Go maps, the keys and values are in no particular order.

So if you need to pair up the slice of keys with the slice of values for some reason, Keys and Values aren't the right way to do that. Instead, loop over the map with range and build up both slices within that loop.

We could use maps.Keys right now to improve the set type we developed earlier in this book, couldn't we?

For example, we can simplify the implementation of the Members method by eliminating the loop:

```
func (s SetC[E]) Members() []E {
    s.mutex.RLock()
    defer s.mutex.RUnlock()
    return maps.Keys(s.data)
}
```

## Finding keys or values

There's no maps.Contains function to tell us if a map contains a given key, because we can already do that directly:

```
_, ok := m[k]
// ok is true if m contains key k
```

But it can often be useful to know if a map contains a given *value*. Now there's an easy way to do that, by combining `maps.Keys` with `slices.Contains:` .

```
m := map[string]int{"answer": 42}
if slices.Contains(maps.Values(m), 42) {
    fmt.Println("found it!")
}
// found it!
```

## Deleting map entries

The built-in `delete` function already deletes a given key from a map, so the `maps` package doesn't need to duplicate that.

However, we can delete any map entries for which some function returns true, by passing it to `DeleteFunc`:

```
m := map[string]bool{
    "generics": true,
    "classes":  false,
}
findFalse := func(k string, v bool) bool {
    return !v
}
maps.DeleteFunc(m, findFalse)
fmt.Println(m)
// map[generics:true]
```

Or if we just want to remove *all* entries from the map, we can call `maps.Clear`.

## Cloning and copying

Just like with slices, we can use the `Clone` function to create a shallow copy of a map:

```
m1 := map[string]float64{
    "e": 0.5,
    "μ": 105.7,
    "τ": 1776.9,
}
m2 := maps.Clone(m1)
fmt.Println(maps.Equal(m1, m2))
// true
```

There's no `Compact` function, since there can't be any duplicate keys in a map, but there is a `Copy` function that copies all the entries from one map to another.

If any of the copied keys already exists in the destination map, it will be overwritten:

```
m1 := map[int]bool{1: false, 2: true}
m2 := map[int]bool{1: true}
maps.Copy(m1, m2)
fmt.Println(m1)
// map[1:true 2:true]
```

Note that the first argument to `Copy` is the destination map, and the second is the source. So we write, for example, `Copy(to, from)`, not `Copy(from, to)`.

## Idioms

Now that we have these wonderful new packages, will they change the way we write idiomatic Go code? You bet.

For example, to create a copy of a slice, we used to have to write, laboriously:

```
b := make([]T, len(a))
copy(b, a)
```

Instead we can now write just:

```
b := slices.Clone(a)
```

Similarly, to delete a number of consecutive elements from a slice, we used to have to write something like this:

```
s := []int{1, 2, 3, 4}
s = append(s[:1], s[3:]...)
fmt.Println(s)
// [1 4]
```

This is much nicer:

```
s = slices.Delete(s, 1, 3)
```

And we'll never have to write loops like this again:

```
s := []int{1, 2, 3, 4}
for _, v := range s {
    if v == 2 {
        fmt.Println("found it")
    }
}
```

Because it's just:

```
if slices.Contains(s, 2) {
    fmt.Println("found it")
}
```

And so on. Similarly, we won't have to write loops to extract the keys or values of a map as a slice, since we now have `maps.Keys` and `maps.Values`.

## Exercise 9.2: Merging in turn

Over to you now: use the `map` library to solve Exercise 9.2.

This task is about writing a `Merge` function that takes any number of maps (all of the same arbitrary type) and merges them together, returning the result.

In other words, the result returned by `Merge` is a map containing all the key-value pairs in all its arguments.

For example:

```
m1 := map[int]bool{ 1: true }
m2 := map[int]bool{ 2: true }
fmt.Println(Merge(m1, m2))
// map[1:true 2:true]
```

If there are any conflicts (that is, if the maps have any key in common), then the "later" map wins. In other words, the maps are merged in the order that they're passed to `Merge`. For example:

```
m1 := map[int]bool{ 1: false }
m2 := map[int]bool{ 1: true }
fmt.Println(Merge(m1, m2))
// map[1:true]
```

If you'd like to see one way of solving this, take a peek at Solution 9.2.

## The future

Of course the introduction of a major new feature like generics is going to have effects that ripple all the way through the standard library, and other libraries. These changes won't happen immediately, for the reasons we discussed earlier in this chapter.

The old `sort` API, using `interface{}`, is partially superseded by the functions in the `slices` package that we explored earlier in this chapter. When `slices` and other generics-based packages migrate into the standard library, `sort` will likely be deprecated.

A new generic `Heap` type is proposed for the `container/heap` package, and other new kinds of container types probably won't be far behind.

Since the `context` package already provides a (rather awkward to use) key-value store using `interface{}`, there is now scope for a generic, but type-safe equivalent. An intriguing proposal suggests that a clever implementation could avoid conflicts between identical keys, but this proposal is currently on hold.

## Bytes versus strings

There's a lot of awkward duplication and overlap in the standard library APIs because of the need to handle two similar types: `string` and `[]byte`.

For example, many of the same operations exist in both the `bytes` and the `strings` package, and use exactly the same logic.

It seems sensible to eliminate much of this duplication by introducing some kind of interface to describe all types that are *byte sequences*:

```
type Bytes interface {
    ~[]byte | ~string
}
```

Instead of having to have a distinct implementation of, say, `Index`, for each type, we could write just one:

```
func Index[T Bytes](s, sep T) int {
```

This would avoid a lot of code and API duplication, and also eliminate otherwise unnecessary conversions back and forth between `string` and `[]byte`.

This change isn't on the roadmap at the moment, but it's a nice idea.

## Transitioning APIs to generics

Go maintainer Russ Cox has pointed out that there are a number of other existing APIs that could obviously benefit from generics:

- `sync.Pool` would naturally be `sync.Pool[T]`
- `sync.Map` would naturally be `sync.Map[K, V]`
- `atomic.Value` would naturally be `atomic.Value[T]`
- `list.List` would naturally be `list.List[T]`
- `math.Abs` would naturally be `math.Abs[T]`

- `math.Min` would naturally be `math.Min[T]`
- `math.Max` would naturally be `math.Max[T]`

The compatibility promise means that these can't simply be changed in place: something else will need to happen.

One suggestion is to add an `Of[T]` suffix. For example, `sync.Pool` would become `sync.PoolOf[T]`).

Another option is to publish a v2 version of the affected library, so that users would have to explicitly import, for example, `math/v2`, to use the new generic APIs.

A still more ingenious idea is to use type aliases to turn the existing types into specific instantiations of a new generic type:

```
type Pool[T any] ...
type Pool = Pool[interface{}]
```

It's clear that some big changes will happen eventually, rather than immediately, and it's possible that there'll be different solutions for different libraries.

For example, the `math` library might need to go to v2, just because there are so many functions that it would be awful to have to provide, document, and maintain two versions of each.

On the other hand, problems like `sync.Pool` might end up being solved with type aliases. Time will tell.

## Third-party libraries

Many of the points we've discussed about the effect of generics on the standard library also apply to third-party libraries.

Here are some changes to the wider Go ecosystem that we might expect to see soon:

1. New general-purpose libraries, providing useful miscellaneous operations and data structures.
2. Elimination of duplicated code and APIs for different specific types, and a much reduced need for code generation.
3. Replacing many or most occurrences of `interface{}` / `any` with type parameters.

These improvements will be especially noticeable in *framework* libraries, which have previously been awkward to integrate with user programs because of the limitations of the pre-generics type system.

For example, the popular `go-kit` microservices framework uses an abstraction called `Endpoint`, which represents a particular "verb" or behaviour provided by a service.

Operations on endpoints need to pass `Request` and `Response` objects back and forth. The exact schema of these request and response structs, though, clearly depends on the service and the endpoint.

For example, a "log in as user" request might contain credentials, while the response might contain a token or cookie. But a "get product details" request might contain a product ID or title, while the response would contain product-specific fields.

Without generics, it's impossible (or just impossibly awkward) to write APIs on arbitrary structs like this. Accordingly, libraries like `go-kit` have been forced to use `interface{}` virtually everywhere, creating ugly code and annoying paperwork for users.

*With* generics, this becomes straightforward. Users can define the request and response structs they need for their various endpoints, and `go-kit` can provide generic operations on them.

This suggests that the APIs of many popular libraries will become easier and more convenient to use thanks to generics. We will likely also see entirely new libraries to do things that no one has yet thought of.

## Takeaways

- Go 1.18 introduces a new official (but not yet standard) library package, `constraints`, containing predefined constraints useful for writing generic functions and types.
- The `constraints` package is available in the "experimental" repo, `golang.org/x/exp`.
- Two more packages, `slices` and `maps`, accompany it, for common operations on container types.
- The `slices` package provides `Equal` / `EqualFunc` for equality checking, `Compare` / `CompareFunc` for relative magnitude, and `Contains` / `Index` / `IndexFunc` for finding elements.
- Also in `slices` are `Insert` / `Delete` for modifying slices, `Clone` for copying, `Compact` / `CompactFunc` for stripping consecutive duplicates, and `Grow` / `Clip` for managing capacity.
- For sorting, `slices` provides `Sort` / `SortFunc` / `SortStableFunc`, and the boolean functions `IsSorted` / `IsSortedFunc`.
- The `slices` package also includes `BinarySearch` and `BinarySearchFunc` for efficient searching of sorted slices.
- The `maps` package provides `Equal` / `EqualFunc` for equality checking, and `Keys` / `Values` for enumerating keys or values.
- Also in `maps` are `Clear` / `DeleteFunc` for deleting entries, and `Copy` / `Clone` for copying entries.

- The new packages eliminate the need for some very common loop operations in Go, such as checking whether a slice contains a value, or getting the keys of a map as a slice.
- The standard library has already adopted `any` as the new name for `interface{}` throughout, and may adopt generics in a number of places, especially to simplify operations on byte sequences.
- Some standard library APIs can be updated to use generics without affecting user code; others may require duplicating existing APIs, or using new module versions.
- Many third-party libraries will also be changing to take advantage of generics, especially "frameworks" such as `go-kit` which were always *implicitly* generic, but were awkward to use because of pervasive empty interfaces.

## Review questions

1. Are the `slices` and `maps` packages part of the standard library? If not, how do we import them?
2. Name the functions from the `slices` library that we can use to:
   - Ask whether a slice contains a given element
   - Ask whether two slices are equal
   - Insert a new element at a given position
   - Delete a sequence of elements between two specified indexes
3. Name the functions from the `maps` library that we can use to:
   - Get the keys of a map as a slice
   - Delete all map entries for which some given function returns `true`
   - Copy all the entries from one map into another, overwriting any duplicate keys
   - Ask whether two maps are equal according to some given function that compares any two values
4. Give an interface definition whose type set contains all byte-sequence types (that is, `[]byte`, `string`, and all types derived from them). Why is such an interface useful?
5. Can generic types or functions replace `interface{}` or `any` wherever it's used in code? Suggest some cases where this wouldn't be possible, and explain why.

# 10. Questions

*Generics are an incredibly useful addition to the language that I'll almost never use.*
—Scott Redig

We've covered a lot of ground already in this book, and I hope you've learned most of what you wanted to know about generics in Go, at least in theory.

In this final chapter, then, we'll try to answer some of the most common questions about putting generics into *practice*.

## How much do I need to know about generics?

Not as much as some have feared!

The fact that you're reading this book implies that you're at least curious about the subject, but not everyone feels that way.

Some people didn't ask for generics, didn't want it, and just want generics to get off their lawn. How much new stuff will they still have to *learn*, though, even if they don't want to use generics themselves?

What you've read so far in this book is probably considerably more than any working Go programmer will actually *need* to know about generics. As we've seen, from the user's point of view, you can call generic functions or use generic types, in most cases, without even knowing that they *are* generic.

If you want to *write* generic functions and types, you'll need to know a little more, but not a lot. Knowing the type parameter syntax, and a few of the most commonly used constraints (`comparable`, `Ordered`), will be enough to get you started. And, as we've discussed, most people probably won't ever need to write generic functions or types.

While you can become a competent software engineer by writing a lot of code, to become a master you need to *read* a lot of code. In order to understand the code you read, you need to be familiar with every aspect of Go syntax and usage, and that now includes generics.

So to some extent, to *know Go* requires knowing generics, even if it's not a language feature that you often need to use in practice. But that doesn't mean that you need to memorise every detail of, for example, the rules of type inference, or weird syntactic edge cases involving parsing ambiguity.

It's fun to learn about that postgraduate-level stuff, for some definition of "fun", but not actually necessary unless you're planning to write a Go compiler yourself.

## Will generics drastically change the way I write Go?

Probably not.

There will, as we saw in the chapter on libraries, be some important changes to the standard library that affect the majority of Go users.

For example, the `slices` and `maps` package introduce completely new APIs, albeit small ones. However, it's not essential for you to *use* them: they'll just save you some time and lines of code.

The other changes to the standard library will likely be rolled out gradually, incrementally, and in a backwards-compatible way.

So if you're not in a position to learn a whole bunch of new stuff just at the moment, that's okay. You'll be able to take advantage of the new or updated APIs in your own time and at your own pace.

And generics is a pretty niche thing anyway. No one should feel that just because generics are there, they're somehow expected to *use* them.

After all, we don't use goroutines in programs that don't need them, and indeed most programs don't. But when we *do* need them, it's great to have them available.

## Do I need to change my existing code?

No.

The backwards compatibility promise means that no change to Go 1.x, including the introduction of generics, will break existing programs, with a few understandable exceptions, such as security fixes.

*Should* you change your existing code to use generics, though? That depends. If you currently *have* a problem that generics would solve, presumably you're currently solving it some other way.

If solving your problem with generics would bring enough benefit to justify the engineering work, then go ahead. Otherwise, if it isn't broken, you don't need to fix it.

Keep in mind also that the use of generics makes your program more complicated: it introduces more *abstraction*. That's not free.

> *Genericity introduces abstraction, and needless abstraction introduces complexity. Move cautiously!*
> —Robert Griesemer

Abstractions can be useful, of course, and software engineering is the *art* of creating useful abstractions. But they come at a cost to readability and simplicity.

We should only use any abstraction when the benefits outweigh the costs, and this applies equally to generics.

For example, there's not necessarily any point in replacing a function that takes an interface type like `io.Reader` with one that takes a type parameter.

Interfaces already solve this problem. There's no benefit from writing a generic function here to offset the extra complexity it introduces.

It *can* be worth replacing interfaces with generics in some situations. For example, if it gives you improved type safety, because you're now dealing with some specific type instead of an interface.

Similarly, if generics would allow you to make more efficient use of memory, or significantly improve performance, it's worthwhile. We'll have more to say about performance later in this chapter.

## Should I use `any` instead of `interface{}` now?

Yes.

Generics aside, if your existing code uses `interface{}`, you can replace it with `any` throughout, just as the standard library has done.

For example, a function like this:

```
func Print(args ...interface{}) {
```

can now be refactored very straightforwardly to:

```
func Print(args ...any) {
```

This doesn't make the function generic, or have any other effect on it at all. `any` is simply a type alias for `interface{}`, so you can (and should) use the shorter name anywhere you previously used the longer one.

In brief, `any` is the new name for `interface{}`.

## When should I start adopting generics?

Whenever you like.

For new projects, you can start writing generic code as soon as you like. If you have existing projects that users rely on, though, and you're planning to update them to use generics, read on.

We've seen that the Go 1 compatibility promise puts a high value on not *changing* Go so as to break existing code, but of course that doesn't stop Go introducing *new* features, such as generics.

The question, then, is how quickly we, as library authors, should *adopt* those features while still keeping our users happy.

Unfortunately, not everyone can upgrade to the current version of Go as soon as it's released. Some are constrained to use the version bundled with their OS distribution, which can be very old indeed by Go standards: as many as five or six minor versions behind.

Others may be *able* to upgrade, for example by building Go from source, but are prohibited from doing so by policy. For example, some companies have a lengthy approval process that means their target Go version may lag a year or more behind the latest release.

So even though Go has generics, that's not the same as saying that every Go user will be able to compile your program with generics right away.

If this is a problem for you, it would be wise to plan your adoption of generics carefully.

A common rule of thumb for library authors is that they shouldn't have their code *require* any new Go feature until that feature has been available for at least two minor versions.

For example, since generics were introduced in Go 1.18, it might make sense not to update your existing package to *require* generics until, say, Go 1.20 has been released.

Since the usual release schedule is twice a year, this means your users will have a year to plan their Go upgrade. Some might need longer, but we have to draw the line *somewhere*, and this seems reasonable.

That doesn't mean we have to wait a year to start using generics, of course. It just means that if we introduce generics to our packages, we need to provide alternatives for people who can't yet upgrade.

We'll see some possible ways to do this in the next section.

## Can I adopt generics without upsetting my users?

Yes.

The Go modules system ensures that if you publish a new version of your module today, users won't be automatically upgraded to it. So you won't break their builds, which is good.

If they *explicitly* upgrade their dependency to your latest version, and they can't build it because their Go is too old, though, that's not something you need to worry about.

Users are always allowed to break their *own* builds with speculative dependency updates, but it wouldn't be okay for *us* to break them by pushing changes, and module versioning prevents that.

So if you're writing a new version of your module taking advantage of generics, but with *no visible changes for users*, go ahead and publish it now. Those who can use it will be able to use it, and those who can't won't be any worse off.

However, if you're planning a new version with significant improvements or new features, then if you *also* use generic code, that may prevent many people from using your update.

If the new features *rely* on generics, then of course there's nothing you can do about that. But if they don't, then you have another option.

You can provide one version of your module with the new features, without generics, and another with the new features but *without* generics. The Go team suggests:

> *If you are updating your package to use generics, please consider isolating the new generic API into its own file, build-tagged for Go 1.18, so that Go 1.17 users can keep building and using the non-generic parts.*
> —Russ Cox

## Is it safe to use generics in production?

No.

It's not *safe* to do *anything* in production, in the sense that you can guarantee there won't be any problems. A more helpful question might be "what additional risk am I taking on by using generics in production?"

But even then, the answer still has to be "some", doesn't it? Any new feature of any software is bound to have bugs, and Go is no exception.

The Go team, it's worth saying, are very experienced engineers who care deeply about the correctness and reliability of Go.

In fact, they're so smart, they know they can't catch every bug in advance:

> *Production uses of generics should be approached with appropriate caution. This is not a criticism of the team's excellent work. It is simply an observation that generics is different from most Go changes.*
> —Russ Cox

Whenever changes are made to Go, the updated toolchain is used to build a vast corpus of existing Go code, to make sure that nothing weird happens.

The difference with generics is that here, there *is* no existing code to check against. So there's no way the Go team can achieve their usual level of confidence in the change.

We should expect *bugs* in any new Go version, naturally, and especially this one. That said, though, almost all such bugs are usually trivial and harmless.

For example, the compiler might give a slightly wrong or unhelpful error message for certain invalid code. It might give a bogus error for code that the spec says *should* be valid, or the compiler itself might panic or crash in weird edge cases.

That's not the sort of bug that any of us should really worry about. And, while it's annoying if you can't build your project because of a compiler bug, that's not the same thing as a production outage.

So don't worry about generics bugs causing *your* programs to crash unexpectedly or give the wrong results. We can't rule that possibility out, but it's very unlikely.

## Will generics change again in the future?

No.

Even if we're not overly concerned about generics crashing our programs, it's quite reasonable to ask "What if I invest time and effort in adopting generics, and then my work is invalidated by future Go changes?"

First, the compatibility promise means that whatever changes come to the Go language itself in future, they won't affect any code you write today.

Second, as we saw in the previous chapter, the generics-related changes to the standard library are being made quite cautiously and incrementally.

If you're risk-averse, then, you might want to hold off on using the `slices` and `maps` packages, at least while they're still in the experimental repo. Their APIs probably won't change much, but they might well change a little.

Once they arrive in the standard library, though, they'll be essentially frozen. New functions may be *added* later, of course, but that's fine, because it won't break anything we've already written.

So, while caution is always advisable, there's no reason not to start enjoying the benefits of generics right away.

## What impact does generics have on performance?

Some.

There's more than one question to unpack here, though, isn't there? One thing we could ask is "How much does the introduction of generics to Go affect the performance of programs *in general*?"

Another is "How much does *using* generics in my program affect its performance?"

You might be surprised to learn that the answer to both questions is, in fact, "Not at all". Programs run no faster or slower when compiled with the latest version of Go, whether or not they use generics.

Maybe this shouldn't really be surprising, though, since we already know that generics is a purely compile-time phenomenon.

The compiler *instantiates* every generic type or function call on some specific type. There are no generic types or functions in compiled Go programs, as we've discussed.

So while your binary may be fractionally *larger*, if it contains multiple compiled versions of some function, that binary will run no more slowly than if it had been written explicitly on a specific type.

However, that's not the end of the story. Your program may actually run *faster*, if you can use generics to eliminate reflection or interface values.

For example, the `tidwall/btree` project provides benchmarks suggesting that the version using generics is roughly twice as fast as the one using `interface{}`.

This makes sense, since interfaces involve a run-time indirection that's bound to be slower than using concrete values directly.

Projects that can use generics to eliminate the need for *reflection* may see an even more significant speedup as a result.

## Does generic code compile more slowly?

Another important question is how generics affects *compile* performance. Go was designed with speed of compilation very much in mind, especially for large codebases.

We'd expect generics to slow down compilation a little; after all, the compiler is *doing* more. But we'd be disappointed if this performance dip were significant.

The Go team's own figures suggest compile times are increased by around 15% in Go 1.18, and most of that change is due to generics.

There isn't much real-world data on this question yet, but a maintainer of the Juju project reported a 6% increase in total compile, build, and link times, and this is probably closer to what most users will experience.

That said, there's been no particular effort to improve compiler performance in Go 1.18: this is scheduled for the Go 1.19 development cycle. So the slowdown, such as it is, will likely be reduced in 1.19, though probably not to zero.

Does it make any difference to compile speed whether your program *uses* generics or not? Yes, a tiny difference, but not one you're likely to notice:

> There is certainly a very small extra amount of work to instantiate generic types and functions. But it should mostly be in the noise, similar to adding one extra function to a normal Go program; hardly noticeable in a full compile. —Dan Scales

The vast majority of Go projects build extremely quickly anyway, of course, in just a few seconds, and in practice it's unlikely that most users will notice any difference at all.

So the answer to the performance question is that Go with generics will perform faster, slower, or about the same, depending on exactly how you ask the question! That may not be a very satisfying answer, but it's at least an accurate one.

## Why didn't they use angle brackets?

The question here is about the syntax for writing a list of type parameters. Here's what it looks like in Go, as you'll recall from previous chapters:

```go
type Bunch[E any] []E
```

And the suggestion, or complaint, is that it should be instead:

```go
type Bunch<E any> []E
```

In other words, why isn't the list of type parameters delimited with angle brackets (<>) instead of square brackets (`[]`)?

This is the syntax used by some languages including C++ and Java. So what's wrong with that?

In fact, the original Go generics proposal used parentheses (`()`) for type parameter lists, for symmetry with function parameter and result lists. It's reasonable to argue that new features in Go should act like *Go*, not Java.

The trouble with parentheses is they it can lead to code that's a little awkward to read:

```go
func Foo(T, U any)(p T, q U) (T, U) {
```

Maybe this is slightly too many parentheses. It was decided, then, to use a different delimiter for type parameter lists, to help them stand out in code.

Unfortunately, not every imaginable delimiter character is available: some are already used for other things in Go.

The compiler has to be able to parse Go code unambiguously, and using < and > here would be a problem, because they're already valid operators in Go expressions.

For example, consider an assignment statement like this:

```
a, b = w < x, y > (z)
```

Is this a pair of expressions (`w < x` and `y > (z)`), or an instantiation of a generic function call `w<x, y>` passing the argument `z`?

There's no way to know, so using angle brackets for generics would lead to ambiguities that can't be resolved. See the type parameters proposal for more details about this.

So while angle brackets might have made Go generics a tiny bit more familiar to people who were already used to generics in C++ or Java, for whatever that's worth, the option just wasn't available.

By all means register your disgust on the internet, but this is where we are. Square brackets are fine. You'll get used to them.

## What still isn't possible with generics?

Lots of things.

One problem that generics doesn't solve, at least in current versions of Go, is that of *option types*, sometimes called "Maybe" types. That is, types that can either contain some value, or no value.

For example, in some languages you can write code something like this, if `x` were a value of an option type:

```
switch x {
case Some(x):
    fmt.Println("Got value", x)
case None:
    fmt.Println("No value for you")
}
```

We can't really do that in Go, though we can approximate it with pointer types, where `nil` represents "no value for you".

Generics doesn't help here either, since even a generic type must be instantiated on something specific, and most Go types can't be `nil`.

The Go idiom in this situation, though, is for functions to return "something and error", where the error value tells you whether or not there's any data in the other value.

You could think of the "something and error" pair as *itself* being a kind of option type. Obviously, this won't satisfy Hacker News comment warriors, but perhaps that's not a good thing to optimise for.

It seems unlikely that anything like full-fledged option types will come to Go, but you never know. In the meantime, there's nothing stopping us implementing our own struct-based option types, along the lines of the optional library.

And generics makes it a lot easier to write such libraries, so perhaps we'll see more use of option-style types in Go.

Another thing that people often miss in Go is *enumerated types*, or "enums", which are types that can hold only one of a set of allowed values.

For example, an enum variable representing a DNA letter might be able to take only the values A, C, G, or T.

It's not possible to do this using the built-in types and operators in Go, and the most common approximation to it is to define a number of constants representing the "allowed" values. That way, at least it's clear to users what values they're supposed to restrict themselves to.

However, there's no way to actually prevent people assigning some non-allowed value, which would be impossible with "real" enums. Again, we can define a struct type and provide validating accessor methods to simulate enums in Go, but it's not entirely satisfactory.

An extension of this idea is the *union type*, or *tagged union*. This is like an enum, but rather than containing one of a set of allowed values, it contains a value of one of a set of allowed *types*.

For example, a given `Animal` might be either a `Cow`, or a `Chicken` (but not both).

If you think about it, a Go pointer is a kind of union, isn't it? It can contain either a valid pointer, or `nil` (but not both).

Indeed, if we had unions in Go, we could use them to implement option types. We don't, though, and again, generics don't solve any of these problems, if they are problems.

Another thing that Go still doesn't have, or want, is *template metaprogramming*, or *macros*, which we might describe loosely as "executing code at compile time".

While generics provides a very simple form of templating, as we've seen in this book, it's limited to instantiating type parameters.

There's no way with Go generics to do any kind of computation at compile time, or make decisions based on conditions. We can't choose between different implementations of a function (*specialization*), or affect the build process in any way other than with build tags.

While metaprogramming is a powerful tool, it also introduces a great deal of complexity, and makes it much harder to build large programs efficiently.

These downsides suggest that a facility like template metaprogramming is very unlikely to ever be added to Go.

Something else that's missing from Go right now is the ability to write *generic methods*, as we saw in an earlier chapter. This has made a lot of people very angry, and has been widely regarded as a bad move.

However, Jaana Dogan has suggested a neat workaround using a pattern called facilitators.

What else is missing? Go doesn't have *generic packages* (that is, the ability to choose which Go package to import based on a type parameter).

This facility seems useful at first glance, but generics just doesn't mesh well with packages:

> *It's very awkward for a function in one instantiation of a package to return a type that requires a different instantiation of the same package. And there is no particular reason to think that the uses of generic types will break down neatly into packages. Sometimes they will, sometimes they won't.*
> —Type Parameters Proposal

There are also no *operator methods*: for example, we can't write some method on a struct type that lets us use Go operators like >, ==, or + with it. That would be cool, but it's too late to add this to Go now.

Since Go doesn't have inheritance, there's also no *covariance* or *contravariance* (and if you don't know what these are, then I doubt you'll miss them).

There's no *currying* (partial instantiation) or *variadic type parameters*. And the same applies to many arcane and exotic generics-related features that exist in other languages. You name it, Go doesn't have it!

It's not that any of these things are bad in themselves. Features are nice, but a language that has more features doesn't *necessarily* make us more productive, or result in better software. In fact, it can make it harder to learn that language in the first place.

There's a sweet spot for programming languages where they're powerful enough to solve any problem fairly efficiently, but still small and simple enough to be easy to learn. For most of its users, Go hits that spot.

## Will there be a "Go 2"?

No.

> *There are no plans for anything called Go 2.*
> —Ian Lance Taylor

Since the first release of Go, people have had *opinions* about it. Boy, have they had opinions.

Most of them, of course, are to the effect that Go should be different in some way. These are usually framed as either "I hate Go because it doesn't have feature X", or alternatively "I love Go, but I wish it had feature X".

The vast majority of these suggestions have not been adopted, either because they weren't really actionable in themselves ("error handling sucks"), or because the benefits of adopting them wouldn't justify the costs.

So if you're considering proposing a change to Go, please don't, as a refusal often offends.

But even just reviewing and responding to such a never-ending stream of proposals is more than the Go team can keep up with, which is one reason why they've provided the language change proposal template.

Usually, in the process of filling out this paperwork, the proposer either gets bored or thinks better of their idea (or realises it's a slightly different form of something that's already been rejected).

Some major changes, though, *have* been adopted, eventually (modules, error wrapping, generics). And there are several proposals that nearly made it, but despite well-considered designs and plenty of popular support (`try`), ultimately didn't *quite* clear the bar.

Most proposals, though, would break compatibility with existing code in one way or another, and so there's no point seriously discussing them. That doesn't mean that many of these proposals aren't excellent, or even that they can't or won't be adopted. They just won't be adopted in *Go*.

Accordingly, such proposals, rather than being rejected outright, are now tagged with the label "Go 2", meaning that they won't be adopted in Go 1, but are still worthy of consideration. This has helped mitigate criticism of the Go team for being stuck-in-the-mud, or "unfriendly" to newbies with bright ideas.

This was a good idea on the face of it, but also generated a widespread buzz that a "Go 2" might be imminent, which in turn prompted even *more* people to jump in with ingenious suggestions.

Compounding this misunderstanding, the Go team even wrote a couple of blog posts giving the perhaps unfortunate impression that a version 2.x release of Go was in prospect. It's not, and never will be.

In general, successful programming languages don't release a version 2, which by definition (at least, by the definition of semantic versioning) would be incompatible with code written for version 1. When this does happen (as it has with Python, for example), the effects are usually regrettable.

First, it renders most existing software useless, destroying the ecosystem that made the language successful. Changes significant enough to justify a "version 2" usually can't be backported to older code using automated tools.

And a successful language will have many users who are so heavily invested in it that they can't, or won't upgrade to the new version for many years, if ever.

Second, just because something is better doesn't mean it will automatically gain widespread adoption, or even any adoption (see IPv6 for details).

What the advent of version 2 *will* do is severely put off those who were considering adopting the language, at least until the situation has stabilised. So existing users quit, and the inflow of new users stops.

Third, the new version never completely displaces the old. So the upgrade fractures whatever language community remains, along with the software ecosystem.

Essentially, you end up with two different languages with the same name, and it would usually have been much better just to give the new language a new name.

So what *will* happen with all those "Go 2" proposals?

One of two things is likely. If it's possible to water down or redesign the proposal in such a way that it could be introduced to Go in a non-breaking way, then that may well be done.

If not, one day some of these ideas may make it into a brand new language that, while clearly inspired by Go, will also be radically different from its predecessor.

If you're going to *make* breaking changes, you may as well go big or go home. This hypothetical future language may be as different from Go as Go is from, say, C.

And that's the point: this language won't *replace* Go, any more than Go has replaced C. It will take the best ideas from Go, combine them with the best proposed improvements, and incorporate the current state of the art in programming language design.

The result will be something just as unique and wonderful as Go. It just won't be Go 2.

## Okay, I'm sold. Where can I find out more about Go?

I'm glad you asked. See the next section for a list of other books and learning resources that may interest you. And thanks for reading this far.

It's been great having you along for the ride. See you in another book!

# About this book

## Who wrote this?

John Arundel is a Go teacher and mentor of many years experience. He's helped literally thousands of people to learn Go, with friendly, supportive, professional mentoring, and he can help you too. Find out more:

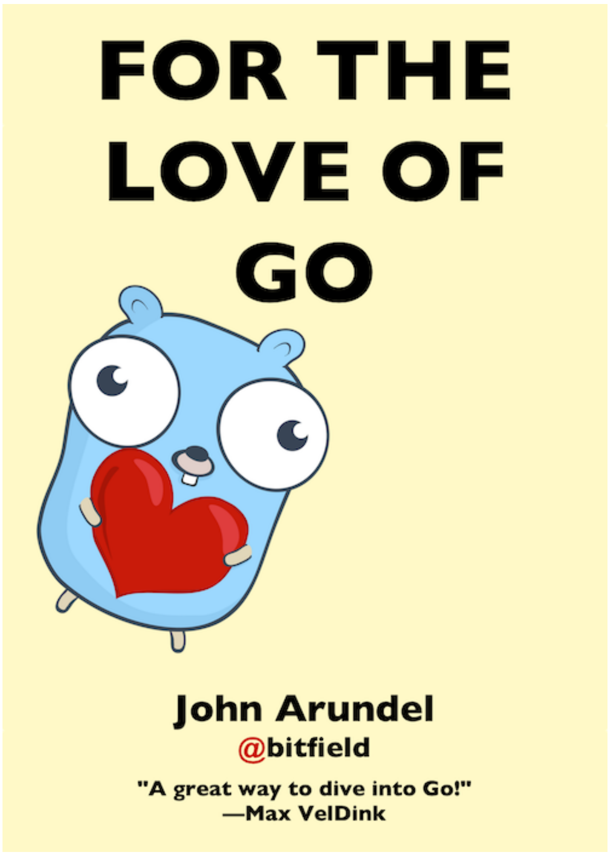
- Learn Go remotely with me

## Feedback

If you enjoyed this book, let me know! Email go@bitfieldconsulting.com with your comments. If you didn't enjoy it, or found a problem, I'd like to hear that too. All your feedback will go to improving the book.

Also, please tell your friends, or post about the book on social media. I'm not a global mega-corporation, and I don't have a publisher or a marketing budget: I write and produce these books myself, at home, in my spare time. I'm not doing this for the money: I'm doing it so that I can help bring the power of Go to as many people as possible.

That's where you can help, too. If you love Go, tell a friend about this book!

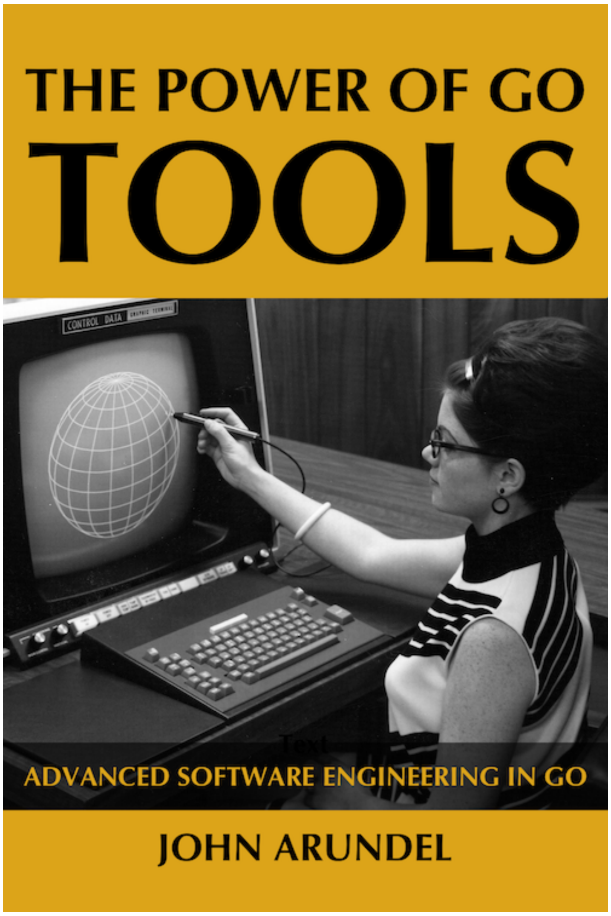

FOR THE LOVE OF GO
John Arundel
@bitfield
"A great way to dive into Go!"
—Max VelDink

with this fun, readable, and easy-to-follow guide.

Throughout the book we'll be working together to develop a fun and useful project in Go: an online bookstore called Happy Fun Books. You'll learn how to use Go to store data about real-world objects such as books, how to write code to manage and modify that data, and how to build useful and effective programs around it.

## The Power of Go: Tools

Are you ready to unlock the power of Go, master obviousness-oriented programming, and learn the secrets of Zen mountaineering? Then you're ready for The Power of Go: Tools.

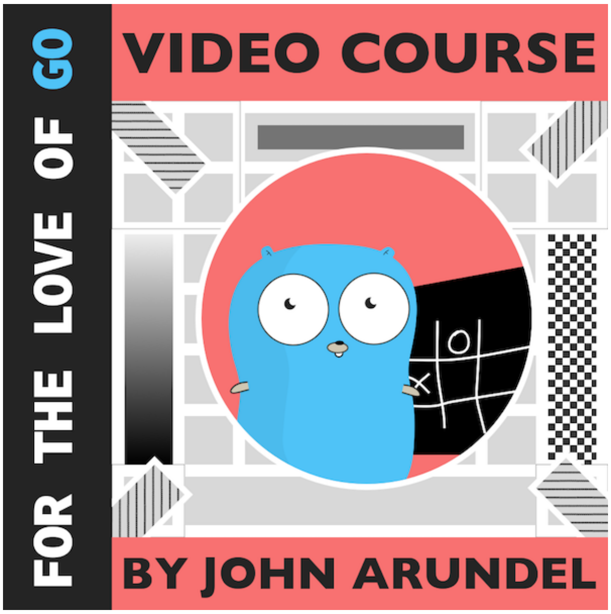


It's the next step on your software engineering journey, explaining how to write simple, powerful, idiomatic, and even beautiful programs in Go.

This friendly, supportive, yet challenging book will show you how master software engineers think, and guide you through the process of designing production-ready command-line tools in Go step by step.

## Video course

If you're one of the many people who enjoys learning from videos, as well as from books, you may like the video course that accompanies the 'For the Love of Go' book:



- For the Love of Go: Video Course

## Further reading

You can find more of my books on Go here:

- Go books by John Arundel

You can find more Go tutorials and exercises here:

- Go tutorials from Bitfield

I have a YouTube channel where I post occasional videos on Go, and there are also some curated playlists of what I judge to be the very best Go talks and tutorials available, here:

- Bitfield Consulting on YouTube

## Credits

Gopher images by the magnificent egonelbre and MariaLetta.

# Acknowledgements

My thanks are due to my students at the Bitfield Institute of Technology who unstintingly gave of their time to help review the drafts of this book, discuss its ideas, provide feedback, and suggest questions and examples. I'm especially grateful to Brian Hasden, who came up with the "cow and chicken" example.

Many thanks also to the hundreds of beta readers who willingly read early drafts of various editions of this book, and took the time to give me detailed and useful feedback on where it could be improved. Shiv Patil, Andrew Smith, BK Lau, Dan Macklin, Pedro Sandoval, and Pavel Anni in particular helped me a good deal.

If you'd like to be one of my beta readers in future, please go to my website, enter your email address, and tick the appropriate box to join my mailing list:

- https://bitfieldconsulting.com/