
# Go Faster

Ollie Phillips

# Go Faster

## Join the thriving community of skilled Go developers!

Ollie Phillips

This book is for sale at http://leanpub.com/gofaster

This version was published on 2023-04-18

# Tweet This Book!

Please help Ollie Phillips by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Go Faster is helping me to learn Golang!

The suggested hashtag for this book is #GoFaster.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#GoFaster

# Contents

# Preface

*When I started learning Go I was productive with the language fairly quickly. After only a short time I could build fast, safe and stable applications which could run anywhere. Before long I was choosing Go in preference to PHP and Node JS - the other languages I'd been developing with - whenever possible.*

*I continued my journey with Go and learned more as I went, but often, I'd find myself in a bind - Golang wouldn't do what I wanted it to. I'd find workarounds sure, but they often seemed clunky. As I tackled more advanced problems this happened more frequently and at times it seemed like I couldn't achieve what I wanted with Go at all!*

*After facing many of these challenges, I realised the actual problem wasn't with Go, but with my knowledge. For one thing, I found I sometimes didn't understand how Go wanted me to solve a problem - in the idiomatic 'Go' way. Other times, I discovered I had missed some of the more nuanced detail about a topic, which on the surface had appeared straightforward.*

*In a nutshell, I had quickly acquired a core of knowledge about Go - enough to write programs - but I had limited understanding of what was happening under the hood when the compiler took my code off me and ran it.*

*So I decided to try cementing my understanding by teaching Go to others via a weekly Blog and on-site training sessions under the GolangAtSpeed[1] brand. Both aimed to discover and expose what I called "the longtail of Go": the less obvious parts, the quirky idioms and the gotchas, the things that may catch other developers out too.*

*Covid-19 brought a premature end to the on-site training and the blog mostly fizzled out due to me jumping back into contract Go development at a time of uncertainty. But, I promised myself (and some blog readers) that I'd write a book on the same theme and, almost three years later, here it is!*

*Before you jump into the book itself, I want to thank you for purchasing it. I sincerely hope you enjoy it and get much value from it. I also hope you enjoy your journey with Go as much as I have my own, and that my book does indeed help you get there faster!*

*Finally, my learning journey is not yet complete, I'm simply sharing what I know in Go Faster because I think it may help others. So, if you have any feedback or spot any mistakes, please do reach out to me. In doing so you'll help me improve the book for others who follow you!*

*Best wishes,*

*Ollie*

[1] https://golangatspeed.com

# About this book

*Go Faster* is a book for those wanting to learn the Golang programming language. Though many great books have been published on this subject with the same aim, *Go Faster* takes a slightly different approach.

For starters, it recognises that Go is a relatively simple language with just 25 keywords and that most developers do not struggle to learn its limited syntax.

It acknowledges that most readers will be able to set up a Golang development environment using one of the operating system-specific binaries, without needing detailed instruction.

It recognises that most readers will have a basic level of programming knowledge already - possibly even prior experience with Go - but that that knowledge may differ, depending on their route into programming and the languages they have used to-date.

The examples in *Go Faster* are simple by design. There are no real world or useful applications, instead, all the non-essential code is stripped away to help the user see the principles being demonstrated and grasp the concepts.

*Go Faster* is designed to assist developers coming from object-oriented programming. It elaborates in areas which can confuse since the Go programming paradigm is different than that of OOP. And like Go itself, *Go Faster* is opinionated with its semantics.

The book also covers features which are relatively new to the language. Features that many developers, even experienced ones, may not yet be familiar with, and many existing texts may not yet explore, such as Generics and Workspaces.

Finally, *Go Faster* reflects the author's own journey learning Go and is laid out accordingly. Not as an A-to-Z style reference, but in a way that should enable understanding earlier, laying solid building blocks on which the reader can progress.

If the author was learning Go again, this is the way they would structure their learning path, and hopefully, it is this structure which helps users at all levels gain a solid understanding of Go, quickly.

# About the Author

Ollie Phillips is a Software Developer based in the United Kingdom.

Over the previous 8 years, he has worked with Go in various domains including Finance, Health, Learning Management and Cloud Automation. Most recently he was building tooling in Go for the *Morpheus Cloud Management Platform*.

He is also the author of the GolangAtSpeed[2] substack blog and is passionate about Go and helping others get productive with it.

[2] https://golangatspeed.com/

# Before you begin

Some pointers to help you get the most from *Go Faster*

## Go Playground

Throughout *Go Faster* we use example code which in most cases is also provided as executable snippets on the Go Playground. You can also use the playground to test your code if you don't have a local development environment. You can find it here:

https://go.dev/play/

All the examples, listed by chapter can also be found on Github here:

https://github.com/golangatspeed/GoFasterExamples

## Installing Go

If you do want to set up your local environment, you'll need to download and install a recent version of Go.

At the time of writing the latest version is 1.19. You can get a distribution for your operating system here:

https://go.dev/dl/

## Editing code locally

You'll have more fun writing Go in a text editor which offers Go-specific helper functions such as code completion, formatting and automatic imports.

Either *Visual Studio Code* - with the appropriate plugins - or *Goland* which has full Go support out-of-the-box, is recommended.

There are links to download pages for both applications below.

- Visual Studio Code - https://code.visualstudio.com/download
- Goland - https://www.jetbrains.com/go/

## A word on the code examples

*Go Faster* was written on *macOS*, so many of the command line examples will contain Mac/Linux-specific syntax. For example, directory paths are Mac/Linux style and not Windows style.

# Chapter 1 - Introduction to Go

In this chapter, we'll cover some essentials. First, we'll lay out a quick introduction to Go explaining what makes it attractive as a programming language, then we'll address some broader Go concepts and terminology that we will make use of as we progress.

## 1.1 Why Go?

Go, or Golang is a programming language developed by a team at Google. Similar to C (or Clang), it draws inspiration from many other programming languages also.

Go is a high-performance language which runs directly against physical machine resources and not on a virtual machine, unlike Java for example. Further, asynchronous programming via concurrency and goroutines can fully utilise the multiple cores of modern CPUs better than many other multi-threaded languages.

Go is a compiled language. Your program code is built into an OS-specific executable and not interpreted at runtime. It is statically typed, so types are known - and fixed - at compile time. Memory management in Go is for the most part automatic. Memory allocation is done for us as required, and memory is deallocated when no longer in use via a *garbage collection* process. Together these traits help to eliminate many common programming errors.

Go is opinionated on many things, one of which is formatting. This opinion, together with its simple syntax, results in very readable and clear code. Go has a complete suite of built-in tooling which includes utilities for testing, building, profiling and more. It is the readable code and powerful tooling which attract many developers to the language.

In version 1.11 Go modules were introduced for dependency management and are now the defacto approach.

In version 1.18 the hotly debated Generics feature was introduced together with Go workspaces.

## 1.2 Language semantics

Go is not *object-oriented* although it feels similar at times.

Possibly because of that similarity, or more likely familiarity with object-oriented programming (OOP) in general, developers often find themselves conversing about Go using *object-oriented* semantics which are often not appropriate to Go.

You may see and hear terms like *instance*, *object* and *method* - even *inheritance* - being used in the context of Go code, and while it sort of works, it is mostly incorrect.

Better we establish the right Go terminology early on and use it. Especially as the correct semantics can help us understand what Go is doing differently from OOP-based languages.

For example, while OOP has *inheritance*, Go has *composition* and *embedding*. There's an overlap between these approaches but they are not the same thing.

Go does not have *objects* or *instances*, neither does it have *methods* in the OOP sense. Instead, we have *receivers* (or *receiver functions*), so named because they *receive* the value (or the address of the value) of the type on which they are bound. While conceptually similar to *methods*, Go *receivers* can be bound to any user-defined type, they are not constrained to a *class* hierarchy.

That said, *receivers* and *methods* are terms that are used interchangeably in Go. I generally use the term *receivers* and will do it exclusively throughout *Go Faster*, purely to break the link with OOP and class-based property access.

Like OOP, Go has *interfaces* - a very powerful aspect of the language when used appropriately - and of course, we have *functions*.

In Go, everything is a value. Even *pointers* are just a value, that value being the *address* of another value in memory. There's no magic, but this concept is core to understanding how to work with values and pointers and avoid some common mistakes.

Errors are just values too, Go does not have exceptions. Errors must be checked and handled in your code, which despite being a little verbose and repetitive, contributes greatly to readability and comprehension.

So, Go has some specific terminology such as *types*, *receivers* and *pointers* and we should use these. For anything else, it is more correct to refer to it as a *value* of something and not an *instance* of something - or indeed an *object*.

There's quite a lot to unpack there, but don't worry we will cover all of the above in more detail as we progress.

## 1.3 Visibility

Unlike *object-oriented programming*, Go does not have the concept of *public* and *private*. Instead, it uses the notion of *exported* and *unexported* visibility modifiers.

Any *variable*, *constant*, *function*, *receiver*, *type* or indeed *struct field*, declared with a lowercase first letter in its name, is considered *unexported* and cannot be directly accessed outside of its package namespace.

The same, when capitalised, is considered to be *exported* and fully visible to the code that imports the package.

The example code below illustrates the principle when applied to types, variables, functions and struct fields. Note the exported struct field in the unexported struct type at line 9 which is a bit odd: an exported field on an unexported struct would appear to be completely redundant.

**Example 1 - Visibility modifiers**

```
package exporting

type ExportedStruct struct {
        unexportedField string
        ExportedField   string
}

type unexportedStruct struct {
        ExportedField string // how is this useful?
}

var ExportedVariable string
var unexportedVariable string

func ExportedFunc() { }

func unexportedFunc() { }
```

Well, it is redundant when it comes to exporting that field outside of the package because the struct itself is unexported, but you may often see code written like this, simply because there is a need to perform some type of conversion of that struct field into an alternative data format, such as JSON.

We call this conversion process *marshalling* and Go will only *marshal* exported struct fields when creating the output. Unexported struct fields are ignored.

Why does *unexported* not mean *private*? Because unexported values can still be made available outside of the package. For example, there is nothing which prevents an exported function or receiver from returning an unexported value from the package namespace to the caller. While code like this can be a source of confusion, so it is not recommended practice, it's perfectly possible to do this.

So when building packages for others to use, we should make decisions about *exporting* and *unexporting*, not on a need for concealment, but instead as a way of communicating which parts of our package should comprise the API that developers use, and which parts should not.

## 1.4 Comments and documentation

The single-line and multi-line comment styles of Go are probably already familiar and are used in many languages inspired by C. See the example below.

**Example 2 - Comment styles**

```
package main

func main() {
        // This is a single-line comment

        /*
                This is a multi-line comment
                which spans more than one
                line.
        */
}
```

For comments to be useful, they should be concise and precise and must be updated in line with the code. Stale comments which are not maintained with the codebase, detract from the code and may even confuse matters.

In Go, comments are especially useful since they are used to generate documentation. Developers write comments inline which then automatically form the basis of their software's instruction manual.

The documentation itself is built from the code comments using a command line tool called *Godoc*. We'll cover installing and using the tool shortly, but first, let's outline some *standards* to help you get the most from the *Godoc* tool.

## 1.4.1 Comment standards

1. All exported functionality of a package is expected to have a comment. The comment should start with the *function*, *variable* or *type* name for which it is written, and should succinctly explain its purpose. Only comments in this format on exported functionality will be used to build documentation.
2. The package itself should include a comment above the package name, which provides an overview of what the package does. This is used in the Overview section of the built documentation.
3. Links can be included in comments via an absolute URL which includes the schema e.g. HTTPS. Relative URLs are ignored. The *Godoc* tool will parse and include links which meet the requirements in the documentation.
4. Comments separated by a commented blank line will be interpreted as paragraphs and output as such in the package documentation.

The example below shows a package with these requirements satisfied. The code is unimportant for the moment.

**Example 3 - Making the most of comments in documentation**

```
// Package user implements functionality for working with users.
//
// This content will be in a new paragraph.
package user

// User is a representation of a single user
type User struct {
        Name string
}

// NewUser is a factory that allows us to configure the properties
// of a new User.
//
// See https://someurl.com for more information.
func NewUser(name string) *User {
        return &User{
                name: name,
        }
}
```

## 1.4.2 Installing and using Godoc

*Godoc* is an external package which can be downloaded and installed using the Go tool. Assuming you have installed Go locally, the following command will fetch the latest version of the package and install it in your workspace.

```
go install golang.org/x/tools/cmd/godoc@latest
```

Once installed, the tool can display documentation for the Go standard library - useful when offline - from any terminal window with this command.

```
godoc
```

The command starts a document server on http://localhost:6060. You can specify a different port if 6060 is in use.

```
godoc -http=:9090
```

For more information on using *Godoc* run the command with the help flag.

```
godoc -h
```

You can also preview how your code comments will appear as documentation using *Godoc*. For any package with *Go Modules* support, simply navigate to the folder which contains `go.mod` in a terminal window and run *Godoc*.

The documentation for the package will be created and listed under the "Third Party" section in the documentation server's index.

## 1.4.3 Including example code

We can include code examples as part of the generated documentation. These appear as text "code" and "output" sections in the documentation. However, if you run *Godoc* with the `-play` option, any such examples become interactive.

When interactive, not only can you run code snippets on the Go Playground using the additional *Play* button which appears top right, but all examples included in the documentation become executable - and editable - programs and not just static text. This is a really useful feature when learning about the standard library.

*Godoc* examples are actually a type of test, and they are run and verified like other tests. We cover testing later in the book, but just like other tests, examples must reside in files named with the `_test.go` suffix.

Within those files, examples are differentiated from normal tests as the function names use the `Example` and not the `Test` prefix as shown in the following example taken from the standard library *stringutil* package.

**Example 4 - An example function**

```
package stringutil_test

import (
    "fmt"

    "golang.org/x/example/stringutil"
)

func ExampleReverse() {
    fmt.Println(stringutil.Reverse("hello"))
    // Output: olleh
}
```

Examples do not contain assertions normally associated with unit tests which determine pass or fail. Instead, they contain a commented `Output` line.

The Go test tool checks the actual output of an example function against the output which is on this line. If it matches it passes, if not it fails.

This is a useful and overlooked feature when you need to make assertions on functionality which outputs to Stdout. You can use an *Example* to check it, there's no need to pipe or redirect the output to a file or variable to compare it.

The online Go package repository[3] hosts documentation in interactive mode, so you can also experiment with example code there.

[3] https://pkg.go.dev

# Chapter 2 - The Go command line interface (CLI)

When you download Go you get the Go standard library packages and the Go CLI which is a tool for managing Go source code. You'll use it for much of what you do with Go, with one or two exceptions, for example, *Godoc* as we've seen.

Exploring all the functionality of the *Go* CLI is left as an exercise for the reader, this section aims to show you how to get started with the tool and highlight specific commands you'll use most often.

You can list all the top-level options of the Go tool using this command.

```
go help
```

If you want to get more help for a specific option use the same command followed by the option. For example, to learn more about the `version` option run the following command.

```
go help version
```

## 2.1 Version information

Go has a backwards compatibility guarantee which means that code written in Go version 1.0 can still be compiled and run on the latest Go 1.x version. This doesn't work the other way, so code written in Go 1.19 may not compile on earlier versions of Go, so you'll sometimes need to check the version you're using.

To get the current installed Go version run this.

```
go version
```

The output will be similar to below, showing the version, OS, and architecture.

```
go version go1.19 darwin/amd64
```

## 2.2 Environment information

You can inspect the configuration for your installed Go environment using this command.

```
go env
```

This outputs all the configuration variables for your installation and can be useful if you are having issues with the Go tool.

The variables worth highlighting here are:

- GOPATH - the location of your Go src code including the standard library
- GOMODCACHE - the location of modules which have been downloaded and cached
- GOPROXY - the proxy from which new packages are downloaded

For information on other environment variables consult this page[4].

## 2.3 Module and workspace management

In a later chapter, we'll cover dependency management in Go and discuss the roles of modules and workspaces. For now, you can familiarise yourself with the module and workspace options via the `help` subcommand.

```
go mod help
go work help
```

## 2.4 Format your code

If you're using an IDE like Visual Studio Code with the Go plugin installed, or Goland, then it's likely your code is formatted automatically upon saving. If you don't have this facility the `fmt` subcommand will format your Go source files according to Go standards. Running the tool against a Go source file is straightforward.

```
go fmt main.go
```

## 2.5 Testing

We'll get to testing later in the book, for now, it's enough to know that testing is initiated from the Go tool also. For example to run all the tests in your project use this command from within the project folder.

[4] https://pkg.go.dev/cmd/go#hdr-Environment_variables

```
go test ./...
```

## 2.6 Cleanup

*Clean* removes object files from package source directories. It's useful if you build an executable and it is placed in the project workspace along with your source. It can be removed, so that it is not committed to version control, with the following command.

```
go clean
```

## 2.7 Downloading packages

We can use the `get` command to download package dependencies and install them. We cover this in a later chapter but here's an example which will add a dependency for a package, or if it is already available, it will upgrade the package to the latest version.

```
go get example.com/pkg
```

## 2.8 Running a program

Assume you've written a program and all the code is saved in file `main.go`. To run the code navigate to the folder which contains `main.go` and execute this command.

```
go run main.go
```

If package `main` was made up of multiple Go source files, as is often done to help organise the source code, we could pass all the files to the `run` command like this.

```
go run *.go
```

The wildcard filename may not work on Windows systems, and you may even run into problems on Mac/Linux when external files for configuration or data are used by your program, in that the paths are not resolvable. So, it's recommended to build the program and run the resulting executable if you encounter any problems.

## 2.9 Building your program

To build an executable for your current operating system, simply use this command.

```
go build -o helloWorld ./...
```

The `-o` flag is used to provide a specific name for the built executable.

By default, if no name is specified and the main package is at the root of your project folder the binary created will be named using the *module* name and the executable will be placed at the root of your project folder.

Later we'll talk about project organisation and how the `cmd` folder is helpful.

If you use the `cmd` folder in your project executables will be built using their sub-folder names unless names are specified with the `-o` flag.

If we take an example project which contains the source for two executables, *api* and *cli*, the below commands would be needed to build them.

```
go build ./cmd/api
go build ./cmd/cli
```

The built executable files are again placed at the root of your project folder. It can be run from the root folder like this.

```
./api
```

## 2.10 Building for other operating systems and architectures

As we've seen, it's simple to build a program for your operating system using `go build` but one of the best features of Go is that we can just as easily compile our code to run on other systems too.

It's very straightforward to cross-compile source code to alternative targets. We use the same *build* command as before but additionally specify both the *OS* and *Architecture* we want to build for using the `GOOS` and `GOARCH` environment variables.

By default, these variables are set to our local OS and architecture so we're overriding them to build for a different target.

You can inspect all of the target platforms your Go version can compile for using this command.

```
go tool dist list
```

In the console output, the text before the forward slash is the operating system, and the text afterwards is the architecture.

For example, on *macOS*, to build the *helloWorld* program as we did in section 2.9 but for Linux OS and a 64-bit AMD architecture we'd do this.

```
env GOOS=linux GOARCH=amd64 go build -o helloWorld ./...
```

To compile for Windows and a 64-bit ARM architecture we'd use this command instead.

```
env GOOS=windows GOARCH=arm64 go build -o helloWorld ./...
```

# Chapter 3 - Structure of a Go program

In this section, we're going to walk through the structure of a Go program using a typical *Hello World* program.

The program itself only prints out a couple of strings - but it does more than you might first expect - run it now on the Go Playground to see for yourself.

**Example 5 - Hello World in Go**

```
1  package main
2
3  import (
4          "fmt"
5          _ "github.com/golangatspeed/pkg/sideeffect"
6  )
7
8  func init() {
9          fmt.Println("Hello World!")
10 }
11
12 func main() {
13         fmt.Println("Hello World again!")
14 }
15
16 // Go Playground: https://go.dev/play/p/HbshOM2vDed
```

There's probably a bit more syntax in the above example than you might expect in a simple Hello World program, but the extra pieces will be explained and should be useful.

## 3.1 Packages and importing

All Go code resides in a package. An application's package is always `main`.

The `import` keyword is used to include other packages, from either the Go standard library as is the case with `fmt` or from external packages.

It's common to see external packages following a semantic naming convention which represents their location on the web e.g. GitHub repository URL.

Rather than use the `import` keyword for each package we can use brackets around all packages, as we have done in the example.

The underscore which prefixes the imported external package is known as the *blank identifier*. This is how we tell the compiler that we don't need to refer to the package contents in our code but we do need to run the `init()` function (see Section 3.2).

We call this importing a package for its *side-effects*. Without the blank identifier, the compiler would complain about the unused import.

The blank identifier has other uses which we'll cover later.

### 3.1.1 Aliasing

Packages can be prefixed with an alias which is useful when we experience naming collisions between similarly named packages, or wish to work with the package contents using a more semantic name than the package itself would otherwise allow.

Aliasing should be used judiciously to add clarity, and not as in the mischievous example below!

**Example 6 - How not to use aliasing**

```
1  package main
2
3  import (
4          log "fmt"
5          fmt "log"
6  )
7
8  func main() {
9          fmt.Println("This looks like log.Println output?")
10         log.Println("This looks like fmt.Println output?")
11 }
12
13 // Go Playground: https://go.dev/play/p/zuXJ0If5OFn
```

### 3.1.2 The 'dot' import prefix

There is also a *dot* prefix - a period - which can be used to promote the functionality of an imported package to the current package namespace, allowing the package prefix to be omitted when making calls to its contents.

This is rarely seen in practice as it mainly detracts from clarity. To illustrate, in the code example below, the `Println()` function could also be from the `fmt` package as well as the `log` package. Only by inspecting the imported packages are we able to determine which it is.

**Example 7 - The dot import prefix**

```
package main

import (
        . "log"
)

func main() {
        Println("This looks like log.Println output?")
}

// Go Playground: https://go.dev/play/p/SEUbOnOb6wH
```

## 3.2 Main and Init functions

The `main()` function is considered to be the entry point of the application - the top of the call stack. There is only one per executable. It takes no arguments and expects no return values.

Every package including `main` can have an `init()` function. Every included package with an `init()` function will have that function executed *once* when the application starts.

`init()` functions execute before `main()` but after package level variables are evaluated. Their use should be constrained to the initial setup, for example reading environment variables or a config file.

If you ran the *Hello World* program on the Go Playground you will have noticed the additional line (below) was printed before any of our program output?

```
This is the side effect?
```

This is the output of the `init()` function in the package `github.com/golangatspeed/pkg/sideeffect`, executing before our `init()` function which itself executes before our `main()` function.

## 3.3 Developing a package

If you are developing a package for use by other applications and not an executable, you will not have a `main()` function and your package name will be something other than `main`.

Below is a very simple package example. It's the content of the package we imported from Github in the *Hello World* example.

**Example 8 - Simple package**

```
package sideeffect

import "fmt"

func init() {
        fmt.Println("This is the side effect?")
}
```

A package may contain a combination of exported and unexported functionality as we discussed in Chapter 1, although this example package exports nothing.

# Chapter 4 - Project organisation

Project organisation relates to how you choose to structure your Go code: both its location on disk and within the project folder.

We stated earlier that since Go 1.11, modules have allowed us to manage our projects outside of the `$GOPATH/src` folder - a big benefit, especially where a project contains more than just Go source code, as many projects organised as mono-repositories frequently do.

When it comes to structuring our code within the project code folder itself, Go is mostly unopinionated. There is one special case to mention - the optional `internal` folder - but beyond this, it's up to us.

That said, there are some best practices which we might choose to adopt, which we'll also cover in this section.

## 4.1 The internal folder

The `internal` folder gets special treatment from the Go tool which uses it to limit access to packages contained within it. It effectively makes code invisible to another package unless it shares a common ancestor.

Most often this is used to prevent external modules from accessing code which a developer does not want to share or maintain a public API for.

Similar to using the unexported visibility modifier in the code itself, we should incorporate `internal` into our project organisation to hide entire packages which are for *internal* use only.

## 4.2 The cmd folder

We've said package `main` is the entry point for your program already, but it also signals there is an executable which can be built by the Go tool.

For simple programs `main` can be located at the root of the project folder structure. The advantage is that users who run `go get` on the module will get the application built and installed automatically.

However, with this approach, there can only be one executable in your project and this often presents a problem. What if you have multiple applications which share much of the same code?

For example, you have a HTTP rest-based API and also CLI which both use the same database access routines and wrapper logic. In this scenario, you'd need two projects. The shared code would

have to be visible to both projects as external modules or duplicated in both projects and therefore maintained in two places.

This is not ideal, but fortunately, there's a simple and widely used solution to the problem. We simply move the executables (anything with package `main` into their own subfolder within a `cmd` folder.

There is no special significance in Go to the `cmd` folder name it could be anything, but short for *command* it has become a common practice to use `cmd`.

Using this approach we can manage multiple executables in one project and we can provide access to the common libraries they use without needing to put them in a third project. Crucially, we are managing all the related code in one project without any need for duplication.

For the API and CLI example above we might have this folder structure.

```
cmd/
    api/
        main.go
    cli/
        main.go
database/
    *.go
logic/
    *.go
go.mod
go.sum
```

## 4.3 The pkg folder

So we've covered the benefits of `cmd` and `internal`, what might we gain from using a `pkg` folder in our project?

Again, the folder name has no special significance in Go, but this name and structure provide two benefits.

First, the name `pkg` is semantic and tells other developers what packages in our module are intended to be reusable outside of the module. Of course, this would be the case for any package not in the `internal` folder, which brings us nicely to the second benefit.

Most projects will have numerous other folders which reside at the root level inside the project folder. Folders for config, for build automation, maybe data folders or folders for assets such as images and scripts, to name just a few examples.

The point is that typically a Go project will have many other folders which will be nothing to do with your Go code. If we structure all Go packages at the same level as these folders it becomes difficult to determine what is a Go package and what is not. Developers coming to your project for the first time will need to establish this by interrogating each folder.

Instead, locating all the Go packages in the `pkg` folder solves this problem very simply. We can see where all the public Go packages can be found without needing to inspect all the folders at the root level.

## 4.4 Wrapping it up

Taking the example we've used throughout this section, using the above principles to organise our code, the structure of our project would look like this.

```
cmd/
    api/
        main.go
    cli/
        main.go
internal/
    hidden-package/
        *.go
pkg/
    database/
        *.go
    logic/
        *.go
go.mod
go.sum
```

None of this is set in stone, you may wish to organise your code differently, and not use the `internal` folder, but, if you adopt the practices outlined here, other developers will be able to identify where your Go code is located, if it contains executables, as well as understand which packages you intend for others to use, and which you do not.

# Chapter 5 - Dependency management

In this chapter, we're going to take a deep dive into dependency management in Go, which changed radically in Go 1.13 with the default adoption of modules and was refined further with the introduction of workspaces in Go 1.18.

After reading this section you'll be comfortable creating and working with package dependencies in Go as well as managing modules and workspaces.

## 5.1 Modules

Go modules were introduced as an experimental feature in Go version 1.11, and later became the default approach to dependency management in version 1.13.

Before the introduction of modules, as we've stated, package management was largely dependent on the `go get` command, and Go source code was managed in the `$GOPATH/src` folder.

Modules allow developers to organise their Go projects locally as they please. Projects no longer need to be located within the system `GOPATH` directory structure - an approach that was often inconvenient.

So what is a module?

A module comprises one or more Go packages managed by a `go.mod` file. This file defines the module's identifier, the minimum version of Go required, and any external modules - together with their versions - on which the module depends.

The module's identifier is also the import path for the module. Remote modules need to be resolvable by that path identifier.

Where external dependencies are a part of a project, an additional `go.sum` file sits alongside `go.mod`. This file contains hashes - or checksums - of dependencies and is used to fix versions and determine where dependencies have been modified. Unlike the `go.mod` file, `go.sum` should never be manually edited.

Below is an example of a very basic `go.mod` file for a module which has no dependencies outside of the standard library.

```
1  module github.com/golangatspeed/pkg/sideeffect
2
3  go 1.13
```

Modules simplify and standardise the approach to dependency management and vendoring for everyone. Prior to module support, several third-party tools existed to solve the problem of dependency management.

Module management is a part of the standard Go tooling, and changes to `go.mod` can be initiated from the command line. The file can also be manually edited and this is often just as convenient.

To see all the available *go mod* commands run the below in your terminal.

```
go mod help
```

## 5.1.1 Direct versus indirect dependencies

A dependency on a module can be either *direct* or *indirect*. A direct dependency is simply a dependency that the module needs itself.

An indirect dependency is usually a dependency requirement of one of the direct dependencies or a dependency listed in `go.mod` which is not currently used in any of the module source files.

Indirect dependencies in `go.mod` are suffixed with `// indirect`.

Below is a more complete example of a `go.mod` file containing both direct and indirect external dependencies.

```
module github.com/my-module

go 1.17

require (
        github.com/spoonboy-io/koan v0.1.0
        github.com/TwiN/go-color v1.1.0 // indirect
        golang.org/x/sys v0.0.0-20210630005230-0f9fa26af87c // indirect
)
```

## 5.1.2 Creating and updating a module

Module creation is straightforward via the `go mod init` command.

```
// create the module
go mod init *module-identifier*
```

Remote modules included in your code with the `import` keyword, and which are not already available on your system, can generally be downloaded and added to `go.mod` with the `go mod tidy` command.

```
// add dependencies and update go.mod
go mod tidy
```

Alternatively, use the traditional `go get` command from within your project workspace to fetch the module. This command is module aware so `go.mod` is automatically updated to include the new dependency.

```
go get *module-identifier*
```

By default, both `go mod tidy` and `go get` will fetch the latest version of the module, equivalent to this command with the `@latest` suffix.

```
// we don't need to explicitly add @latest, that is the default
go get *module-identifier*@latest
```

If you require a specific version of a module it is possible to obtain it by specifying the *tag*, *branch* or even *commit reference*.

Below are three examples which show how to download and use specific module versions by tag, branch and commit. In all cases `go.mod` is updated with the requested version and `go.sum` will be used to track changes to that version.

```
// get a specific tag (version)
go get *module-identifier*@v1.0.0

// get the development branch
go get *module-identifier*@my-dev-branch

// get a specific commit reference @07434ea
go get *module-identifier*@07434ea
```

## 5.1.3 The replace directive

The *replace* directive provides a simple mechanism to substitute a required module with another version of that module. This code can either be stored locally on your machine or at an alternative remote URL.

Imagine finding a bug in a third-party module, reporting it and then being reliant on the code owner to fix their code. Until it's fixed your project can't use that code or must accept the buggy code. Imagine another scenario, needing a new feature in a module, and asking the maintainer to add it. Until it is added, your project has to do without the feature.

In both above scenarios, the codebase can be forked and amended to fix the bug or add the feature. You can send a pull request to the respective maintainer(s) as a good citizen, but you then have to wait for it to be accepted before your project can use the updated code.

The *replace* directive presents a way around the above problem. We simply add a replace directive to `go.mod` which tells Go to use code in the forked repository in place of the original module.

When the original module is fixed or has the new feature added, the *replace* directive can simply be removed. There is no impact on your go codebase: no need to find and replace every import statement which contains that module.

Replacements can be made using the `go mod replace` command but it is very simple to include them in `go.mod` directly. The example `go.mod` file below shows two replacements being made.

The first *replace* directive replaces the module with a locally available version using the absolute path to the module's location on disk.

It's worth stating the obvious here - that local replacements will break builds for other users who don't have the module available at that location so they should probably be removed before the code is shared.

The second *replace* directive substitutes in a forked repository and is safer to share. It achieves the same end, but other users will be able to build the project because the forked version is accessible on the Internet.

```
1  module github.com/golangatspeed/replace-example
2
3  replace github.com/original/module => /Users/olliephillips/module-my-version
4  replace github.com/original/module-two => github.com/golangatspeed/module-two
5
6  require (
7          github.com/original/module v1.0.0
8          github.com/original/module-two v1.0.0
9  )
```

## 5.2 Workspaces

The *workspace* feature was added in Go version 1.18 and furthered the flexibility of modules and dependency management generally.

*Workspaces* allow replacements of modules to be made with modules found locally, **without** making changes to the `go.mod` file.

This avoids the problem of having to add and subsequently remove the replace directives, when working with local module versions, before committing code to version control.

A workspace is defined by a single `go.work` file and is best used for local development, so excluded from version control.

To create a workspace in your project use the commands below.

```
// create a workspace file
go work init

// or create the file and add a local path to the workspace
go work init *path/to/module/or/modules*
```

A `go.work` file created with the second of the two commands would have the following contents.

```
go 1.18

use *path/to/module/or/modules*
```

Modules which are located within that path will be used in preference to any online versions or locally cached versions.

We can also use the *replace* directive to replace specific modules and versions in the `go.work` file instead of `go.mod`.

In summary, workspaces help developers isolate changes that should only apply in their development environment, so mitigating the risk of committing changes which would detrimentally impact other developers and their environments.

## 5.3 Vendoring

Vendoring includes a local copy of external module dependencies inside the project itself in a `vendor` folder. Go then uses the vendored modules in builds and testing instead of their external equivalents.

Vendoring is useful in many situations. Where a project needs to be built on a machine with no internet access for example; or where strict dependency management has been adopted and submitting vendored modules to version control assists in that end.

Vendoring can also ensure that modules remain available to a project, even if the module is removed from the Internet. This risk is mitigated by the Go module proxy to an extent, but it can still happen - and a project which cannot locate a dependency cannot be built.

Finally, where builds are automated in some form of continuous integration process, build times can be reduced if modules are vendored locally and don't need to be downloaded from the Internet for each build.

Vendoring is very straightforward, simply run the below command from within your project.

```
go mod vendor
```

A `vendor` folder will be created with local copies of all external module dependencies. There is also a `modules.txt` file created which lists vendored modules and their versions and this is used as a manifest by the Go tool.

# Chapter 6 - Variables and constants

Let's discuss variables, constants and scope next. A word of warning, things may feel a little 'chicken-and-egg' for a short time, as we'll have to briefly mention *type* which we haven't covered so far, and won't until the next chapter.

## 6.1 Variables

We've said that Go is opinionated. Usually, there is a specific way to do something - maybe a couple of ways - but one area where being opinionated would have been nice is variable declaration and initialisation. There are too many ways to do it.

*Declaration* is the act of creating a variable with a name and type, *type* being what the variable can hold, such as a number or a string. *Initialisation* is the act of setting a variable equal to some value at the time it is declared. *Assignment* is the act of changing the value stored in a variable by assigning a new value at any point during program execution.

We'll look at the different styles shortly so that you know how to recognise them but, generally speaking, for clarity you should limit the styles you employ in your code to just three, two of which use the `var` keyword, and the third a short-form declaration and initialisation using `:=`.

Here they are:

**Example 9 - Recommended variable declaration styles**

```
1  // Style 1. Declaration only in the global or function scope
2  var myVar int
3
4  // Style 2. Short form declaration and initialisation within the function scope only
5  myVar := 10
6
7  // Style 3. Multiple variables, omitting the repeating var keyword
8  var (
9      myVar int
10     myVar2 int
11 )
```

We'll cover scope soon, but to summarise, you may use all three styles inside functions, but you cannot use `Style 2` in the global scope.

Note that when we refer to function scope, we mean anything, *not* in the global scope, so variables declared in receivers would also have function scope.

You should probably move to the next section on constants in all honesty, but for completeness, the example program below shows all of the different styles which you may see, and indeed, can use.

Some are declarations, others also initialise the variable. Some infer the type, while others explicitly state the type, but they're all valid ways to create variables.

**Example 10 - All variable declaration styles**

```
1  package main
2
3  import (
4          "fmt"
5  )
6
7  var myVar1 int
8
9  var (
10         myVar2 int = 1
11         myVar3 int
12 )
13
14 var myVar4 int = 4
15 var myVar5 = 5
16
17 func main() {
18         var myVar6 int
19     var myVar7, myVar8 int
20         var myVar9, myVar10 = 9, 10
21
22         myVar11 := 11
23         myVar12, myVar13 := 12, 13
24
25         fmt.Println("Go is not so opinionated on some things...")
26         fmt.Println(myVar1, myVar2, myVar3, myVar4, myVar5)
27         fmt.Println(myVar6, myVar7, myVar8, myVar9, myVar10, myVar11, myVar12, myVar13)
28 }
29
30 // Go Playground: https://go.dev/play/p/c8eFFqUfVTz
```

Let's break that example down.

A variable can be created using the `var` keyword in the global scope or function scope e.g. myVar1 and myVar6.

If we declare the variable and type it will be initialised to its *nil* value e.g. myVar1.

We can initialise it ourselves and specify the type if we wish e.g. myVar4, or let the compiler infer it from the initialised value e.g. myVar5.

As with the `import` keyword, we can avoid repeating the `var` keyword by enclosing multiple declarations within braces. You can do this in the global or function scope.

Within the function scope - as well as the styles above - we can also use the short-form notation `:=` which declares and initialises a variable. Type is inferred based on the value e.g. myVar11.

Finally, we can declare multiple variables **of the same type** using standard or short form notation as a comma delimited list e.g. myVar7 and myVar8, myVar9 and myVar10 and myVar12 and myVar13.

A bit confusing yes?

Remember, Go is compiled and not interpreted. We don't need clever optimisations in our syntax because the compiler will optimise our source code during a build. Our focus should be on writing clear code that the average Go developer can understand and limiting the styles we use to declare variables is a step towards that objective.

To wrap up variables we should mention *unused* variables.

The Go compiler will complain if a variable within a function scope is not used by your program at build time.

It's nice that the compiler has our backs - leaving unused code hanging about could be a source of confusion, but, it's worth stating that the compiler will not complain about any unused global variables simply because they cannot be detected during compilation.

However, it's common to have temporary placeholder code and if you're not ready to use a variable in your program it can be frustrating to keep commenting it out to run your program.

The solution is to use the variable in some way. A common approach is to print out the variable, but while this satisfies the compiler it can add noise when attempting to debug a program.

An alternative approach, which also keeps the compiler happy is to discard the unused variable using the *blank identifier*.

**Example 11 - Discarding with the blank identifier**

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          myVar := 1
7          _ = myVar
8
9          fmt.Println("compiler is happy")
10 }
11
12 // Go Playground: https://go.dev/play/p/NYjfpotoOtx
```

The example program above declares and initialises a variable then immediately discards it. The program outputs "compiler is happy" as expected, but if you comment out line 7 the compiler will not build the program and will output the following error:

```
1  ./prog.go:6:2: myVar declared but not used
2
3  Go build failed.
```

Temporarily discarding the variable is my preferred approach, because it is clear what the intent is, we're saying we don't need it *at the moment*.

Whether we choose to comment out, print or discard the variable, all should be considered temporary measures for use during early development only.

## 6.2 Constants

Constants are immutable values. They are declared with the `const` keyword and may be created in the same way as variables, with one exception, the short-form declaration `:=` is not valid for use with constants.

Just like variables, a constant's type can be inferred if it not explicitly set. However, unlike variables this inference may be a loose typing. The type only becomes fixed at the point of conversion, or point of use.

Note, this fluidity applies only to numerical types.

Check out the example below which demonstrates this fluidity, then uncomment line 18 and run it again.

**Example 12 - Loose typing of number constants**

```
package main

import (
        "fmt"
)

const myConst = 1
var myVar = 1

func needsAFloat(val float64) {
        fmt.Println("looks like we got a float:", val)
}

func main() {
        fmt.Printf("myConst is type '%T'\n", myConst)
        fmt.Printf("myVar is type '%T'\n", myVar)
        needsAFloat(myConst)
        //needsAFloat(myVar)
}

// Go Playground: https://go.dev/play/p/MC8HsTfnVz9
```

Output:

```
myConst is type 'int'
myVar is type 'int'
looks like we got a float: 1

Program exited.
```

## 6.2.1 Constants and the iota identifier

Sometimes the value stored by constant has meaning, but often that won't be the case, instead we simply want to differentiate between constant values, in fact, any value would do.

Typically, where the value doesn't have meaning and we simply want to tell between the constants we'll use integer values.

We can, if we wish, manage these values manually by maintaining the list ourselves, as in the example below.

**Example 13 - Managing number constants manually**

```
package main

import (
        "fmt"
)

const (
        const1 = 1
        const2 = 2
        const3 = 3
        const4 = 4
        const5 = 5
)

func main() {
        fmt.Println("constants values are:", const1, const2, const3, const4, const5)
}

// Go Playground: https://go.dev/play/p/EVMSF3BCCn8
```

Output:

```
constants values are: 1 2 3 4 5

Program exited.
```

Alternatively, we can use the *iota* identifier as a shorthand way of initialising all the constants.

**Example 14 - Managing number constants with iota**

```
package main

import (
        "fmt"
)

const (
        const1 = iota
        const2
        const3
        const4
        const5
```

```
)

func main() {
        fmt.Println("constants values are:", const1, const2, const3, const4, const5)
}

// Go Playground: https://go.dev/play/p/h4fNoptnCJZ
```

Output:

```
constants values are: 0 1 2 3 4

Program exited.
```

Note *iota* is zero-based, if you wanted to start the sequence at 1 as in the previous example you would simply add 1 as shown below.

**Example 15 - Rebasing the numbering from 1**

```
const (
        const1 = iota + 1
)
```

To use a non-linear sequence, we can employ the *blank identifier* once again to discard the specific iota values.

**Example 16 - Non-linear constant sequences**

```
package main

import (
        "fmt"
)

const (
        const1 = iota
        _
        _
        const4
        const5
)

func main() {
        fmt.Println("constants values are:", const1, const4, const5)
```

```
}

// Go Playground: https://go.dev/play/p/8QypEyOoxT2
```

Output:

```
constants values are: 0 3 4

Program exited.
```

Using *iota* can make constant management much easier, but can be prone to occasional problems if new constants are not added to the end of the list, but instead inserted into the list - say alphabetically - and the value itself has some meaning in your program. For example, the code below is fragile because we use the values, not the constants themselves in switch case comparisons.

**Example 17 - Values used in place of named constants**

```
package main

import (
        "fmt"
)

const (
        blue = iota
        //red
        yellow
        purple
)

func main() {
        chosenColor := yellow
        switch chosenColor {
        case 0:
                fmt.Println("Chose blue")
        case 1:
                fmt.Println("Chose yellow")
        case 2:
                fmt.Println("Chose purple")
        }
}

// Go Playground: https://go.dev/play/p/n-POyR0-k7a
```

Output:

```
Chose yellow

Program exited.
```

Uncomment line 9 and and now the program thinks we chose a different colour.

Output:

```
Chose purple

Program exited.
```

If you use *iota* to simplify constant management, be sure to use constant names throughout your code and never perform comparisons on actual values, which could change as new constants are introduced into your codebase.

## 6.3 Scope

Scope is an important concept, especially for variables, so let's review what we mean by *scope*.

A variable declared in the global scope may be accessed and mutated by any function in the same package. Equally, an exported package variable declared in the global scope may be mutated by any code that imports the package.

This is not the case for constants, since they cannot be mutated after first initialisation. For this reason, there's a strong argument for only using constants in the global scope.

A variable of the same name can be declared in multiple scopes, as each scope has a separate namespace. Avoid this if possible, it can lead to something we call *variable shadowing* where it can become unclear what value is stored in the variable we are using in a specific scope.

Consider the example below where a variable is declared in both the global and function scope.

**Example 18 - Global and function scope**

```
package main

import "fmt"

var myVar = 10

func main() {
        myVar += 5
        var myVar int
        myVar -= 5
        fmt.Println("myVar is:", myVar)
}

// Go Playground: https://go.dev/play/p/V7OT8HX_XMP
```

Go will use the locally scoped variable in preference to any parent scoped variable. In the example above we printed the `myVar` variable declared in `main()` as that was the locally scoped variable.

The same is also true for declarations made inside control structures like loop and conditional statements. Go will use the variable inside the control structure, in preference to the function or globally scoped variable.

**Example 19 - Local scope in control structure**

```
package main

import (
        "fmt"
)

var myVar = 10

func main() {
        // global scope
        fmt.Println(myVar)

        myVar := 5

        for i := 0; i < 1; i++ {
                //control structure scope
                myVar := 0
                fmt.Println(myVar)
        }
```

```
20
21          // function scope
22          fmt.Println(myVar)
23  }
24
25  // Go Playground: https://go.dev/play/p/edNkZtJuCPk
```

Output:

```
1  10
2  0
3  5
4
5  Program exited.
```

For this reason, duplicated variable names should be avoided for all but the most generic of values, such as errors.

## 6.4 Variable semantics. Pointers and values

When working with variables we often need to pass them around as arguments and return values in functions and receivers.

In Go, we can share variables as *values* or *pointers*. A value is a copy of the contents of the variable, whereas a pointer is a copy of the address of the variable in memory.

As stated in an earlier chapter, both are values, just different kinds of values, but to avoid confusion here, when we say *pass by value* we mean sharing a copy of the variable, whereas with *pass by reference* we mean sharing the memory address of the variable.

When we pass something to a function by value, the function receives a copy of that value. This could be ideal in a function which prints simple output, like a username for example, but, passing a copy of a value may not be ideal if we are passing a large struct around in our program, with each copy duplicating that value in memory.

The fact that we have a copy of the original value, and not the original value itself means we cannot mutate the original. In many cases, this change protection will be exactly what we want in code which should be able to read but not amend a value.

By contrast, if we pass by reference, the value we pass is the address in memory of the variable. It is not a copy of the variable, nor is it the original variable, it is simply the address of the original variable in hexadecimal form, which could look like this `0xc0000ac018`.

Using the memory address we can get the original variable stored in memory at that address by *dereferencing* the value and we may, if we wish, mutate that value using the same approach.

Two examples follow which show the syntax used to pass by value, pass by reference and deference the value.

In the first example, we have a function that accepts a value of type string. Although we attempt to change it, we are mutating a copy of the value and not the original, so the final output is unchanged.

**Example 20 - Pass by value**

```
1  package main
2
3  import (
4          "fmt"
5  )
6
7  func SayHello(name string) {
8          fmt.Printf("Hello %s\n", name)
9          name = "Dave Blogs"
10 }
11
12 func main() {
13         name := "Joe Blogs"
14         SayHello(name)
15         fmt.Println(name)
16 }
17
18 // Go Playground: https://go.dev/play/p/uESKRz6r-tS
```

Output:

```
1  Hello Joe Blogs
2  Joe Blogs
3
4  Program exited.
```

Moving to the second example, the `SayHello()` function signature has been modified to accept a pointer to a value of type string. We use `*string` in the signature to denote this, and this function will no longer accept the value itself.

**Example 21 - Pass by reference**

```
1  package main
2
3  import (
4          "fmt"
5  )
6
7  func SayHello(name *string) {
8          fmt.Println(name)
9          fmt.Printf("Hello %s\n", *name)
10         *name = "Dave Blogs"
11 }
12
13 func main() {
14         name := "Joe Blogs"
15         SayHello(&name)
16         fmt.Println(name)
17 }
18
19 // Go Playground: https://go.dev/play/p/7Ne6OPnCs1N
```

Output:

```
1  0xc00009e210
2  Hello Joe Blogs
3  Dave Blogs
4
5  Program exited.
```

On line 15 we take a pointer to the `name` variable by prefixing it with the *&* character and pass that as the single argument to `SayHello()`.

On line 8 we print the pointer stored in the `name` parameter which outputs a memory address, and on line 9 we dereference the value using the asterisk prefix - as in we ask for the value stored at that memory address - so that we may print it.

Finally, on line 10 we assign a new value to the dereferenced `name` variable and we can see that we have mutated the original variable as when we print it again at line 16, the new value is displayed.

## 6.5 Value initialisation

Building on the information in the previous section we're going to look at how we can create values and pointers.

Values are simple, we've seen them many times in this section already. Whenever we create a variable and share it, we ordinarily pass it by value. Of course, we can obtain a pointer to it after creation if needs be, as we did at line 15 in the previous example.

So what if we want to work with pointers exclusively and we don't want the value which can be copied, but a single representation of the thing in memory. To do this we need to take a pointer directly - the memory address - and share that.

There are three ways to achieve this. We can use the short-form *&* prefix,the *new* function, and *make*. All give us a pointer to a value in memory.

The example below shows two approaches relevant to creating a *struct* type. We'll cover structs in the next section, and later we'll also look at the third approach, *make* when we consider reference types.

**Example 22 - Obtaining a pointer directly**

```
1  package main
2
3  import "fmt"
4
5  type User struct {
6          Name string
7  }
8
9  func ChangeName(name string, user *User) {
10         user.Name = name
11 }
12
13 func main() {
14         u1 := new(User)
15         u2 := &User{
16                 Name: "Joe Blogs",
17         }
18
19         ChangeName("Dave Blogs", u1)
20         ChangeName("Trevor Blogs", u2)
21
22         fmt.Println("u1:", u1)
23         fmt.Println("u2:", u2)
24 }
25
26 // Go Playground: https://go.dev/play/p/t3iJJ47laIt
```

Output:

```
u1: &{Dave Blogs}
u2: &{Trevor Blogs}

Program exited.
```

On line 14 we use the new keyword to obtain a pointer to a *nil* value of *User* and on line 15 we use the *&* prefix to obtain another pointer to a *User* using struct literal syntax, which allows us to initialise it to a non-nil value.

Did you notice anything different in this example compared to Example 21? We didn't need to dereference the structs either to assign a new value or to print them, when previously we did?

Go performs automatic dereferencing on struct types for us which results in a much cleaner syntax.

Consider the `changeName()` function in the previous example. The syntax below is equivalent to that in Example 22. Although it is perfectly valid, and the compiler will not complain, it is less readable, so we should let Go perform the dereferencing on our behalf:

```
func ChangeName(name string, user *User) {
        (*user).Name = name
}
```

Note that when we print the structs at lines 26 & 27, although Go performs the dereferencing automatically and prints the struct values it reminds us that we are working with pointers by prefixing the output with the *&* character.

# Chapter 7 - Data types

In this chapter we're going to review Go's built-in data types, often referred to as *primitives*.

Data types define what a value can be and are fixed at compile time. In Go conversions between types must be performed explicitly.

We can split Go data types into several categories:- basic, aggregate, reference and interface, although the classifications are not strictly mutually exclusive, as we will see.

The term *reference type* has been removed from official Go terminology, and instead aggregate types and reference types are often referred to collectively as *composite types*. We're going use the term anyway, and we'll see why soon.

## 7.1 Basic types

Basic types include types for numbers, strings and booleans.

### 7.1.1 Number

There are multiple ways to represent numbers depending on how big that number is, whether it is positive or negative and whether it's a whole number or a decimal.

Unless your program code needs to be optimised for memory usage, choose *int* for whole numbers and *float64* for decimals. Both types use 64 bits or 8 bytes in memory.

Don't use specific integer types such as *int8*, *int16*, *int32* or *int64* unless forced to by an external dependency which accepts or returns these types specifically.

Similarly, if you **do** choose to store decimals as *float32* in your program, to perform operations on these values, for example using the standard library math package[5], you'll need to convert them to *float64*, since the package accepts and returns *float64* types exclusively.

Performing type conversions such as the above may complicate your syntax and in many cases, this trade-off won't be worth the memory saved by using types that occupy less memory.

If memory constraints do become a concern, you can revisit the types used by your program of course. The point is this is a performance optimisation, which should come later.

[5] https://pkg.go.dev/math

Did you know *int* is actually an alias for *int32* or *int64*. Which, depends on whether your code is compiled for 32-bit or 64-bit architecture. No such alias exists for *float* although *float* was once a valid type but was removed from the language in 2011 before Go version 1.0 was released.

Go also supports unsigned integer types, where the value must be positive. For reasons already stated should you wish to use an unsigned integer use *uint* rather than a more specific type.

In addition to *int*, *uint* and *float64* types there are *complex* number types available in Go, but we're not going to cover those.

Variables of any number type are initialised to zero.

Before leaving number types, a note about working with floating point numbers. Below is a *Gotcha* relating to floating point number precision.

**Example 23 - Floating point number addition problem**

```
1  package main
2
3  import (
4          "fmt"
5  )
6
7  func main() {
8
9          a := 2.1
10         b := 4.2
11
12         c := a + b
13         d := 2.1 + 4.2
14         e := float64(2.1) + float64(4.2)
15
16         fmt.Println(c == d)
17         fmt.Println(c == e)
18 }
19
20 // Go Playground: https://go.dev/play/p/_uoAHQeoh9h
```

Output:

```
false
true

Program exited.
```

Looking at the simple additions in the example, we might expect both comparisons to be *true* but they are not. This is due to floating point number rounding precision.

A real-world scenario in which we could encounter problems caused by different precision is in unit testing. An assertion that should pass, fails because the expected result is ever so slightly different than the result obtained.

The takeaway is that comparisons between floating point numbers can be unreliable where their precision is different, so if you wish to add decimal numbers which are not already *float64* type it is worth performing that conversion explicitly.

It's also worth ensuring the same level of precision when making assertion comparisons in your unit tests.

## 7.1.2 Boolean

The type *bool* stores true and false values and it uses one byte in memory. A variable of type *bool* is initialised to its *nil* value which is false.

## 7.1.3 String

Though a basic type, a *string* actually points to a backing slice of bytes. We cover the *byte* type next but think of a slice as an array for the moment, we'll also cover them soon.

To demonstrate, in the example below we print a portion of the string from index position 6 onwards by accessing the underlying byte slice:

**Example 24 - String cut using byte slice**

```
package main

import "fmt"

func main() {
        myString := "Hello GolangAtSpeed"
        fmt.Println(myString[6:])
}

// Go Playground: https://go.dev/play/p/egXJA0gm97x
```

In the example above we make a common mistake. The code could result in unexpected or buggy behaviour depending on the string literal we operate on. We'll point out the gotcha shortly when we get to the *rune* type.

Variables of type *string* have a value of an empty string at declaration.

### 7.1.4 Byte

A *byte* is an alias for *uint8*. It stores integers between 0 and 255 (the largest number we can represent in binary with 8 bits). Therefore a *byte* can be used to represent any single ASCII character.

As an alias for *uint8*, a variable of type *byte* is initialised to zero.

### 7.1.5 Rune

We'll start this section with a classic gotcha! Check out the example program below in which we have two 'identical' strings but which appear to be of different length:

**Example 25 - Identical strings with different lengths**

```
1  package main
2
3  import (
4          "fmt"
5  )
6
7  func main() {
8          greet1 := "hello"
9          greet2 := "ello"
10
11         fmt.Println(len(greet1))
12         fmt.Println(len(greet2))
13 }
14
15 // Go Playground: https://go.dev/play/p/ujUnmx-LsWu
```

Don't worry, we'll explain what's happening here in our discussion of the *rune* type which follows.

To represent other characters beyond ASCII alone we need to use a *rune*.

A *rune* is an alias for *int32* which can represent a Unicode codepoint as a numerical value. As it's an *int32* it is large enough to represent any Unicode character in a sequence of almost 150,000 at the time of writing.

ASCII is a subset of Unicode so ASCII character codes map directly to the same character codepoint in Unicode.

A Unicode codepoint also has a UTF-8 hexadecimal representation. Go will present this hex as a sequence of up to 4 bytes, how many bytes, depends on the UTF-8 hexadecimal code itself.

Let's illustrate with an example. In the program below we've a string which contains a single character - which happens to be our favorite Unicode character - the rocket. You can inspect the character and its data here[6].

**Example 26 - Unicode codepoint with UTF-8 code**

```
package main

import (
        "encoding/hex"
        "fmt"
)

func main() {
        myString := "🚀"
        fmt.Println("Unicode codepoint represented by rune:", []rune(myString))
        fmt.Println("UTF-8 code represented by up to 4 bytes:", []byte(myString))
        fmt.Println("UTF-8 code represented as Hexadecimal:", hex.EncodeToString([]byte(myS\
tring)))
        fmt.Println("length", len(myString))
}

// Go Playground: https://go.dev/play/p/luDIj6DwPAG
```

Output:

```
Unicode codepoint represented by rune: [128640]
UTF-8 code represented by up to 4 bytes: [240 159 154 128]
UTF-8 code represented as Hexadecimal: f09f9a80
length 4

Program exited.
```

The output shows the Unicode identifier for the rocket character stored as a rune. Then, the same UTF-8 hexadecimal representation encoded as a byte slice.

Next we output the bytes, converted to their hexadecimal equivalent. Note, this hexadecimal sequence matches the UTF-8 sequence for the rocket on the website above.

[6] https://codepoints.net/U+1F680

Finally, we output the string "length" and we see something odd. We get a length of `4` when the string is only one character. That's right, `len()` operates on the underlying byte slice by default - not the string.

Careful here, it's a common mistake to use `len()` to determine the length of a string. If you're dealing purely with ASCII it will be fine, because one character is represented by one byte - swap a simple ASCII character like *a* into the Go Playground example to see for yourself - but if your string contains Unicode characters, `len()` will not be accurate, unless you wanted the number of bytes, not the number of actual characters. Think back to the string example in 7.1.3. What might happen if we printed a substring of a string which contained Unicode characters?

To safely get the length of a string and not the length of the byte representation of the string, first convert it to a slice of runes as shown in the following example:

**Example 27 - Safely obtaining length of a string**

```
1  package main
2
3  import (
4          "fmt"
5  )
6
7  func main() {
8          myString := "□"
9          fmt.Println("length", len([]rune(myString)))
10 }
11
12 // Go Playground: https://go.dev/play/p/wTnoddQnjvJ
```

Output:

```
1  length 1
2
3  Program exited.
```

As an alias for *int32* a variable of type *rune* is initialised to zero at declaration.

Going back to the example at the start of this section, the second string contains a cyrillic character[7] that looks like a "h" but it isn't. It's not an ASCII character, it's Unicode with a UTF-8 encoding of `D2 DB`. We need two bytes - and not one - to represent this character in the underlying byte slice, so the string length appears longer.

[7] https://codepoints.net/U+04BB

# 7.2 Aggregate types

Aggregate types are types that contain aggregates of other data types, either basic types or nested aggregate, reference and interface types.

In Go we refer to *array* and *struct* types as aggregate types.

## 7.2.1 Array

An *array* can contain a fixed number of elements of a single type.

Elements in an array are initialised to their *nil* value. For example, an array of type *int* with 5 elements, will have all 5 elements initialised to zero. Length of the array will be 5.

An *array* cannot be resized once declared, its length is fixed, and a part of its type. A variable with type `[5]int` is not considered to be of same type as a variable with type `[6]int`, despite both arrays holding integers.

An *array* has the same capacity as its length. We'll cover *capacity* later when we discuss slices.

Arrays, as in many programming languages, are "zero-bound". The first element in the array will be referenced by an index of 0 and not 1.

Two example programs are included next to demonstrate what we've discussed above.

**Example 28 - Array length, capacity and element initialisation**

```
1  package main
2
3  import (
4          "fmt"
5  )
6
7  func main() {
8          var intArray [5]int
9
10         fmt.Println("Array length:", len(intArray))
11         fmt.Println("Array capacity:", cap(intArray))
12
13         for i := range intArray {
14                 fmt.Println("Array element index", i, "contains value", intArray[i])
15         }
16
17         intArray[0] = 1
18         fmt.Println(intArray)
19 }
```

```
20
21  // Go Playground: https://go.dev/play/p/D_hs2NHHoAs
```

Output:

```
1  Array length: 5
2  Array capacity: 5
3  Array element index 0 contains value 0
4  Array element index 1 contains value 0
5  Array element index 2 contains value 0
6  Array element index 3 contains value 0
7  Array element index 4 contains value 0
8  [1 0 0 0 0]
9
10 Program exited.
```

Walking through the previous example, we first declare a variable of type *[5]int*. We print length and capacity which are both 5, and then iterate over the elements using the *range* keyword to show that each element in the array has been initialised to its nil value, which is zero for the *integer* type. For completeness, we then make a simple assignment to the first element in the array and print the entire variable.

The next example demonstrates how length is part of an arrays type.

**Example 29 - Array length is part of its type definition**

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          var intArray [5]int
7          var intArray2 [6]int
8
9          if intArray == intArray2 {
10         // this will never print
11                 fmt.Println("Array values and type are same")
12         }
13 }
14
15 // Go Playground: https://go.dev/play/p/xgNuJjZQHZM
```

Output:

```
./prog.go:9:17: invalid operation: intArray == intArray2 (mismatched types [5]int an\
d [6]int)

Go build failed.
```

In this final example, we show two ways to declare and initialise an array to non-zero values:

**Example 30 - Set array length with the spread operator**

```
package main

import "fmt"

func main() {
        intArray := [3]int{1, 2, 3}
        intArray2 := [...]int{4, 5, 6}

        fmt.Println(intArray)
        fmt.Println(intArray2)
}

// Go Playground: https://go.dev/play/p/bxX2YplIPa8
```

First, we declare a variable *intArray* of type [3]int by specifying the length and capacity explicitly.

When declaring the second variable *intArray2* we do much the same, but instead, we use the number of elements in the array to initialise it using the *spread operator* denoted by the ellipse (three dots).

The spread operator has other uses, which we'll see as we progress.

Arrays are rarely used in Go as most use cases favour slices over arrays which are generally more flexible. However, where we know the number of elements we need ahead of time, and this number is fixed, it can be more memory-efficient to use an array, and we'll see why when we discuss slices later in the chapter.

## 7.2.2 Struct

A *struct* - short for structure - is a type which can hold a collection of fields which are themselves typed. *Structs* are useful for representing collections of data in our programs.

Structs can be used *anonymously* - their properties defined at the point of use - which is useful if we only need a temporary representation of something, or, we can use the struct type as a basis for building our own named custom types that represent the collections we work with in our program.

In both cases, fields are initialised to their nil values unless explicitly provided a value.

We show both approaches in the example below:

**Example 31 - Named custom struct type and anonymous struct**

```
1  package main
2
3  import "fmt"
4
5  // Named custom struct
6  type User struct {
7          Name string
8          Age  int
9  }
10
11 // Second named custom struct
12 type User2 struct {
13         Name string
14         Age  int
15 }
16
17 func main() {
18         var user User
19
20         // Anonymous struct created at point of use
21         data := struct {
22                 Name string
23                 Age  int
24         }{
25                 Name: "Joe Blogs",
26                 Age:  20,
27         }
28
29         user = data
30
31         //var user2 User2
32         //user2 = user
33
34         fmt.Println(user)
35 }
36
37 // Go Playground: https://go.dev/play/p/gWnMhWF_OVK
```

At line 6 we define a named custom type *User* and we create a value of that struct type at line 18 in the variable *user*.

At line 21 we create an anonymous struct literal to represent some data, it's fields happen to match the named *User* structs fields.

Since they match, we can assign the *data* variable directly to the *user* variable as we do on line 29.

We can **only** do this with anonymous struct types, not with named custom struct types.

To illustrate, if we uncomment line 31 and 32 and attempt to run the example code again we'll get a compilation error, despite *User* and *User2* types having identical fields:

```
./prog.go:32:10: cannot use user (variable of type User) as type User2 in assignment

Go build failed.
```

Struct values can be compared using simple comparison operations. For them to be considered equal, they must either be of the same type or, one or both structs must be anonymous, and the values of all of their fields must also be equal.

With both custom struct types and anonymous structs, where we define them in their struct literal form we can omit the field names in the values on two conditions. First, we must provide the values in the same field order as the struct's type definition and second, we must provide a value for all fields.

There's a brief example showing this form below:

**Example 32 - Unnamed struct fields**

```
package main

import "fmt"

// Named custom struct
type User struct {
        Name string
        Age  int
        Sex  string
}

func main() {
        user := User{
                "Joe Blogs",
                20,
                "Male",
        }
        fmt.Println(user)
}
```

```
20
21  // Go Playground: https://go.dev/play/p/hun_q6dx-VC
```

Once declared we also can get and set struct fields by using a dot separator as below.

**Example 33 - Get and set struct fields**

```
1   package main
2
3   import "fmt"
4
5   // Named custom struct
6   type User struct {
7           Name string
8           Age  int
9   }
10
11  func main() {
12          var user User // all fields initialised to nil value
13          user.Name = "Joe Blogs"
14          user.Age = 20
15          fmt.Printf("User is %s. They are %d years of age", user.Name, user.Age)
16  }
17
18  // Go Playground: https://go.dev/play/p/gM5fADxC9gB
```

### 7.2.2.1 Composition and embedding

Structs can be embedded inside other structs. By taking smaller pieces of code we can combine their attributes and behaviour into a new piece of code and add extra attributes and behaviour we need.

In Go, we call this *composition*.

We stated in chapter one that *composition* roughly aligns with inheritance in OOP but there are key differences.

Many OOP languages do not offer multiple inheritance, so users can only inherit one superset class of functionality. When using Go's composition we can embed unlimited structs, each representing specific attributes and behaviour.

For clarity, the *behaviour* we talk of is provided by *receivers* which can be created on the custom struct types, just as on other any custom type.

Typically in OOP languages we inherit one big thing that *almost* does what we want, then extend and override it in our class to make it do want we need. In Go, composition is like building the thing from smaller pieces. This provides more flexibility but also the potential for more code reusability.

When we embed a struct inside another struct we *can* promote its fields and receivers so that they can be used as if they were a part of that struct.

The example below illustrates composition through struct embedding.

**Example 34 - Composition and struct embedding**

```
1  package main
2
3  import "fmt"
4
5  type Eyes struct {
6          Color string
7          Shape string
8  }
9
10 type Human struct {
11         Eyes
12 }
13
14 type Dog struct {
15         Eyes Eyes
16 }
17
18 func main() {
19         var human Human
20         var dog Dog
21
22         // promoted field access
23         human.Color = "Blue"
24         human.Shape = "Round"
25
26         // long form field access
27         human.Eyes.Color = "Brown"
28
29         // dog.Color = "Brown" // won't build
30
31         fmt.Println(human, dog)
32 }
33
34 // Go Playground: https://go.dev/play/p/6DSmzLgW3Xo
```

Picking out the important pieces, we embedded the *Eyes* struct in the *Human* struct. We can access its fields (and receivers) as if they were directly associated with the *Human* struct, as seen on lines 23 & 24.

However, in this example, although we *can* do this, we lose some clarity in our syntax - what do colour and shape refer to - so we may consider using the long form field accessor, which is still available, as shown on line 27.

Note, the *Dog* struct also uses the *Eyes* struct, but we do not embed it this time. Instead, we use it as the type on a named field *Eyes*. In this situation, only long-form access is possible, there is no field or receiver promotion.

Note also that when embedding struct types from imported packages the usual rules on visibility apply. Only exported functionality can be used.

Finally, the example serves also to illustrate the benefits of composition. We can reuse *Eyes* which describes attributes of colour and shape for anything that has eyes, of which *Human* and *Dog* are but just two examples.

#### 7.2.2.2 Memory use, alignment & padding

Before leaving structs we need to briefly mention memory use, and specifically alignment and padding.

Memory usage may be a consideration if a struct has a large number of fields and we are working with a many of these struct values in memory at the same time - perhaps when retrieving a list of rows from a database table of users, for example.

In such circumstances, it *may* be necessary to consider how struct fields are ordered, and possibly alternative integer types rather than the single *int* we've recommended thus far. The reason comes down to how structs and their fields are stored in memory.

Think of a 64-bit architecture as storing data in 8-byte slots. An *int* variable, which in this architecture is an alias for *int64*, requires 8 bytes in memory and so would sit perfectly inside one of these slots.

Two *int32* types would occupy a slot, whereas an *int8* type occupies a single byte in memory, as does a *bool* type, so we could hold eight booleans or eight *int8* typed variables in a single 8-byte slot.

For efficiency of access, adjacent struct fields that cannot fit in the current 8-byte slot will use the next 8-byte slot in memory. This results in some level of *padding* in slots that do not utilise all eight bytes, meaning that not all memory used by the struct, is holding data.

Field ordering determines how much padding is required, so, large structs with suboptimal field type ordering may use significantly more memory, which can become an issue when we need to hold many of these struct values in memory at once.

It's simple to demonstrate with an example.

**Example 35 - Alignment and impact on memory**

```
package main

import (
        "fmt"
        "unsafe"
)

type Suboptimal struct {
        bool1 bool
        int1  int
        bool2 bool
        int2  int
        bool3 bool
        int3  int
        bool4 bool
        int4  int
}

type Optimal struct {
        bool1 bool
        bool2 bool
        bool3 bool
        bool4 bool
        int1  int
        int2  int
        int3  int
        int4  int
}

func main() {
        subOptimal := [1000]Suboptimal{}
        sizeSuboptimal := float64(unsafe.Sizeof(subOptimal))

        optimal := [1000]Optimal{}
        sizeOptimal := float64(unsafe.Sizeof(optimal))

        fmt.Printf("Size of suboptimal array = %v bytes\n", sizeSuboptimal)
        fmt.Printf("Size of optimal array = %v bytes\n", sizeOptimal)

        diff := (sizeSuboptimal - sizeOptimal) / sizeSuboptimal * 100
        fmt.Printf("Padding means we're using %d%% more memory...", int(diff))
}
```

```
43
44 // Go Playground: https://go.dev/play/p/q7GJI1xfEqb
```

Output:

```
1 Size of suboptimal array = 64000 bytes
2 Size of optimal array = 40000 bytes
3 Padding and int type means we're using 37% more memory...
4
5 Program exited.
```

In the first *Suboptimal* struct we have an admittedly extreme example. Due to alignment, each *int* field is using the next 8-byte slot resulting in 7 bytes of padding for each *bool*, so 28 bytes of padding in total per struct value.

In the second *Optimal* struct, all of our *bool* fields fit in a single 8-byte slot, and all *int* fields take one 8-byte slot each. Because of this, the total padding for each struct value is only 4 bytes.

# 7.3 Reference types

In this section, we'll look at three other important Go data types: *maps*, *slices* and *channels*.

Recall we said *reference type* isn't official Go terminology but that we're going to use it anyway. The distinction is helpful when learning about pointer and value semantics, specifically when it is necessary to pass a pointer to mutate the original value and when it is not.

The reason we're classing map, slice and channel as reference types is that values of these types behave as if they *were* pointers.

All are essentially descriptors which point to (or reference) some kind of backing data. When using the descriptor, operations are performed on that backing data automatically, there's no need to use a pointer to work with the backing data.

Functions and pointers are also reference types by the above definition, but we're not covering them here. We're also specifically excluding strings. Of course, we know a string points to a backing slice of bytes, so fits the above definition, but strings are a special case, the backing data cannot be mutated directly.

Let's look at how we work with each in turn.

## 7.3.2 Map

The *map* type provides for associative data storage. Quite simply, keys can be associated with values and stored in memory. Both *key* and *value* have a type which may be different and together they comprise the maps type definition.

A map must be initialised before use. Prior to initialisation we have a *nil map* which we cannot use. Attempting to do so will cause our program to panic.

Post-initialisation we have an *empty map*, a descriptor which points to an initialised area of memory in which we can add our key and value pairs.

In the following example, we declare a variable of type `map[string]string` meaning the key and value are both of type string. We then try to work with the nil map, without initialising it, and our program panics.

Uncomment line 7 and run again to initialise the map using the `make` keyword. Our program now runs as expected.

**Example 36 - Trying to use a nil map**

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          var myMap map[string]string
7          //myMap = make(map[string]string)
8
9          myMap["key"] = "value"
10         fmt.Println(myMap)
11 }
12
13 // Go Playground: https://go.dev/play/p/dOBBmtyVYuV
```

Output:

```
1  panic: assignment to entry in nil map
```

Maps can be initialised to an empty map, with the `make` keyword or by using a map literal which may, optionally, be populated with keys and values.

The next example shows the options available when creating an empty map. The comments describe each approach.

**Example 37 - Creating an empty map**

```
package main

import "fmt"

func main() {
        // declaration followed my initialisation with make
        var myMap map[string]string
        myMap = make(map[string]string)

        // declaration & initialisation with make
        myMap2 := make(map[string]string)

        // declaration & initialisation as empty map literal
        myMap3 := map[string]string{}

        // declaration & initialisation as map literal with key/values
        myMap4 := map[string]string{
                "key": "value",
        }

        fmt.Println(myMap, myMap2, myMap3, myMap4)
}

// Go Playground: https://go.dev/play/p/kDhy20FhD6B
```

Shortly, we'll explore how to work with maps in our code, but before that here's a quick demonstration of the *reference type* qualities of maps.

In the example below we pass the map by value and not as a pointer. Despite this, we are able to mutate the original value, because maps behave as if they were pointers:

**Example 38 - Map passed by value**

```
package main

import "fmt"

func alterMap(mp map[string]string) {
        mp["myKey"] = "my new value"
}

func main() {
        myMap := map[string]string{
```

```
                "myKey": "my value",
        }

        fmt.Println(myMap)
        alterMap(myMap)
        fmt.Println(myMap)
}

// Go Playground: https://go.dev/play/p/i20LKvkaC0J
```

Output:

```
map[myKey:my value]
map[myKey:my new value]

Program exited.
```

We *can* apply pointer semantics and the program will build and run. Indeed, in the example below we rewrite `alterMap()` to accept a pointer to the map.

**Example 39 - Map passed by reference**

```
package main

import "fmt"

func alterMap(mp *map[string]string) {
        (*mp)["myKey"] = "my new value"
}

func main() {
        myMap := map[string]string{
                "myKey": "my value",
        }

        fmt.Println(myMap)
        alterMap(&myMap)
        fmt.Println(myMap)
}

// Go Playground: https://go.dev/play/p/lgiCDOYzSrj
```

We get the same result from the program. However, it achieves nothing extra but makes the syntax less clear. We shouldn't do this in our code, instead, we should pass maps by value and let the compiler worry about the semantics.

#### 7.3.2.1 Working with maps

We can create map keys and values by simply specifying the key and assigning it a value. The syntax is very similar when we access a key to obtain its value.

As well as accessing keys individually to obtain the corresponding value, we may also access the entire contents of a map by iterating over it in a loop. When iterating in this manner, it is impossible to predict in what order keys will be retrieved since map access is non-deterministic: keys are fetched neither on a LIFO nor FIFO basis, but instead pseudo-randomly.

We can determine the length of a map - how many keys it contains - using the `len()` function which we've seen previously when discussing arrays.

Removing a key from a map is performed with the `delete` keyword. Currently, keys must be removed one by one, there is no way to delete an entire map, although this feature is in an experimental Go branch[8]

The example below demonstrates the map operations we've just discussed, the code is commented to describe the operations but it should be fairly obvious.

**Example 40 - Working with maps**

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          var myMap = make(map[string]string)
7
8          // create three keys in the map
9          myMap["key1"] = "value1"
10         myMap["key2"] = "value2"
11         myMap["key3"] = "value3"
12
13         // obtain the number of key/values
14         fmt.Println("Map keys:", len(myMap))
15
16         // access the value stored for a key
17         fmt.Println(myMap["key1"])
18
19         // iterate over the map with range and print each key/value
```

[8] https://go.dev/play/p/qIdnGrd0CYs?v=gotip

```
20          for k, v := range myMap {
21                  fmt.Println(k, v)
22          }
23
24          // delete key3
25          delete(myMap, "key3")
26
27          fmt.Println(myMap)
28  }
29
30  // Go Playground: https://go.dev/play/p/6-ht0hCoQFj
```

### 7.3.2.2 Safely accessing keys in a map

Obviously, care should be taken to ensure program code cannot read and write to a map - in fact to any variable - in a way that leaves the variable in an uncertain state. Known as a race condition, this is often a problem in asynchronous code, and it's something we'll cover in a later chapter when we look at goroutines and concurrency.

In this section, we are still concerned about safe access, but in ensuring that if we request the value stored for a key in the map, that the requested key exists in the map.

Should we request the value for a key which is not present, our program will return the nil or zero value for the map value type. To avoid this risk, we can make use of a second return value when we try to access a key. This second return, a bool, will be false if the key is not present and true if it is.

This is a common cause of intermitment bugs. If we request the value of a key that does not exist, the returned nil value could be perfectly valid and our program will proceed to process that value as if the key had existed. That might not go to well for us.

Two short examples follow. In the first, we do not check for the existence of a key, and our program receives valid data for a map key which does not exist:

**Example 41 - Unsafe map access**

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          activeUser := map[string]bool{
7                  "Joe Blogs":  true,
8                  "Dave Blogs": false,
9          }
```

```
10
11          // keys exists no issue
12          fmt.Println("Joe is active:", activeUser["Joe Blogs"])
13          fmt.Println("Dave is active:", activeUser["Dave Blogs"])
14          // key does not exist
15          fmt.Println("Trevor is active:", activeUser["Trevor Blogs"])
16  }
17
18  // Go Playground: https://go.dev/play/p/9jX2B7x1eha
```

Output:

```
1  Joe is active: true
2  Dave is active: false
3  Trevor is active: false
4
5  Program exited.
```

In this second example, we correctly verify the key exists in the map, before attempting to work with the value associated with the key:

**Example 42 - Safe map access**

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          activeUser := map[string]bool{
7                  "Joe Blogs":  true,
8                  "Dave Blogs": false,
9          }
10
11         if active, ok := activeUser["Trevor Blogs"]; !ok {
12                 fmt.Println("User does not exist")
13         } else {
14                 fmt.Println("Trevor is active:", active)
15         }
16 }
17
18 // Go Playground: https://go.dev/play/p/a4zZyEM8viF
```

Output:

```
User does not exist

Program exited.
```

The takeaway is that we should always check for the existence of key in a map before trying to use its value in our program.

### 7.3.3 Slice

Slices are one of the most versatile and commonly used data types in Go. Often we'll use a slice in preference to a sized array without evening thinking about it. On other occasions, we may choose a slice specifically because we need a container for an unknown number of elements, which makes an array unsuitable, or certainly harder to work with.

Slices point to an underlying backing array of data, but Go manages that array for us. Slices have a length and a capacity. Depending on how we create the slice, length and capacity can be the same, or different, which is not the case for arrays, where both length and capacity are the same.

Length, determined using the `len()` command is the number of elements which are initialised in the slice, either to a value of our choosing or to the nil value for the slice type. Capacity, which we can determine with the `cap()` command is how many elements in total, are available in the backing array that underpins the slice.

In the example below we create a slice of integers in five slightly different ways.

**Example 43 - Creating a slice**

```
package main

import "fmt"

func main() {
        var sl1 []int
        sl2 := []int{}
        sl3 := []int{1, 2, 3, 4, 5}
        sl4 := make([]int, 5)
        sl5 := make([]int, 1, 4)

        // check length
        fmt.Println(len(sl1), len(sl2), len(sl3), len(sl4), len(sl5))

        // check capacity
        fmt.Println(cap(sl1), cap(sl2), cap(sl3), cap(sl4), cap(sl5))
```

```
17
18          if sl1 == nil {
19                  fmt.Println("sl1 is nil!")
20          }
21          if sl2 != nil && sl3 != nil && sl4 != nil && sl5 != nil {
22                  fmt.Println("sl2, sl3, sl4 and sl5 are not nil!")
23          }
24  }
25
26  // Go Playground: https://go.dev/play/p/922R9tr-3Aq
```

Output:

```
1  0 0 5 5 1
2  0 0 5 5 4
3  sl1 is nil!
4  sl2, sl3, sl4 and sl5 are not nil!
5
6  Program exited.
```

Variable `sl1` is what is referred to as a *nil slice*. At this point, it has no length nor capacity and indeed no backing array. Only a nil slice is equal to `nil`.

`sl2` creates an empty slice literal. Again length and capacity are zero but we have a backing array. An empty slice like this cannot be compared to `nil`.

Variable `sl3` uses the same approach but initialises the slice with data such that its length and capacity equal the number of elements added.

`sl4` is created with the `make` keyword. The single number `5` is used to set both the length and capacity to 5, whereas variable `sl5` uses `make` again to set the length and capacity to different values.

Slices are a reference type because there's a backing data array, but we need to caveat that statement, because slices behave as a reference type only in certain circumstances. Lets look at why.

Every slice has a descriptor - or slice header - which contains three important pieces of information about the slice. First, it contains a pointer to the backing array where the actual slice data is stored. Second, it contains the current length and lastly, the capacity of the backing array.

We can add new data to a slice and not concern ourselves with the finite length of the backing array. When the backing array has no capacity left in which to store data, Go will automatically create a new array on our behalf and copy the contents of the old array over.

At this point, the slice header is changed. The address of the new array is stored and we have a new length and capacity.

It's worth us understanding the impact of this backing array replacement and we'll cover it in more depth later in this section.

Where there is sufficient capacity in the backing array to add the new element, the slice header is still changed because length is a property of the slice header and that is updated.

So, when does a slice behave as a reference type and when does it not?

The general rule is that if an operation on a slice does not modify the slice header it *can* be used as a reference type.

For example, if we add elements which exceed the capacity of the slice backing array, the slice header is updated with a new backing array. This is an entirely new value, elements added to this array will not be visible to the calling code.

Alternatively, if we alter existing elements in the slice and don't add new elements, we don't change the slice header. In such circumstances we're able to modify the original value's elements, and the caller will see the changes.

The following three examples demonstrate the different behaviours.

In the first example, we need a larger backing array as a result of the append operation in the `addToSlice()` function.

Go dutifully copies any existing elements over to a new array with new length and capacity. The slice header is changed, and the append operation is performed as expected, but to the new backing array.

The original is unchanged when we print it again on line 19. This append operation has effectively *unlinked* the slice from the original value we passed to `addToSlice()`.

**Example 44 - Slice not behaving as a reference type**

```
1  package main
2
3  import "fmt"
4
5  func addToSlice(sl []int) {
6          sl = append(sl, 2)
7          fmt.Println(sl)
8  }
9
10 func main() {
11         sl := []int{}
12
13         sl = append(sl, 1)
14         fmt.Println(sl)
15
```

```
        addToSlice(sl)
        fmt.Println(sl)
}

// Go Playground: https://go.dev/play/p/0cOx0y5Xc_L
```

Output:

```
[1]
[1 2]
[1]

Program exited.
```

To safely add to a slice in a function such as the above, recognising the append operation is creating a new value, we should return the new value to the caller, and share that going forwards. We show this below.

**Example 45 - Safely return the new slice to caller**

```
package main

import "fmt"

func addToSlice(sl []int) []int {
        sl = append(sl, 2)
        return sl
}

func main() {
        sl := []int{}

        sl = append(sl, 1)
        fmt.Println(sl)

        sl = addToSlice(sl)
        fmt.Println(sl)
}

// Go Playground: https://go.dev/play/p/A-Lg98S0GRJ
```

Output:

```
[1]
[1 2]

Program exited.
```

In the last of the three examples, we pass the slice to a `modifyElement()` function in which we increment every element by one. This has no impact on the contents of the slice header, so the calling code can see the changes made to the slice without us needing to explicitly return it.

**Example 46 - Slice behaving as a reference type**

```
package main

import "fmt"

func incrementElement(sl []int) {
        for i := range sl {
                sl[i]++
        }
}

func main() {
        sl := []int{1, 2, 3, 4, 5}
        fmt.Println(sl)

        incrementElement(sl)
        fmt.Println(sl)
}

// Go Playground: https://go.dev/play/p/CvKjbcRyOAr
```

Output:

```
[1 2 3 4 5]
[2 3 4 5 6]

Program exited.
```

Understanding the behaviour of slices is helpful in spotting and avoiding certain types of bugs. In practice, consider treating slices exclusively as values which must be passed to, and returned from functions. This results in clearer code and no hidden side-effects. Should memory be identified as a constraint and copies of a slice are deemed expensive, either because of their size or type, then you may consider treating it as a reference type, but clearly comment the decision in your code.

### 7.3.3.1 Working with slices

Elements in a slice are accessed by index in exactly the same way as arrays. As with arrays, we can also obtain subsets of array elements using the *:* operator.

Let's look at few examples obtaining subsets of a slice in this next example. We refer to this operation as *reslicing*.

**Example 47 - Reslicing & working with slices**

```
package main

import "fmt"

func main() {
        sl := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

        fmt.Println("Index 0:", sl[0])
        fmt.Println("Elements Index 1 - 5 (not including 5):", sl[1:5])
        fmt.Println("Elements Index 5 - end:", sl[5:])
        fmt.Println("Elements Index 0 - 3 (not including 3):", sl[0:3])
}

// Go Playground: https://go.dev/play/p/Aiw7WhYGePH
```

Output:

```
First element: 1
Elements 1 - 5 (not including 5): [2 3 4 5]
Elements 5 - end: [6 7 8 9 10]
Elements 0 - 3 (not including 3): [1 2 3]

Program exited.
```

Take care when working with subsets of slices. What may appear to be a copy or a new slice value, may infact be pointing at the same back array of data as the original slice. It is possible to mutate elements in the original by mutating elements in what you may believe to be a copy, when it is not.

This example below shows how an operation on variable `sl2` to change an element to 20, perhaps counter-intuitively also amends the element in the original slice `sl`, because the same backing array is used by both slice variables.

**Example 48 - Slices of slices**

```
package main

import "fmt"

func main() {
        sl := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
        sl2 := sl[1:5]

        fmt.Printf("len sl: %d , cap sl: %d\n", len(sl), cap(sl))
        fmt.Printf("len sl2: %d , cap sl2: %d\n", len(sl2), cap(sl2))

        sl2[2] = 20
        fmt.Println("Slice sl2:", sl2)
        fmt.Println("Slice sl:", sl)
}

// Go Playground: https://go.dev/play/p/NAWtoteoj5x
```

Output:

```
len sl: 10, cap sl: 10
len sl2: 4, cap sl2: 9
Slice sl2: [2 3 20 5]
Slice sl: [1 2 3 20 5 6 7 8 9 10]

Program exited.
```

If you want a new slice which you can safely edit and append to without risk of mutating the original, you can unlink it by explicitly taking a copy. We cover the built-in `copy()` function shortly.

### 7.3.3.2 Append

We've seen the `append` command, which we used for adding additional elements to a slice. We should mention that append function is *variadic*, meaning it can accept any number of trailing arguments.

This example shows the different syntax you may consider when using `append`, to add a single element or multiple new elements:

**Example 49 - Using append**

```
package main

import "fmt"

func main() {
        sl := []int{1, 2}
        sl2 := []int{7, 8, 9}

        sl = append(sl, 3)       // add single element
        sl = append(sl, 4, 5, 6) // add many new elements
        sl = append(sl, sl2...)  // append all of sl2 using the spread operator

        fmt.Println(sl)
}

// Go Playground: https://go.dev/play/p/WUPIBwbVCEL
```

Output:

```
[1 2 3 4 5 6 7 8 9]

Program exited.
```

Often we'll combine both the mechanics of *append* and *reslicing* when manipulating slices.

For example, below we create a new slice with the element at the specified by index removed, whilst maintaining the ordering:

**Example 50 - Reslicing to remove a specified element from a slice**

```
package main

import "fmt"

func removeAtIndex(sl []int, index int) []int {
        return append(sl[:index], sl[index+1:]...)
}

func main() {
        sl := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
        fmt.Println(sl)
        sl = removeAtIndex(sl, 9)
```

```
        fmt.Println(sl)
}

// Go Playground: https://go.dev/play/p/e_IrhpQHBXE
```

Output:

```
[1 2 3 4 5 6 7 8 9 10]
[1 2 3 4 5 6 7 8 9]

Program exited.
```

#### 7.3.3.3 Copy

We can use the `copy()` command to make a copy of an entire slice, or a subset of its elements. With *copy*, the elements are copied to a new backing array and we're able to mutate elements in the copy, **without** impacting the source slice.

Let's repeat Example 48 to demonstrate, but, instead of taking a shallow copy as we did in that example, we'll use the `copy()` command. This time the change we make to `sl2` has no side-effects, the source slice remains unchanged.

**Example 51 - Using copy to create a slice with new backing array**

```
package main

import "fmt"

func main() {
        sl := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
        sl2 := make([]int, 4)

        // copy parameter order is the destination then the source
        copy(sl2, sl[1:5])

        sl2[2] = 20
        fmt.Println("Slice sl2:", sl2)
        fmt.Println("Slice sl:", sl)
}

// Go Playground: https://go.dev/play/p/YHVH-gE9lYW
```

Output:

```
Slice sl2: [2 3 20 5]
Slice sl: [1 2 3 4 5 6 7 8 9 10]

Program exited.
```

#### 7.3.3.4 Resizing

To conclude our discussion of slices, we're going to look at what happens when Go resizes a slice so that more data can be added.

We've said already that slices simplify matters when working with data of unknown length. If a slice has no room i.e. the backing array has no capacity, Go will handle creating additional capacity for us. It will copy the backing array to a new array with some extra capacity, pointing the slice header at the new backing array, and the old array will be garbage collected.

Like many conveniences, we won't often think too much about what is happening in the background on the machine, but we should, because it can come with a cost.

Specifically, the new capacity of the backing array is not under our control. Go decides how much extra to provide in the new backing array based on an algorithm.

Now hands up, I'm not exactly sure what the algorithm does. I *thought* that at some level of current capacity it doubled each time, then started to tail off, until eventually 25% additional capacity was allocated each time.

And, that rule mostly holds in my testing. At about the 1000 element mark, the throttling starts, but, in the example below clearly, there is more than 100% of new capacity created initially at levels below 1000 elements.

**Example 52 - New Slice backing array with additional capacity**

```
package main

import (
        "fmt"
)

func main() {
        // amend this start capacity to see the impact
        cp := 100
        sl := make([]int, 0, cp)
        els := 1000
        addr := fmt.Sprintf("%p", sl)

        for i := 0; i < els; i++ {
                sl = append(sl, i)
                if fmt.Sprintf("%p", sl) != addr {
```

```
17
18                          // when we detect address change inspect
19                          extra := (float64(cap(sl)) - float64(cp)) / float64(cp) * 100
20                          fmt.Printf("length: %d, capacity: %d, address: %p. Additional capacity: %0.
21 ", len(sl), cap(sl), sl, extra)
22                          cp = cap(sl)
23                          addr = fmt.Sprintf("%p", sl)
24                  }
25          }
26
27          overCap := (float64(cap(sl)) - float64(els)) / float64(els) * 100
28          fmt.Printf("Backing array over capacity: %0.2f%%", overCap)
29  }
30
31  // Go Playground: https://go.dev/play/p/JMbFUxzyNzK
```

Output:

```
1  length: 101, capacity: 224, address: 0xc000102000. Additional capacity: 124.00%
2  length: 225, capacity: 512, address: 0xc00010c000. Additional capacity: 128.57%
3  length: 513, capacity: 848, address: 0xc000112000. Additional capacity: 65.62%
4  length: 849, capacity: 1280, address: 0xc00011e000. Additional capacity: 50.94%
5  Backing array over capacity: 28.00%
6
7  Program exited.
```

In the above example, we set initial capacity and then incrementally use up the capacity by appending to the slice 1000 times.

We detect when the runtime creates a new backing array by observing when the memory address changes, at which point we output length and capacity, plus calculate a percentage increase.

On the Go Playground, change the `els` variable to a value of 50000 and run it again. You'll see the resizing settle to about 25% additional capacity every time a new backing array is created.

But, we're getting off the point a bit. It's not so much how the algorithm works that concerns us here. No, the point is that when using slices, we can end up using more memory than we need because of this automatic resizing process.

In the previous example, we created capacity for 1280 integer values in memory, but we have only used 1000 elements of that. So we've reserved 28% more memory than we need. Now, these are just integers, so that's only 2,240 bytes, but if the slice was a slice of structs, that 28% extra could be very significant in memory.

To wrap up, **if** memory use is identified as an issue in our program, we might consider how we optimise our slice usage.

With an operation such as the above, we know how much capacity is needed ahead of time, so could create the slice with the exact capacity we require or we could use an array instead.

Both approaches would reserve only the memory required for the data.

Where we don't know the capacity we need exactly, but can reliably estimate it, we may be able to reduce memory use by testing different initial slice capacities and their impact when resizing takes place.

### 7.3.4 Channel

Channels are a mechanism for communicating between asynchronous code. We'll cover them in detail when we talk about Concurrency, here we're going to briefly cover what they are, and how we create and use them, so we can demonstrate their *reference-type* characteristics.

Channels are created using the `make()` built-in function that we've used several times already. They can be *unbuffered* or *buffered* and are typed - each channel can send and receive values of a specific type.

In the sample program below, we have an unbuffered channel of *int* type which we share between two independent functions which use the channel to communicate.

**Example 53 - Send and receive on unbuffered channel**

```
1  package main
2
3  import "fmt"
4
5  func send(ch chan int) {
6          for i := 0; i < 100; i++ {
7                  ch <- i
8          }
9  }
10
11 func read(ch chan int) {
12         for msg := range ch {
13                 fmt.Println(msg)
14         }
15 }
16
17 func main() {
18         ch := make(chan int)
19         fmt.Println(ch)
20         go read(ch)
21         send(ch)
22 }
```

```
23
24 // Go Playground: https://go.dev/play/p/f8M2OFSLnJA
```

In the above example, notice when we print the channel at line 19, the output is a memory address. When we create a channel using `make()` we get a pointer back automatically. So, we don't need to pass by reference when we share the channel in our program, we're already sharing the address.

A good rule of thumb is that if you create a variable using `make()` it is a reference type, and what you store is a memory address so there is no need to obtain a reference - which would be a pointer to a pointer if you think about it!

To summarise, in this section we've explored how some types behave as if they are pointers without us explicitly needing to pass them by reference. Though the reasons differ slightly, types like *map*, *slice* and *channel* can be shared as values because the variable is already a reference to the backing data.

## 7.4 Interface types

We're only going to touch on interfaces in this section. In *Chapter 9 - Digging deeper*, we'll go much further.

So what is the *interface* type?

An interface represents behaviour, what something can do. Behaviour is defined by one or more *receiver* function signatures. These are functions with no implementation, only the argument types and return types are defined.

Notice we said *what something can do* rather than *what something is*? The distinction is important.

In Go, interfaces are implemented implicitly. You won't see the *implements* keyword anywhere. Instead, in Go we use a principle known as *duck typing*.

If something can do what a duck does i.e. walk like a duck and quack lack a duck, then, as far as Go is concerned, it is a duck. Go doesn't care if it is a real duck, just that it can do duck stuff.

This point needs reinforcing. We could have a dog type, but if the dog type has the *walk* and *quack* behaviours, that dog is as good as a duck.

Additionally, Go has an empty interface type. This is an interface that defines no behaviour. You will often see it represented by `interface{}` syntax and, since every type provides at least no behaviours, the empty interface is implemented by every single type!

Personally, I struggled to grasp the significance of this initially, I recall it took me some time. Put another way, whereever you see `interface{}` you can substitute in *any* built-in or custom type. Used as argument in a function or receiver it means that for that parameter, any *type* is acceptable.

Coincidentally, in Go 1.18, as a part of the new generics implementation we can now use an alias for `interface{}` with the `Any` identifier. With *Any* the intent is much more obvious and I think I'd have grasped the empty interface far quicker with *Any*.

To demonstrate a simple but typical use of interfaces we'll look at the standard library's *Stringer* interface.

```
1  type Stringer interface {
2          String() string
3  }
```

*Stringer* specifies just one behaviour, the ability to provide a custom string representation of itself. It uses a single receiver `String()` which takes no arguments but returns a value of type *string*.

For a type to implement the *Stringer* interface, it only needs to implement the `string` receiver. But, so what?

Consider the following example. The `Println()` statement on line 19, takes a *slice* and *map* value and outputs them to Stdout.

For each printed variable we get the value with helpful formatting which provides context about its type.

**Example 54 - The Stringer interface**

```
1  package main
2
3  import "fmt"
4
5  func main() {
6          sl := []int{1, 22, 3}
7          mp := map[string]string{"name": "Joe Blogs"}
8
9          fmt.Println(sl, mp)
10 }
11
12 // Go Playground: https://go.dev/play/p/xl1phIcBej1
```

Output:

```
1  [1 22 3] map[name:Joe Blogs]
2
3  Program exited.
```

Under the hood, although `fmt.Println()` is a variadic function which accepts multiple empty interface types[9], the logic which supports the function will output each variable using the *stringer* interface **if** it is implemented on the type. It does this by calling each type's `String()` receiver which returns a string to print.

Both the built in *slice*[10] and *map*[11] types implement the *stringer* interface. They both have a `String()` receiver bound to them.

Ultimately, the logic behind both implementations results in a type check. If the type is a *slice* the value is wrapped in square brackets like so `[]` and if the type is a *map* it is prefixed with `map[` and suffixed with `]`.

You can inspect that logic in this function[12].

Before we leave interfaces for the moment, let's create a custom struct and implement the *Stringer* interface on it.

**Example 55 - Implementing Stringer on a custom struct type**

```
1  package main
2
3  import "fmt"
4
5  type person struct {
6          name string
7          age  int
8  }
9
10 func (p person) String() string {
11         return fmt.Sprintf("Hey there I'm %v and I'm %v years young!", p.name, p.age)
12 }
13
14 func main() {
15         st := person{
16                 "Joe Blogs",
17                 25}
18         fmt.Println(st)
19 }
20
21 // Go Playground: https://go.dev/play/p/0a4sBfwsx78
```

Output:

[9] https://cs.opensource.google/go/go/+/refs/tags/go1.19.3:src/fmt/print.go;drc=668041ef66ddafffccf1863e6180b83ea1ad30c9;l=293
[10] https://cs.opensource.google/go/go/+/master:src/go/types/slice.go;drc=521828091c73e2af67bc2210b7c94cc54076f17b;l=19
[11] https://cs.opensource.google/go/go/+/master:src/go/types/map.go;drc=521828091c73e2af67bc2210b7c94cc54076f17b;l=24
[12] https://cs.opensource.google/go/go/+/master:src/go/types/typestring.go;drc=1fbfc2f6eba6cc88a8fb0ae8e83afe80553f65df;l=113

```
Hey there I'm Joe Blogs and I'm 25 years young!

Program exited.
```

Try deleting lines 10 - 12 and then run the example again. This time the default *stringer* implementation is used. Since the *struct* type also implements the interface, the output is simplified and begins with `struct{` and ends with `}` as shown here[13].

If a type implements the *Stringer* interface, the printed output of a variable of that type is likely a custom representation of the value stored by that variable, and it sometimes may not be what you expect.

## 7.5 Creating custom types

So far we've looked at Go's built-in types. We've also used named custom struct types.

We've said that types and the statically-typed nature of Go enable compile-time checking which makes our programs safer at runtime.

But, built-in types alone won't protect us in all circumstances. Consider the example below.

**Example 56 - Simple transposition error**

```
package main

import "fmt"

func divider(d, n int) float64 {
        return float64(n) / float64(d)
}

func main() {
        var n = 1
        var d = 4

        result := divider(n, d)
        fmt.Println(result) // we think we will get 0.25
}

// Go Playground: https://go.dev/play/p/XhTIz_A-dxd
```

Output:

[13] https://cs.opensource.google/go/go/+/master:src/go/types/typestring.go;drc=1fbfc2f6eba6cc88a8fb0ae8e83afe80553f65df;l=147

```
4

Program exited.
```

We have a basic function which expects two integers, a denominator and numerator. It performs a division calculation and returns the result.

But, when we call the function we make a simple transposition error when passing the numerator and denominator.

We got the parameters backwards. As both are integers, the compiler has no issue with our code, which builds and runs fine, but, instead of the `0.25` result we expected, we got a result of `4`, which is incorrect. This kind of mistake is very easy to make and can be hard to debug.

But, we can help ourselves a little here. We can employ Go's type system to ensure this kind of error is something the Go compiler will flag during compilation. And we do this by creating custom types.

Custom types are based on built-in types but they have their own type identity. A custom type which can hold a *string* value is not the same as the *string* type, for example. It's possible to convert it to a string, but unless it is converted it cannot be used as a *string*.

Let's use custom types to rewrite the code in Example 56 and see if we can't catch that transposition error when the code is compiled.

**Example 57 - Type safety using custom types**

```
package main

import "fmt"

type numer int
type denom int

func divider(d denom, n numer) float64 {
        return float64(n) / float64(d)
}

func main() {
        var n numer = 1
        var d denom = 4

        result := divider(n, d)
        fmt.Println(result)
}

// Go Playground: https://go.dev/play/p/6UQcR54_wYX
```

Output:

```
./prog.go:16:20: cannot use n (variable of type numer) as type denom in argument to \
divider
./prog.go:16:23: cannot use d (variable of type denom) as type numer in argument to \
divider

Go build failed.
```

What did we change?

First, we created two custom types `numer` and `denom` both based on the `int` type so they can hold integers but are *not* of type `int`. Each is distinct.

Next, we declared `n` and `d` with their respective types and finally, we modified the function signature so that it accepts one of each type and not two integers.

When we attempt to run the program we get a compilation error which indicates we are passing the wrong types and, upon inspection, it is clear why - we are using the variables the wrong way around.

If we swap the arguments on line 16, so that they are passed correctly, the program builds, runs and outputs the expected result.

So, developing using custom types is a habit to be encouraged. By defining custom types we're able to add an extra layer of type checking, and thus, type safety. The more safeguards we have in place, the more errors we can catch at compile time and the safer our programs will be when they run in production.

Finally, the last thing to say about custom types, which we briefly mentioned in Chapter 1, is that any custom type can have receivers bound to it. If you recall, this is a key difference between Go receivers and OOP methods, the latter forming part of a class structure.

The benefit is twofold. First, it means we can create receivers which use the type's data directly, we don't need to pass it via an argument in the receiver signature.

Secondly, we can use our custom types anywhere a particular interface is required, provided we can implement the interface for the type, which entails adding the required behaviour to our custom type.

So let's wrap up this section by quickly adding a single `String()` receiver on the *numer* type, thereby implicitly implementing the `Stringer` interface.

**Example 58 - Implementing the Stringer interface on a custom type**

```
package main

import "fmt"

type numer int

func (n numer) String() string {
        return fmt.Sprintf("Numerator is set to %d", n)
}

func main() {
        var n numer = 1
        fmt.Println(n)
}

// Go Playground: https://go.dev/play/p/RSGm_FPhf6L
```

Output:

```
Numerator is set to 1

Program exited.
```

When we run the program we get nicely formatted output with additional context about the *numer* typed variable, provided by the `String()` receiver.

# 7.6 Converting between types

To complete our discussion of data types, for now, we're going to look at how we convert values from one type to another.

## 7.6.1 Type conversion

Go does not support *type casting*, only type conversion. What's the difference?

Type casting can often be used with non-compatible data types as well as compatible types, while *type conversion* is restricted to compatible data types, and this is the case with Go. You can't convert a *string* to an *int* using type conversion, for example.

In other languages *type conversion* is often implicit, but in Go because of its strong type system the conversion is explicit. This requires we state what we want the compiler to convert the value to

using a wrapper function. The compiler will perform the conversion if the types are compatible, or fail to build with an error if they are not.

The example below demonstrates several type conversions between compatible data types. Generally, type conversion between number types is fine. As is type conversion back and forth between *[]byte* and *string*.

The commented code also shows type conversions between incompatible data types, try removing the comments and running the program again. It won't build, instead, we will receive compilation errors.

**Example 59 - Type conversion examples**

```
package main

import "fmt"

func main() {
        var myInt = 1
        var myFloat = 0.0
        var myString = "Hello World"
        var myBytes = []byte{240, 159, 154, 128}

        fmt.Printf("Before conversion, myInt is: %T, myFloat is: %T, myString is: %T, myByt\
es is: %T\n",
                myInt, myFloat, myString, myBytes)

        convertedInt := float64(myInt)
        convertedFloat := int(myFloat)
        convertedBytes := string(myBytes)
        convertedString := []byte(myString)

        fmt.Printf("After conversion, myInt is: %T, myFloat is: %T, myString is: %T, myByte\
s is: %T\n",
                convertedInt, convertedFloat, convertedString, convertedBytes)

        // incompatible data types
        //convertedFloat2 := string(myFloat)
        //convertedBool := int(true)
        //convertedString2 := int(myString)
}

// Go Playground: https://go.dev/play/p/FOkyY6XJp1q
```

Output:

```
Before conversion, myInt is: int, myFloat is: float64, myString is: string, myBytes \
is: []uint8
After conversion, myInt is: float64, myFloat is: int, myString is: []uint8, myBytes \
is: string

Program exited.
```

Note a simple gotcha. The Go compiler will **not** error if you try to convert an *int* to a *string* using type conversion, but you will not get a string representation of the integer. The conversion to string will take place, but the integer is treated as a Unicode codepoint. In the following example we don't get a string type of value "128640", we get the Unicode rocket character.

**Example 60 - This won't print what you expect**

```
package main

import "fmt"

func main() {
        var myInt = 128640
        convertedInt := string(myInt)

        fmt.Printf("The int is now a string: '%s', of type '%T'\n", convertedInt, converted\
Int)
}

// Go Playground: https://go.dev/play/p/GHVwfyD-mlH
```

Output:

```
The int is now a string: '🚀', of type 'string'

Program exited.
```

## 7.6.2 Other conversion mechanisms

So, *type conversion* won't allow us to convert between incompatible data types and for some use-cases, even where it operates on compatible data types, the result of conversion may not be what was expected.

Fortunately, we have other options, particularly when converting to and from, strings.

A very simple way to solve the *problem* above, where the integer is converted to a string, not as is, but to a Unicode character, is to use the `fmt` package instead.

**Example 61 - fmt.Sprintf to the rescue**

```
package main

import "fmt"

func main() {
        var myInt = 128640
        convertedInt := fmt.Sprintf("%d", myInt)

        fmt.Printf("The int is now a string: '%s', of type '%T'\n", convertedInt, converted\
Int)
}

// Go Playground: https://go.dev/play/p/b6APk-xVK9X
```

Output:

```
The int is now a string: '128640', of type 'string'

Program exited.
```

We can also perform conversions between strings and what would otherwise be incompatible data types using the `strconv` package from the standard library - a package which can assist us with all manner of string and numeric conversions.

Exploring the `strconv` package[14] is left as an exercise for the reader, but the example below shows three common use cases.

**Example 62 - Package *strconv* examples**

```
package main

import (
        "fmt"
        "strconv"
)

func main() {
        myInt := 10
        myIntString := "10"
        myFloatString := "3.1415926535"

```

[14] https://pkg.go.dev/strconv

```
13          // int to string
14          resStr := strconv.Itoa(myInt)
15          fmt.Printf("resStr is '%T, of '%v'\n", resStr, resStr)
16
17          // string to int
18          if resInt, err := strconv.Atoi(myIntString); err == nil {
19                  fmt.Printf("resInt is '%T', '%v'\n", resInt, resInt)
20          }
21
22          // string to float
23          if resFloat, err := strconv.ParseFloat(myFloatString, 64); err == nil {
24                  fmt.Printf("resFloat is '%T', '%v'\n", resFloat, resFloat)
25          }
26  }
27
28  // Go Playground: https://go.dev/play/p/5kW2FyxBGgU
```

Output:

```
1  resStr is 'string, of '10'
2  resInt is 'int', '10'
3  resFloat is 'float64', '3.1415926535'
4
5  Program exited.
```

Finally, to conclude this section, what should we do if *type conversion* won't work since the types are incompatible, and there is no built-in helper function that does what we need, for example, to convert a *bool* to an *int*?

In these rare circumstances, we as developers get an opportunity to step up and devise a conversion helper of our own, rather than rely on the Go standard library for all of our needs ;)

In the example below we create a simple `boolToI()` helper function to perform just such a conversion.

**Example 63 - A custom BoolToI helper function**

```
package main

import "fmt"

func boolToI(b bool) int {
        var bToI int
        if b {
                bToI = 1
        }
        return bToI
}

func main() {
        myBool := true
        myBool2 := false

        boolAsInt := boolToI(myBool)
        boolAsInt2 := boolToI(myBool2)

        fmt.Printf("boolAsInt: %d, boolAsInt2: %d\n", boolAsInt, boolAsInt2)
}

// Go Playground: https://go.dev/play/p/gseJ2vzyDSc
```

Output:

```
boolAsInt: 1, boolAsInt2: 0

Program exited.
```

# Chapter 8 - Managing program flow

Now that we've got many of the fundamentals in place, let's look at how we bring it all to life. In this section, we'll cover control structures and error handling: the glue of our program.

## 8.1 Control structures

Programs are dumb. They need to be told how to respond to every input and outcome. We call this *flow control* and we may employ three forms of *control structure* logic in our programs to facilitate this: *sequence*, *selection* and, *iteration*.

*Sequence* logic is linear. Statements are executed one after the other provided expectations are met.

*Selection* logic concerns itself with the conditional flow - what to do when one or more, of several possible outcomes, is satisfied.

*Iteration* logic covers the conditions under which a section of code should be repeatedly run, and of course, when that repetition should finish.

Let's look at how we implement flow control in Go.

### 8.1.1 Sequence logic

With *sequential* flow, execution continues statement by statement until a result is obtained and/or the function returns. Statements are ordered, such that the next builds on the last to achieve a result. Unless there is an error, panic or control signal to terminate, there is nothing to interrupt the execution of the logic.

Though we can call blocks of code asynchronously via a goroutine, the operations performed by such a goroutine would also execute in the same manner: top to bottom.

We may ocassionally see the `goto` statement used, which allows for an unconditional jump to a label identifier **within** the same function.

While it is unconditional, `goto` may be used to act on the result of some expectation or condition.

In the example below we use `goto` to implement an endless loop which cycles every three seconds.

**Example 64 - Using goto to restart function execution**

```
package main

import (
        "fmt"
        "time"
)

func main() {

START:

        fmt.Println("Timer running")
        time.Sleep(3 * time.Second)
        goto START
}

// Go Playground: https://go.dev/play/p/5dsPU41vPQ7
```

Output:

```
Timer running
Timer running
Timer running
...
```

This example is included only for completeness. The use of `goto` is not common in Go, nor recommended. It can make code harder to read and reason about. We have alternative control structures we can use to achieve the same as `goto`.

We should also mention *defer* here since it is another exception to the normal sequential flow. The `defer` keyword is used to link associated logic to aid readability and help prevent bugs.

For instance, consider the operations of opening, reading and then closing a file. Between the opening and closing of the file, there could be many lines of code to process the file contents, the result being that the opening and close operations could become somewhat disassociated.

Failing to close the file would create a bug, but in long sequences of code, the omission of the close operation may be difficult to spot.

The `defer` keyword solves the problem by allowing us to position the open and close operations near one another. Any deferred statements are run before the function returns, if there are multiple deferred statements they are run on a LIFO basis.

```
    // open file
    src, err := os.Open(filename)
    if err != nil {
        return
    }

    // defer closing file
    defer src.Close()

    // file processing operations
    ...
```

## 8.1.2 Selection logic

Go provides similar conditional statements to those found in other languages:

- if/else/elseif
- switch/case/default

### 8.1.2.1 If/else/elseif

When using if/else conditional logic, an established best practice is to minimise our use of *else* statements. We can often achieve this by making the *else* condition the default case.

Our motivation, once again, is improved readablity. For it is easier to understand how a program should run by reviewing code at the same level of indentation. Stepping into, and then out of, conditional code makes reasoning and comprehension more difficult.

Example 65 shows a fairly simple function written in a perfectly acceptable manner using *if* and *else* conditional statements.

In the example, we pass two numbers as arguments and the function tells us if the first number is even, and less than the second.

**Example 65 - Simple function with conditional if/else logic**

```
package main

import "fmt"

func isEvenLessThan(n int, c int) (bool, bool) {
        if n%2 == 0 {
                if n < c {
                        return true, true
                } else {
```

```
                        return true, false
                }
        } else {
                if n < c {
                        return false, true
                } else {
                        return false, false
                }
        }
}

func main() {
        fmt.Println(isEvenLessThan(2, 10))
}

// Go Playground: https://go.dev/play/p/hUPNR_4gRy_6
```

Output:

```
true true

Program exited.
```

The function `isEvenLessThan()` makes use of multiple return values, which we've not seen so far. We'll cover this attribute of functions and receivers in Chapter 9 - Digging deeper.

In Example 66, next, we rewrite the function logic in such as way that we can dispense with the *else* statements. By making use of the fact that variables are initialised to their *nil* value, we can simplify the conditional logic required to set the function return values. The result is less code, less indentation and, improved readability.

**Example 66 - Eliminating the *else* statements**

```
package main

import "fmt"

func isEvenLessThan(n int, c int) (bool, bool) {
        var even, less bool // initialised to false

        if n%2 == 0 {
                even = true
```

```
        }
        if n < c {
                less = true
        }

        return even, less
}

func main() {
        fmt.Println(isEvenLessThan(2, 10))
}

// Go Playground: https://go.dev/play/p/lJm4fJ3REPa
```

Output:

```
true true

Program exited.
```

Of course, it won't always be possible to remove all usage of *else*. In situations where we need to use it, we should try to minimise its footprint by writing the code in the *else* body as concisely as possible.

Go also has an *elseif* conditional statement, which may help reduce our use of nested *if* and *else* statements.

### 8.1.2.2 Short-form if statement

Where possible, it's preferable to use a short-form *if* statement. Indeed it's something we will see frequently when reviewing other users' code.

It helps us reduce the verbosity of our syntax, which is particularly useful for operations such as error checking that we perform repeatedly.

The syntax appears slightly unusual initially, but it is worth getting used to. Once you are familiar with the short-form syntax it does not detract from clarity.

In the example below, we use a short-form *if* statement to print to Stdout **only** if the result is even. We discard the second return value, which we do not need, by using the *blank identifier*.

**Example 67 - Short-form if statement**

```
package main

import "fmt"

// function omitted for brevity

func main() {
        if even, _ := isEvenLessThan(2, 10); even {
                fmt.Println("Result was even")
        }

    //fmt.Println(even)
}

// Go Playground: https://go.dev/play/p/qmoU-gAtzCw
```

Output:

```
Result was even

Program exited.
```

Note that in this short-form if statement, the variable `even` is scoped to the *if* statement body. It exists only while that block is executed and not beyond it. Try uncommenting the last line to print `even` and running the code again.

Note also, that if the variable `even` was declared in the parent function scope too, it would not be the value used in this *if* block. So take care with variable shadowing. Replacing `:=` with `=` would ensure that no new variables are declared, and the `even` variable used in the *if* statement body was the same as declared in the parent scope.

### 8.1.2.3 Switch/case/default

An alternative to using an *if* condition with multiple *elseif* statements, is *switch*. When using *switch* we can implement all the conditional logic of *if/else/elseif*, and more.

In place of *elseif* statements, we can implement *case* statements. We can also use *default* to specify an equivalent to *else* if none of the *case* statement conditions match.

Go's implementation of *switch* is slightly different to other languages. It only runs the matched *case* (and not those that follow) and then implicitly breaks out of the switch statement although we can use the *fallthrough* keyword to emulate how other languages implement *switch*.

Another distinction is that in Go *switch* can operate on variables and constants of any type, and not only integers.

We show a couple of *switch* examples next. First, an example with a *default* statement which operates on a string variable. This executes if none of the case statements are matched and executed.

**Example 68 - Simple switch with default**

```
package main

import (
        "fmt"
)

func main() {
        myName := "Joe Blogs"

        switch myName {
        case "Joe Blogs":
                fmt.Println("Hi Joe!")
        case "Dave Blogs":
                fmt.Println("Hi Dave!")
        default:
                fmt.Println("Hi there!")
        }
}

// Go Playground: https://go.dev/play/p/3FiTf9lGEDd
```

Output:

```
Hi Joe!

Program exited.
```

Next, we show an example of an *expressionless* switch statement which allows us to write more complex *case* conditions than would otherwise be possible.

**Example 69 - Expressionless switch statement**

```
package main

import (
        "fmt"
)

func main() {
        num := 12

        switch {
        case num >= 0 && num <= 10:
                fmt.Println("Between 0 and 10")
        case num >= 10:
                fmt.Println("Greater than 10")
        }
}

// Go Playground: https://go.dev/play/p/cx8jV4QXHRI
```

Output:

```
Greater than 10

Program exited.
```

In this example, we use the short-form notation and set multiple match tests per *case* statement.

**Example 70 - Short-form switch with multiple match tests**

```
package main

import (
        "fmt"
)

func main() {
        switch letter := "z"; letter {
        case "a", "e", "i", "o", "u":
                fmt.Println("letter was a vowel")
        default:
                fmt.Println("letter was not a vowel")
        }
```

```
}

// Go Playground: https://go.dev/play/p/DjPtZbdigfm
```

Output:

```
letter was not a vowel

Program exited.
```

The final example illustrates the use of *fallthrough*, which will execute the body of the next case and then exit the switch statement.

**Example 71 - Fallthrough to execute the next case**

```
package main

import "fmt"

func main() {
        day := "Mon"

        switch {
        case day == "Mon":
                fmt.Println("Monday")
                fallthrough
        case day == "Tue":
                fmt.Println("Tuesday")
        case day == "Wed":
                fmt.Println("Wednesday")
        }
}

// Go Playground: https://go.dev/play/p/hw1WLkjYPUb
```

Output:

```
Monday
Tuesday

Program exited.
```

It is an often-held misconception that *fallthrough* moves to the next case statement and then checks for a match, executing only if the case test is satisfied. Though intuitively this seems sensible, it is not the case - the body of the next case statement is executed whether or not there is a match.

## 8.1.3 Iteration logic

Though many languages offer several constructs for implementing different types of loops, in Go we implement all loops using only `for` loops.

A single `for` keyword keeps matters simple and we're able to emulate all of the looping mechanisms like `do while`, `while` and `for each`, if we know how to construct the `for` loop.

So, let's look at all the different loop implementations we can create.

Let's not be confused. Though we label the loops as their equivalents in other languages, we're using only `for` to create them. That's the only looping construct we have in Go!

### 8.1.3.1 Infinite

Infinite loops are achieved using the most basic `for` loop syntax.

**Example 72 - Infinite loop with for**

```
1  package main
2
3  import (
4          "fmt"
5  )
6
7  func main() {
8          // this will timeout on the playground
9          for {
10                 fmt.Println("infinite loop")
11         }
12 }
13
14 // Go Playground: https://go.dev/play/p/ZBcHX0L11Qq
```

Output:

```
infinite loop
infinite loop
infinite loop
...
```

### 8.1.3.2 Three Component

Common to most languages is the three-component loop which uses an *init statement*, a *loop condition* and a *post-loop statement* to specify the loop behaviour.

The three-component `for` loop is implemented in the same way in Go.

**Example 73 - Three component loop with for**

```
package main

import (
        "fmt"
)

func main() {
        for i := 0; i < 5; i++ {
                fmt.Println("iteration", i+1)
        }
}

// Go Playground: https://go.dev/play/p/IFE1-HGEGkQ
```

Output:

```
iteration 1
iteration 2
iteration 3
iteration 4
iteration 5

Program exited.
```

### 8.1.3.3 While equivalent

*While* loops execute while a certain condition exists. In the below example, the condition is that n should be less than or equal to `5`. The condition is checked before execution of a loop iteration. Notice how `n` finishes and prints at a value of `6`.

**Example 74 - While equivalent using for**

```
package main

import (
        "fmt"
)

func main() {
        n := 1
        for n <= 5 {
                fmt.Println("Iteration", n)
                n++
        }
        fmt.Printf("n finished at %d\n", n)
}

// Go Playground: https://go.dev/play/p/oC25p-YNNBx
```

Output:

```
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
n finished at 6

Program exited.
```

### 8.1.3.4 Do while equivalent

A `do while` style loop is very similar to `while` but the condition check is made after the loop iteration. In Go, using `for` alone the equivalent syntax would be like in the below example. Notice that `n` finishes at a value of `5`.

**Example 75 - Do while equivalent with for**

```
package main

import "fmt"

func main() {
        n := 1
        for ok := true; ok; ok = n != 5 {
                fmt.Println("Iteration", n)
                n++
        }
        fmt.Printf("n finished at %d\n", n)
}

// Go Playground: https://go.dev/play/p/SZUMlRfW3Fn
```

Output:

```
Iteration 1
Iteration 2
Iteration 3
Iteration 4
n finished at 5

Program exited.
```

### 8.1.3.5 For Each

*For each* type implementations use the *range* keyword. We've seen *range* in many of the examples to date, and we can use it to iterate over collections such as *arrays*, *slices* and *maps* while getting convenient access to the data contained in the collection.

**Example 76 - For each performed using idiomatic for range**

```
package main

import "fmt"

func main() {
        sl := []int{1, 2}
        mp := map[string]string{"key1": "value1", "key2": "value2"}

        // slice/array
```

```
        for index, value := range sl {
                fmt.Printf("index: %d, value: %d\n", index, value)
        }

        // map
        for key, value := range mp {
                fmt.Printf("key: %s, value: %s\n", key, value)
        }
}

// Go Playground: https://go.dev/play/p/CVjj4pUtc2c
```

Output:

```
index: 0, value: 1
index: 1, value: 2
key: key1, value: value1
key: key2, value: value2

Program exited.
```

Beware of trying to mutate a slice element within a `for range` loop using the `value` variable. This won't work. The `value` variable is a copy of the slice element created at each iteration of the `for range` loop. Altering `value` will change it for that iteration only, and will not change the element in the slice itself. To mutate the slice element properly we should use the index position of the element in the slice, so using the example above if we wanted to add 1 to each element in the slice we would do the following

```
for index, value := range sl {
        fmt.Printf("index: %d, value: %d\n", index, value)
    sl[index]++ // and not value++
}
```

### 8.1.3.6 Break & Continue

Go implements `break` and `continue` exactly as in C and similar languages. To exit out of the innermost loop and carry on execution we use `break`. To exit a single iteration and have the loop start the next iteration we use `continue`.

In this example we look for a value (3). When found we break out of the loop and print what we found.

**Example 77 - Using break to exit a loop**

```
package main

import "fmt"

func main() {
        sl := []int{1, 2, 3, 4, 5}
        found := 0

        for _, v := range sl {
                if v == 3 {
                        found = v
                        break
                }
        }

        fmt.Println("found:", found)
}

// Go Playground: https://go.dev/play/p/o2vXkdAYNk3
```

Output:

```
found: 3

Program exited.
```

In this example we use `continue` to advance to the next iteration if a number is positive, so that we only print odd numbers.

**Example 78 - Using continue to advance to next loop iteration**

```
package main

import "fmt"

func main() {
        sl := []int{1, 2, 3, 4, 5}

        for _, v := range sl {
                if v%2 == 0 {
                        continue
                }
```

```
                fmt.Printf("%d is an odd number\n", v)
        }
}

// Go Playground: https://go.dev/play/p/qbVGN5S3hLY
```

Output:

```
1 is an odd number
3 is an odd number
5 is an odd number

Program exited.
```

Note `break` is also used with `switch/case` statements and `select`, whereas `continue` is only relevant in `for` loops.

## 8.2 Error handling

Go's error-handling capabilities have earned it a reputation as one of the most reliable languages for production-level applications.

We said right at the beginning when introducing Go that errors are just values. In Go, there are no exceptions and no *try/catch* type operations.

We're able to create error values, and, decide how we handle the error values we receive, in our program flow. Generally, we'll either log the error if execution should continue, or return from the function/receiver with the error if it should not. As a rule, we shouldn't do both.

The default, zero value of an error value is nil. Only non-nil values are considered to be errors. Function signatures should be designed with the error as the last return value, and it is generally expected that when returning a non-nil error value, any other return values should be set to their *nil* or *empty* representation rather than include data. Error values themselves, should usually be written all in lowercase.

In Go, errors are represented by the built-in type *error*. This type is an interface that defines the behavior of an error, which is any value that can describe itself as a string via an `error()` receiver.

```
type error interface {
    Error() string
}
```

For many use cases, when we don't need to create custom error types, we'll simply leverage Go's built-in packages for handling errors, and we'll look at how we use those first.

## 8.2.1 Error helpers

The most commonly used error helper package is `errors`. This package provides a simple way to create and manipulate errors. It contains helpers for creating errors, checking for errors, and formatting errors.

We can create an error using `errors.New(string)` this is factory type function for a built-in custom data struct type, `errorString` which implements the `error` interface.

We can see what is happening under the hood by inspecting the package.

```
type errorString struct {
    s string
}

func (e *errorString) Error() string {
    return e.s
}

func New(text string) error {
    return &errorString{text}
}
```

In the code snippet, `New(string)` creates a pointer of type `errorString`, sets the `s` field of the type and returns `errorString` which is an error because it satisfies the *error* interface.

We can also use `fmt.Errorf()` to generate formatted errors which include data. Ultimately this implementation also wraps `errors.New()`.

### 8.2.1.1 Predefined errors

We can create predefined errors (often referred to as sentinel errors) in our code using either of the above approaches. We can then use the errors package to inspect the type of error returned using `errors.Is()`. This gives us options about how we handle different classes of error perhaps based on severity: can we log and proceed or should we halt and return. Predefined errors are also useful in unit testing when performing assertions.

In Example 79, we use predefined errors to check what kind of error was received.

**Example 79 - Using errors.Is to handle different error values**

```
package main

import (
        "errors"
        "fmt"
)

var ErrScoreLessThanMin = errors.New("score is less than minimum")
var ErrScoreOverMax = errors.New("score is greater than maximum")

func checkRating(score int) (int, error) {
        min := 0
        max := 5

        if score < min {
                return 0, ErrScoreLessThanMin
        }
        if score > max {
                return 0, ErrScoreOverMax
        }

        return score, nil
}

func main() {
        rating, err := checkRating(-1)
        if err != nil {
                switch {
                case errors.Is(err, ErrScoreLessThanMin):
                        fmt.Println("rating score too low")
                case errors.Is(err, ErrScoreOverMax):
                        fmt.Println("rating score too high")
                default:
                        fmt.Printf("unexpected error: %s\n", err)
                }
                return
        }
        fmt.Printf("Rating : %d\n", rating)
}

// Go Playground: https://go.dev/play/p/gb2eikJ9Hp6
```

Output:

```
rating score too low

Program exited.
```

## 8.2.2 Custom error types

Custom error types allow us to convey more specific information about the error.

To create a custom error type, we need to create a new type that implements the error interface. This can be achieved by simply defining a new type that has an `Error() string` receiver.

In the example which follows we use a custom struct type for the custom error, the benefit being it allows us to store additional data about the error if we wish: data that is returned with the error and available to inspect by calling code, if desired.

The `Error() string` receiver is called when the error is converted to a string, which is what the `fmt` package does when it prints an error.

Once we've defined a custom error type, we can use it like any other error type in Go.

**Example 80 - A custom error type**

```
package main

import (
        "fmt"
        "time"
)

type MyError struct {
        // struct fields here, if any
}

func (e MyError) Error() string {
        return fmt.Sprintf("this is custom error generated at %s", time.Now())
}

func dummyFunctionError() error {
        return &MyError{}
}

func main() {
        if err := dummyFunctionError(); err != nil {
                fmt.Println(err)
```

```
        }
}

// Go Playground: https://go.dev/play/p/x7pZDTqda50
```

Output:

```
this is custom error generated at 2009-11-10 23:00:00 +0000 UTC m=+0.000000001

Program exited.
```

In the example, the `dummyFunctionError()` function returns a custom error type. The caller can then check the error and handle it as needed. If additional fields were set on the error type, they could be inspected or logged.

### 8.2.3 Error wrapping

Errors may be wrapped by creating a new error value that includes the original error as part of its message or data.

The `fmt.Errorf()` function formats a new error message that includes the original error value. The `%w` format specifier is used to include the original error in the error message.

Wrapping is useful when trying to understand where an error originated in our code.

```
wrappedErr := fmt.Errorf("error occurred: %w", err)
```

Once the error is wrapped, we can use the new error value like any other error in Go. We can also use `errors.Unwrap()` to unpack a wrapped error message should we need to work with the error values individually, and not as a single value.

Below is an example of wrapping and unwrapping errors in Go. Each *wrap* includes the function name that returned the error. Note, the program has no purpose other than to generate and wrap errors.

**Example 81 - Error wrapping and unwrapping**

```
package main

import (
        "errors"
        "fmt"
)

func call3() error {
        return fmt.Errorf("call3: this is the original error")
}

func call2() error {
        if err := call3(); err != nil {
                return fmt.Errorf("call2: %w", err)
        }
        return nil
}

func call1() error {
        if err := call2(); err != nil {
                return fmt.Errorf("call1: %w", err)
        }
        return nil
}

func main() {
        if err := call1(); err != nil {
                fmt.Println(err) // prints all wrapped errors

                // unwrap the errors and print
                err1 := errors.Unwrap(err)
                err2 := errors.Unwrap(err1)
                fmt.Println(err1)
                fmt.Println(err2)
        }
}

// Go Playground: https://go.dev/play/p/UzPyJIGc1DH
```

Output:

```
call1: call2: call3: this is the original error
call2: call3: this is the original error
call3: this is the original error

Program exited.
```

### 8.2.4 Panic and recover

The *panic* and *recover* functions are used in exceptional situations, such as for a runtime error which can't be handled or following an unexpected event, which makes it unsafe to continue execution of the program in an unknown state.

A *panic* is caused either by a runtime error or an explicit call to the built-in `panic()` function, which is called with a single argument, which is the value that describes the *panic*. This value is typically an error, but it can be any value.

`recover()` is used to recover from a *panic* and continue with the execution of the program.

When `panic()` is called, it stops the normal execution of the code and begins unwinding the call stack. It will call any *deferred* functions and run any clean-up code, such as that for closing open files or releasing resources.

Once the call stack has been completely unwound, the runtime will look for a `recover()` function. If found, it will be called with the value that was passed to the `panic()` function. The `recover()` function will then handle the panic and return a value to continue the normal execution of the program. That return value is the value it was passed via `panic()`.

If a `recover()` function is not implemented the program will exit, with a stack trace for debugging.

Below we show an example of using `panic()` and `recover()`.

**Example 82 - Basic panic and recover**

```
package main

import (
        "errors"
        "fmt"
)

func dummyFunctionError() {
        err := errors.New("dummy unhandlable error")
        if err != nil {
                panic(err)
        }
}

```

```
15 func main() {
16         defer func() {
17                 if err := recover(); err != nil {
18                         fmt.Println("Recovered from panic:", err)
19                 }
20         }()
21
22         dummyFunctionError()
23 }
24
25 // Go Playground: https://go.dev/play/p/9vmgVqKTbU5
```

Output:

```
1 Recovered from panic: dummy unhandlable error
2
3 Program exited.
```

In the above example, the `dummyFunctionError()` function panics if an error occurs, an error which we provide. The `main()` function includes a deferred `recover()` function which is called when the panic occurs. The `recover()` function handles the panic and prints a message to indicate that the panic was recovered from.

The `recover()` function can only be used inside of a deferred function. Using it outside will not affect the panic, which will continue.

Finally, try commenting out or deleting lines 16 to 20 to see what happens if the deferred `recover()` function is not present. You should see the stack trace in the output

```
1 panic: dummy unhandlable error
2
3 goroutine 1 [running]:
4 main.dummyFunctionError(...)
5         /tmp/sandbox1162862492/prog.go:10
6 main.main()
7         /tmp/sandbox1162862492/prog.go:16 +0x49
8
9 Program exited.
```

## 8.3 Logging

To conclude the chapter we will briefly discuss logging. Logging refers to the process of recording events or messages that happen during program flow, usually to provide debug or state information.

### 8.3.1 Log package

As a part of its standard library, Go implements a `log` package which offers several functions for logging messages, such as `Println()`, `Printf()`, and `Fatal()`.

We've already seen the `log` package when we looked at import aliasing in an earlier section but, to recap, it's very simple to implement logging in Go.

```
1  package main
2
3  import "log"
4
5  func main() {
6      log.Println("Hello world!")
7  }
```

When run, the above program prints the message to *Stdout*.

The `log` package provides a simple and convenient way to add logging to our program, and for many use cases, we won't need anything else.

However, there are some limitations with Go's logging implementation when compared with other languages. For example, the `log` package does not support severity levels such as *info*, *warn* and *error*.

### 8.3.2 Custom logger

If we need more than the standard library can offer, we can create a custom *Logger*. This enables us to customize the way log messages are handled. Alternatively, we can use one of the available third-party logging packages, which themselves are implementations of a custom *Logger*.

We might choose to create a custom *Logger* if we need to direct output to a destination other than Stdout - a file for example - or, if we need more control over the formatting of logged message output.

Note that we don't *need* to create a custom *Logger* in order to direct output to a different target. We can instead use the `log` package's `SetOutput()` function to direct logs to any target which implements the `io.Writer` interface.

To create a custom `Logger` we use the `New()` function, which returns a pointer to a `Logger` struct value. We need to pass an `io.writer` for the log target, a prefix `string` and, some flag arguments as part of the setup.

Many of the functions such as `Println()`,`Printf()`, and `Fatal()` are also implemented as receivers on the `Logger` struct, so we can use them on the custom *Logger* too. These might be enough for our needs.

But, we can override these default implementations, as well as create new receivers, by embedding the standard library `Logger` in a custom struct type.

Below is a very basic example of how we'd embed the standard library `Logger` in our logger struct, so by promoting its fields and receivers. We override the default `Println` implementation with our own, allowing us to set a log level and format the output differently.

**Example 83 - Embedding log.Logger to augment its features**

```
package main

import (
        "fmt"
        "log"
        "os"
)

type myWrappedLogger struct {
        *log.Logger
        level string
}

func (ml *myWrappedLogger) Println(level string, v ...any) {
        fmt.Println("Fancy Logger")
        fmt.Println("------------")
        fmt.Printf("Level: %s\n", level)
        ml.Output(2, fmt.Sprintln(v...))
        fmt.Println("------------")
}

func main() {
        mwl := myWrappedLogger{log.New(os.Stdout, "LogPrefix: ", log.LstdFlags), ""}
        mwl.Println("error", "this is the log output")
}

// Go Playground: https://go.dev/play/p/ca17kF91BtL
```

Output:

```
Fancy Logger
------------
Level: error
LogPrefix: 2009/11/10 23:00:00 this is the log output
------------

Program exited.
```

Custom loggers provide a flexible and powerful way to handle logging. We can use them to log messages to a variety of different destinations, and to customize the format of log messages to suit any need.

# Chapter 9 - Digging deeper

Now that we have a good grasp of Go fundamentals, next we will delve deeper into the language and explore more advanced concepts. We'll expand upon previously covered topics and introduce new techniques such as assertion, reflection, and Go's new generics implementation, tools which should enhance our abilities as Go programmers.

## 9.1 Developing with functions

We've used functions a few times already. We've called functions from standard library packages such as `log` and `fmt` and we've also created functions in the examples ourselves.

Next, we'll focus on some of the less common aspects of functions in programming languages, features which we do have in Go. Specifically, we'll focus on function signatures including variadic arguments, and return styles, including multiple returns.

Note, that what we cover here, is equally applicable when working with *receivers*.

### 9.1.1 Function parameters

Functions often accept values passed to satisfy parameters in the function signature.

In Go, when defining function parameters we need to specify both the variable name and its type. To shorten the signature somewhat we can group parameters of the same type if it makes sense to do so, and it **isn't** detrimental to the API.

The snippets below show ungrouped and grouped parameters. See how in this case, grouping makes the ordering less intuitive.

```
// ungrouped parameters
func normalParams(name string, age int, houseNumber int, address1 string, address2 s\
tring){
    ...
}

// grouped parameters, which makes the API a bit awkward in this example
func groupedParams(name, address1, address2 string, houseNumber, age int){
    ...
}
```

Long function signatures can also be split with line breaks to improve readability. No escape character is needed when doing so.

```
// split over lines
func normalParams(
        name string,
        age int,
        houseNumber int,
        address1 string,
        address2 string,
        ) {
        return
}
```

Note the trailing comma which is required when the parameters (or arguments) are separated in this manner.

## 9.1.2 Variadic arguments

Go functions can accept variadic arguments. A variadic argument comprises one or more values which are supplied to a single function parameter, building a slice of those values for use within the function body.

There can be only one variadic parameter in each function signature and it must be the last parameter. It is also optional, so we can omit the argument if we wish.

A parameter which accepts variadic arguments is denoted by the *spread* operator, which precedes its type e.g. `...string`.

Within the function body, we will need to inspect the slice length to determine if one, multiple or no values were passed to the variadic parameter, and then handle those values accordingly.

Let's demonstrate with an example function which includes a variadic parameter, in which we call the function four different ways to demonstrate its flexibility.

**Example 84 - Passing values as variadic arguments**

```
package main

import "fmt"

func myVariadicFunc(name string, address ...string) {
        fmt.Printf("Hello %s!\n", name)
        fmt.Println("Addresses:")
        if len(address) > 0 {
                for i, addr := range address {
                        fmt.Printf("%d: %s\n", i+1, addr)
                }
        } else {
                fmt.Println("No address data supplied")
        }
}

func main() {
        // single argument
        fmt.Println("Single Argument")
        myVariadicFunc("Joe Bloggs", "Address 1")

        // multiple arguments
        fmt.Println("\nMultiple Arguments")
        myVariadicFunc("Joe Bloggs", "Address 1", "Address 2", "Address 3")

        // no argument
        fmt.Println("\nNo argument")
        myVariadicFunc("Joe Bloggs")

        // passing a pre-built slice, note the trailing spread operator
        fmt.Println("\nPassing a slice as a variadic argument")
        addresses := []string{"Address 1", "Address 2"}
        myVariadicFunc("Joe Bloggs", addresses...)
}

// Go Playground: https://go.dev/play/p/NPutwf2aF4l
```

Output:

```
Single Argument
Hello Joe Bloggs!
Addresses:
1: Address 1

Multiple Arguments
Hello Joe Bloggs!
Addresses:
1: Address 1
2: Address 2
3: Address 3

No argument
Hello Joe Bloggs!
Addresses:
No address data supplied

Passing a slice as a variadic argument
Hello Joe Bloggs!
Addresses:
1: Address 1
2: Address 2

Program exited.
```

### 9.1.3 Multiple return values

Another feature not present in many other programming languages is support for multiple function return values.

Typically, we'll take advantage of this by using two return values in our functions, usually to pass an error value, or boolean value (to denote success or failure), in addition to the value we are interested in.

While the flexibilty to return more than two values *is* often useful, there's a balance to be found. If the number of values we want from a function goes beyond three, it could mean our API would benefit from having those values as fields on a struct, or keys in a map. This way we could use a single value to return virtually unlimited data without complicating our codebase. It's also much easier to maintain.

I used three as the *standard* since I couldn't find a function in the Go standard library with more than three return values. Of course, there are no hard rules here, but, as Go developers, we should always consider the readability of our code.

When using multiple returns we must comma delimit them and enclose the group in brackets. For example, here is the signature for `Split()` from the standard library *strings* package.

```
func Split(s, sep string) (string, string, bool){}
```

### 9.1.4 Function return styles

Functions can use *either* of two return styles, *Anonymous* or *Named* return parameters. We can't mix styles within the same function.

We saw an example of anonymous return parameters just above in the `Split()` function signature. That same signature using named return parameters could look like this.

```
func Split(s, sep string) (param1, param2 string, param3 bool){}
```

When using named return parameters, there is no need to return the parameters explicitly. A simple `return` statemement, referred to as a *naked* return, is sufficient. Note also that we don't need to declare the variables explicitly in the function body, in the same way that we don't need to declared the input parameters in the function body. The return body of the function, performs the declaration and intialisation.

An example will make this much clearer.

**Example 85 - Anonymous and named return parameters**

```
package main

import (
        "fmt"
)

func anonymousReturns(firstname, lastname string) string {
        return fmt.Sprintf("%s %s", firstname, lastname)
}

func namedReturns(firstname, lastname string) (fullname string) {
        fullname = fmt.Sprintf("%s %s", firstname, lastname)
        return
}

func main() {
        fmt.Println(anonymousReturns("Joe", "Bloggs"))
        fmt.Println(namedReturns("Joe", "Bloggs"))
}

// Go Playground: https://go.dev/play/p/RdVlD9CwJS4
```

Output:

```
Joe Bloggs
Joe Bloggs

Program exited.
```

We have two simple functions, the first uses anonymous return parameters in the signature, the style we've used in all the examples up to this point.

The second function employs named return parameters. Observe the *naked* return statement. Observe also, that we make an assignment to `fullname`, we're not actually declaring it. Try replacing the = with the shortform to declare/initialise it := then run the example again.

The compiler complains that there are no new variables, since `fullname` was declared in the function signature.

```
./prog.go:12:11: no new variables on left side of :=

Go build failed.
```

The style you choose to adopt is largely a matter of preference. For larger functions with multiple return values, one can argue that named return parameters make the function body less clear and harder to read.

The counter to that is that the function signature itself is much more explicit. Developers have a clearer idea of what a function will give them back when named returns are used. With anonymous return parameters, the developer is presented with a list of types and they will need to inspect the function body to ascertain the usefulness of the values returned for those types.

The other consideration is the potential for bugs and confusion.

Because named return parameters are declared in the signature, and because they are initialised to their zero value, they satisfy the function return automatically even if they are never assigned.

**Example 86 - Bug risk or not?**

```
package main

import (
        "fmt"
)

func anonymousReturns() (string, string) {
        var firstname, lastname string // we had to declare lastname
        firstname = "Joe"
```

```
        return firstname, lastname // we had to return lastname
}

func namedReturns() (firstname, lastname string) {
        firstname = "Joe"
        return // did we forget the signature also includes `lastname`?
}

func main() {
        mask := "firstname: '%s', lastname: '%s'\n"
        f1, l1 := anonymousReturns()
        fmt.Printf(mask, f1, l1)
        f2, l2 := namedReturns()
        fmt.Printf(mask, f2, l2)
}

// Go Playground: https://go.dev/play/p/gHxfLdGgTaX
```

Output:

```
firstname: 'Joe', lastname: ''
firstname: 'Joe', lastname: ''

Program exited.
```

Compare the two functions in previous example. In the first, the `lastname` variable must be explictly created and explicitly returned by us. Yes, it could still be its zero value, but that would be down to us, it would be our mistake.

But, in the second function, we've not made an assignment either, and we don't mention the `lastname` variable at all, not even in a comment, and nor does the naked return.

But, `lastname` is returned as its zero value and passed back up the call stack. That could well be ok, something we plan to come back to, but we might not have intended to do this. What if we've forgotten we added it to the function signature in the first place?

Edge case? Yes, probably, but that's a mistake I've made myself. It's especially easy to make when the function body is large, so the function signature isn't in view to remind us. I'd suggest taking care when using named returns. For most usecases anonymous return parameters will be a better choice, but that's only *in my opinion*.

### 9.1.5 Functions are a type

Functions are themselves a type. Values of the type can be created and used just like any Go type. This means functions can be assigned to variables, passed as arguments to other functions, and returned as values from functions.

We call functions which accept and return other functions, *higher-order functions* and they offer a high degree of flexibility in how we arrange our code. Functions which accept other functions are also known as *callbacks*.

The example below shows a simple callback type function, we can implement as many different formatting functions as we wish. Providing they are of the type, of `func(string)string`, we can use them in `Formatter()`.

**Example 87 - Callback style functions**

```
1  package main
2
3  import (
4          "fmt"
5          "strings"
6  )
7
8  // Formatter accepts any func of type func (string) string
9  func Formatter(str string, format func(in string) string) string {
10         return format(str)
11 }
12
13 func main() {
14         upper := func(in string) string {
15                 return strings.ToUpper(in)
16         }
17
18         lower := func(in string) string {
19                 return strings.ToLower(in)
20         }
21
22         myText := "SomE rANDOm text TO foRmat"
23         fmt.Println(Formatter(myText, upper))
24         fmt.Println(Formatter(myText, lower))
25 }
26
27 // Go Playground: https://go.dev/play/p/KXSxlwcWJB4
```

Output:

```
SOME RANDOM TEXT TO FORMAT
some random text to format

Program exited.
```

We should also mention *closures*. A *closure* is a function that remembers the values of the variables from the place where it was created, as if it was bound to them. Nothing here is unique to Go, but closures can be useful when we want to use a function in a different place, but we still want it to have access to the variables it needs.

**Example 88 - Closure functions**

```
package main

import "fmt"

func main() {
        x := 10

        increment := func() int {
                x++
                return x
        }

        fmt.Println(increment()) // prints 11
        fmt.Println(increment()) // prints 12
}

// Go Playground: https://go.dev/play/p/x8XnvAM609C
```

Output:

```
11
12

Program exited.
```

Observe, in the example above, that the `increment` function has access to the `x` variable even though it is defined outside of the function. This is possible because `increment` is a closure.

## 9.1.5 Pointer or value returns

When writing functions, we need to decide whether to return values or pointers. The decision can have an impact on both memory use and how memory is managed. This shouldn't be an initial

consideration, as always we should prioritise correctness and maintainability until we can verify we have some performance issues.

The choice between returning a value or a pointer from a function should depend on the intended use case. If we want to create a single value of something in a function and share it throughout our program, we should return a pointer to that value. On the other hand, if we only need to use the value within the function and do not need to mutate anything we share up the call stack, it may be more efficient to return a copy of the value instead of a pointer.

A good example of when it is beneficial to return a pointer is when creating a database connection. By returning a pointer, multiple parts of the program can use the same connection simultaneously, rather than each part creating its own connection. This can help to improve the efficiency of the program by reducing the number of connections that need to be created and managed.

**Example 89 - Pointer return for database connection**

```
package main

import "fmt"

type Database struct {
        ConnString string
}

// NewDatabase is a naive factory to create a database connection
// it returns a pointer to share the address of the connection info
func NewDatabase(server, username, password string) *Database {
        if db != nil {
                return db
        }
        connString := fmt.Sprintf("%s@%s:%s", username, server, password)
        return &Database{connString}
}

var db *Database

func main() {
        db = NewDatabase("localhost", "joeblogs", "password")
        fmt.Printf("Connection String: %+v\n", db)
}

// Go Playground: https://go.dev/play/p/0afFBh0GaFG
```

Generally, returning a pointer forces the compiler to allocate that variable to slower memory, and we'll talk more about this in the next section on *memory management*.

# 9.2 Memory management

Go performs memory management on our behalf. The compiler chooses where to put the values our program creates - on either the *stack* or the *heap* - when compiling our program, and the garbage collector manages heap memory during program execution.

The *stack* is a region of memory used for storing local variables and function parameters. The total size of the stack is limited in size but stack memory offers fast access. Each function call or goroutine starts with an initial stack of 2KB dedicated to it. This may be grown as required.

The *heap* is a region of memory used for dynamically-allocated objects at runtime, or variables the compiler decides to put there and not on the stack. Heap memory is shared and not bound to any one function. There's more heap memory available than stack memory, but it's also slower to access. The process of garbage collection on the heap adds further overhead.

We refer to variables placed on the heap as *allocations*. As a rule of thumb, the fewer allocations our programs make, the better. An allocation represents data in slower access memory, which has to be cleaned up by the garbage collector.

It is not our intention to eliminate allocations. That's often impossible. For example, the simple act of printing using `fmt.Println()` may create allocations. The compiler treats this function from the standard library, like any custom function. If it cannot be sure what the function does with the argument it receives, it cannot determine the lifetime of the value. Contrast that with the Go's built-in `println()` function. The compiler *always* knows that `println()` does nothing else with the passed argument, so can leave the data on the stack, and no allocations are made.

The goal is to minimise allocations, especially in functions that are called frequently. Indeed the compiler will make many optimisations at compilation time, one of which is *inlining*, where code in function bodies is pulled into the main function eliminating calls and returns, and in many cases reducing allocations in the process.

The compiler uses a process known as *escape analysis* to determine what can be *inlined*, it uses the same process to determine whether a value can sit on the stack or must be placed on the heap. The compiler will make an allocation for multiple reasons. Wherever an address is shared, rather than a copy of the value, the compiler will make an allocation if it cannot be certain it is not used beyond the scope of the current function's lifetime.

As a *general* rule passing values around rather than pointers allows us to reduce allocations.

We can observe the escape analysis performed by the compiler, and we'll do this shortly.

## 9.2.3 Garbage collection

Heap memory management is the responsibility of the garbage collector which will deallocate memory when it can no longer find any references to it. The garbage collector is packaged with every Go program.

At runtime, the garbage collection process has a cost. Known as the GC pause, historically it was a *stop-the-world* type operation in which all goroutines were paused while heap memory was deallocated. In most applications this pause was imperceptible, but in high-throughput, high-performance applications it could cause performance issues such as bottlenecks and throttling.

In modern versions of Go, the garbage collection process is more sophisticated, it does not need to suspend all goroutines at the same time and generally completes much faster.

## 9.2.4 Observing compiler escape analysis

So, we have some basic criteria which may indicate when the compiler will use the heap and not the stack, but, the fact is, we will not be able to read a complex piece of code and reliably determine which type of memory will be chosen and how many allocations will result.

Furthermore, the algorithm, or ruleset, which decides what is stack memory and what is heap is constantly being refined. So, rather than guess, it's better that we undertake the same escape analysis that the compiler performs during compilation.

The snippet below shows how we use the `-gcflags "-m"` flag to view escape analysis output when running or building a program.

```
## build
go build -gcflags="-m"

## run
go run -gcflags="-m" ./main.go
```

Unfortunately, it's not possible to pass build flags when we run code on the Go Playground. To see the output from the following examples yourself, you'll need to run the code in a local environment, or just follow the output included.

Alongside escape analysis we can also use benchmarks, part of the testing suite we'll cover later, to determine when allocations are made. So, for each example, we'll create a benchmark file which we'll be able to run with this command.

```
go test -bench . -benchmen
```

The command instructs the *go test* tool to run all benchmarks, and include benchmark metrics for memory in the output.

Benchmarks will tell us how many allocations a piece of code makes. Escape analysis lets us identify the specific lines of code which cause those allocations.

A word on inlining. We are not passing the `-l` flag which turns off inlining globally. Instead, we will use the `//go:noinline` compiler directive in our examples to prevent functions from being inlined. This is my preference, since the benchmarks will also run with inlining disabled without changing the benchmark command.

The code shown below is used to generate the benchmark results. In a later chapter, we'll cover benchmarking when we discuss *quality assurance*.

```
package main

import "testing"

func BenchmarkMain(b *testing.B) {
        for i:=0; i< b.N; i++ {
                main()
        }
}
```

In this first example we're not calling any of our own code, we're simply using the standard library `fmt` package and the built-in `println()` function.

**Example 90 - Escape analysis with benchmark**

```
package main

import "fmt"

func main() {
        input1 := 1
        input2 := "Joe Blogs"

        fmt.Println(input1)
        fmt.Println(input2)

        //println(input1)
        //println(input2)
}

// Go Playground: https://go.dev/play/p/rqTcK20eVqV
```

Output:

```
// go run -gcflags="-m" ./main.go
./main.go:11:13: input1 escapes to heap
./main.go:12:13: input2 escapes to heap

// go test -bench . -benchmen
51032             22628 ns/op              16 B/op          1 allocs/op
```

The escape analysis output is abbreviated. We are interested in messages which say `x escapes to heap` or `moved to heap: x`.

If we see `x escapes to heap`, it means the value is leaving the functions local scope. It *often* means that the compiler will store the value on the heap, but not always.

If we see `moved to heap: x`, that's a clear message that the compiler has decided the value needs to be allocated on the heap, and there will be an allocation associated with that decision.

Looking at the above example output, we may expect the compiler to have made two allocations, but looking at the benchmark output, it only made one, see `1 allocs/op` on the right of the output. The allocation is made on the string value. The compiler does not know that the function does not reslice the string's backing data so it puts it on the heap. Comment the code appropriately and run it again to see.

That's the first takeaway. Escape analysis and benchmarking are worth using together, and *escapes to heap* does not always mean an allocation was made.

Finally, alter the code to use the `println()` built-in function, and no allocations are made. The compiler knows exactly what this function does with the passed argument, so places everything on the stack.

In the next example, we create several functions of our own and use escape analysis with benchmarking to understand the allocations which may result.

**Example 91 - Escape analysis on function returns**

```
// go run -gcflags="-m" ./main.go
// go test -bench . -benchmem
package main

type Customer struct {
        Name string
        Email string
        Age int
}

//go:noinline
```

```
12 func NewCustomer(name string) Customer {
13         cust := Customer{Name:name}
14         return cust
15 }
16
17 //go:noinline
18 func NewCustomer2(name string) *Customer {
19         cust2 := Customer{Name:name}
20         return &cust2
21 }
22
23 //go:noinline
24 func NewCustomer3(name string) *Customer {
25         return &Customer{Name:name}
26 }
27
28 func main() {
29         input := "Joe Blogs"
30         _ = NewCustomer(input)
31         _ = NewCustomer2(input)
32         _ = NewCustomer3(input)
33 }
34
35 // Go Playground: https://go.dev/play/p/rC5x05MpKP5
```

Output:

```
1 // go run -gcflags="-m" ./main.go
2 ./main.go:19:2: moved to heap: cust2
3 ./main.go:25:9: &Customer{...} escapes to heap
4
5 // go test -bench . -benchmen
6 15239967               66.54 ns/op           96 B/op          2 allocs/op
```

In this example, we have three very similar *factory* style functions which create and return a `Customer{}`, either as a value or as a pointer.

The first function returns a value, which is a copy of the value created inside the function. The compiler knows everything can sit on the stack - when the function is finished that copy does not reference anything inside the function. There are no allocations made.

The second function creates a `Customer{}` value `cust2`, but then its address is taken and a pointer is returned. The compiler can't know how this value will be accessed and used later and it lives beyond the function's lifetime, so this value can't sit on the stack, it must be *moved to the heap*.

The final function is similar to the last, except an address is immediately taken and returned. Escape analysis indicates this *escapes to the heap* indicating there may be an allocation, and the benchmark results confirm that an allocation was made.

## 9.3 Using receivers with custom types

Let's dig deeper into receivers.

Receivers are essentially a different kind of function. All the learning we've acquired in the earlier section on `developing with functions` is valid here too. However, unlike functions, receivers have a unique characteristic that makes them similar to methods in OOP.

Unlike methods which exist on a class, receivers can be bound to any user-defined type, which allows the receiver to access the value of that type. For example, if a receiver is bound to a custom struct, it can access the fields of the struct just like a method can access the properties of a class.

Receivers are useful when writing functionality intended to work with a value of a specific type, for example, *getter* and *setter* routines which read the value and can change the value.

Because they can be bound to a custom type, receivers have internal access to the value of that type. This avoids the *value* having to be passed into the receiver as arguments, which would be necessary if we chose to use functions to perform the *getter* and *setter* operations mentioned above.

Similarly, because the access is internal, any mutation of the value is also performed inside the receiver and not by return values as would be the case with functions.

Receivers can either *receive* a copy of the value, which as you would expect can only be read, not mutated, or, they can receive a pointer to the value, which allows the receiver to mutate the value from within the receiver body. These are known as *value receivers* and *pointer receivers* respectively. We covered *pointer* and *value* semantics in Chapter 6.

A *value receiver* gets a copy of the value on every call, which could be significant in size for large types. A *pointer receiver* uses less memory since only a copy of the memory address is passed into the receiver on every call. Consequently, for very large types, there can be a case for using *pointer receivers* even if the value will only be read and not mutated.

In the same way receivers have access to the value, they can also access other receivers bound to the same type, which means we can call other receivers from within receivers.

In the next example, we show a value and pointer receiver implementation on the same `customer` struct. Mixing them in this way is not considered good practice - we're doing it here to keep the example concise.

**Example 92 - Value and pointer receivers**

```
package main

import "fmt"

type customer struct {
        Name string
}

// UpdateName is a pointer receiver
func (c *customer) UpdateName(newStr string) {
        c.Name = newStr
}

// PrintName is a value receiver
func (c customer) PrintName() {
        fmt.Printf("This is reading the value: %s\n", c.Name)
    c.PrintLine() // calling another receiver
}

func (customer) PrintLine() {
        fmt.Println("-------------------------------------")
}

func main() {
        var cust = &customer{"Joe Blogs"}

        // read via value receiver
        cust.PrintName()

        // update
        cust.UpdateName("Dave Blogs")
        fmt.Printf("Updated string is value: %s\n", cust.Name)
        cust.PrintLine()
}

// Go Playground: https://go.dev/play/p/qTBFdsIDdCL
```

Output:

```
This is reading the value: Joe Blogs
------------------------------------
Updated string is value: Dave Blogs
------------------------------------

Program exited.
```

The `Printline()` receiver makes no use of the `customer` value at all, so we have removed the variable accessor `c` to indicate this is the case. The only case for using a receiver rather than a function here is that in text editors with code completion, it will form part of the same prompt API as it is bound to the same `customer` type.

Finally, a few *suggested* style rules when using receivers. The compiler will not enforce these rules, but they do help improve the consistency and clarity of our code.

If one or more of our receivers should mutate the value on which it is bound, then we need to use at least one *pointer receiver*, so should make all receivers bound to that type, a *pointer receiver*.

Conversely, if nothing mutates the value but only reads it, then we should bind only *value receivers* to the type, **unless** we are dealing with very large types and making copies of values would be very inefficient.

We should use a short variable to access to the value inside the receiver body. The context and scope are very clear. There is no need for semantic naming.

If a receiver neither reads nor mutates the value, we should remove the variable accessor entirely as a mechanism for communicating this. We may also consider using a simple function instead since there is no need for the receiver to be bound to a type it neither reads nor mutates.

# 9.4 Working with interfaces

We've already discussed interfaces in Chapter 7 on *type*. Despite my intention, we covered many of the core points of learning back then.

In this section we're going to quickly recap, and then look at some interface implementations of our own, demonstrating their value in building flexible, more extensible programs. We're also going use interfaces to create mock representations of services which we can use in our testing. We'll try not to go too deep into testing itself at this stage.

## 9.4.1 Recapping

In Go, an interface is a set of receiver signatures. A value of an interface type can hold any value that implements those receivers. Interfaces are a way to specify the required behaviour of a type: if

a type has the receivers specified by an interface, then it is said to implement that interface. It does not need to be explicitly stated in the code that it implements the interface.

The empty interface, `interface{}` aliased `any`, specifies no behaviour. Consequently, all built-in and user-defined types implement this interface since they all have at least no receivers bound to them. So we can supply any type as a value for the empty interface.

Interfaces are useful when we want to define generic behaviour that several types may need to implement. They allow us to write code using the interface - an abstraction of the implementation - instead of the implementation itself.

Writing code that accepts interfaces in preference to concrete types is a common pattern in Go. For example, many functions accept the *io.writer* and *io.reader* interfaces, enabling them to call the *Write* and *Read* implementations on the interface value passed. The function knows how to use the interface value because the interface defines the receiver signature parameters and return types. It cares nothing for the underlying implementation itself, only that it *has* an implementation.

While accepting interface values makes code more flexible, we generally prefer not to return interfaces except in specific situations such as the one we'll see shortly. It's rarely a good idea to return an empty interface type. Doing so imposes a burden on the caller and can result in ugly syntax. The caller may need to perform *type switch* logic on each return value to determine what the concrete type is, and then how to handle it.

## 9.4.2 Creating interfaces of our own

Consider this fictional scenario. We are writing a program that has a requirement to store data, when it restarts it should be able to access that same data.

We need a store for the data, and a means to write to and read from that store. We're not using an in-memory store such as a *map* because it can't persist the data beyond the current runtime. We might use a map as a cache, for fast data access but we're not doing since there's no requirement for exceptionally fast reads and writes.

So, the next simplest store we could use would be a file-based store, with data getting written to disk. We decided to implement a file store and the below code is what we came up with. Three receivers bound to the struct will let us perform all the CRUD operations we need, with the update operation being a rewrite of the entire record for an id.

**Example 93 - Simple file store**

```
package filestore

// FileStore is a struct representation of a file based store
type FileStore struct {
        Folder string
}

// Set writes a payload to the file store
func (fs *FileStore) Set(payload []byte) (string, error) {
        return "", nil
}

// Get retrieves an item of data from the store by id
func (fs *FileStore) Get(id string) ([]byte, error) {
        return []byte{}, nil
}

// Delete removes an item of data from the store by id
func (fs *FileStore) Delete(id string) error {
        return nil
}
```

Note the implementation is not needed for the examples, let's pretend we know what it is and that we have added it to the above code. For now, we can simply use NOOPs.

We'd use the file store in our code like this. Again note we're only concerned with how to create and access the store. Processing and error checking would be needed on the return values, but we've discarded them here.

**Example 94 - Using the store**

```
package main

import "store/filestore"

func main() {
        // new file store
        fs := &filestore.FileStore{"data"}

        // write to it
        id, err := fs.Set([]byte("some data"))
```

```
        _, _ = id, err

        // read an item
        data, err := fs.Get("xyz")
        _, _ = data, err

        // delete an item
        err = fs.Delete("xyz")
        _ = err
}
```

Everything is good. With this piece complete we can continue to write the rest of our application.

But, then we get a new requirement. The business wants the ability to configure additional storage types. We should support databases and also key-value stores such as *Redis* and *etcd*. The requirements do not state specifically which.

We now face a challenge. The requirements are vague, and we can't possibly write implementations for every type of database or key-value store the business *may* wish to use.

How can we accommodate this change so that we can move on? Maybe we report that we are blocked until the business tells us exactly what it needs, or maybe we know how to use interfaces.

Luckily for us, we've recently learned about interfaces, so we decide to perform a small refactor of our project. If we refactor around interfaces we suspect we won't need to concern ourselves with those unknown implementations at the moment, while making it very simple to accommodate them when the business makes its mind up.

Let's design the interface. Where should we start?

Well, we know what the program needs to do regardless of the storage medium chosen. It needs to *read*, *write* and *delete* on the store. So we will create an interface that represents this behaviour.

**Example 95 - Interface design**

```
package store

// ReadWriteDeleter must be implemented for each storage mechanism
type ReadWriteDeleter interface {
        Read(id string) (payload []byte, err error)
        Write(payload []byte) (id string, err error)
        Delete(id string) error
}
```

We've named our interface *ReadWriteDeleter* and we defined three behaviours on the interface. Though we changed the naming slightly, the signatures are otherwise the same as those of the receivers we implemented on the `FileStore` struct.

We should use small interfaces whenever possible, each with a limited set of behaviour since implementation is simpler. This tends to make an interface more flexible. Interfaces are also composable, just like structs, so we can build more specific large interface types from smaller interfaces by embedding them. We could for example, have composed our *ReadWriteDeleter* interface from three smaller interfaces, *Reader*, *Writer*, and *Deleter*.

Observe also the naming convention. That's generally the idiomatic way of naming an interface in Go. For example, *io.Reader* defines a *Read* receiver and *io.ReadWriter* defines both *Read* and *Write* receivers. Our *store.ReadWriteDeleter* interface defines, *Read, Write* and *Delete* receivers.

We're also going to add a factory function to the *store* package. This will allow us to quickly swap between stores. We use a switch statement so we can create store types when we need to. Currently, we only have one `case` statement as we only have a file store.

**Example 96 - Implementing store.NewStore**

```
// NewStore creates a store from available stores, depending on storeType argument
func NewStore(storeType string) ReadWriteDeleter {
        var s ReadWriteDeleter
        switch storeType {
        case "file":
                s = &filestore.FileStore{
                        Folder: "data",
                }
        }
        return s
}
```

Happy with our chosen interface design, and our `NewStore` factory function, next we'll alter our `FileStore` struct type to make sure it implements the interface. This involves changing two receiver names.

**Example 97 - Implementing store.ReadWriteDeleter**

```
package filestore

// FileStore is a struct representation of a file based store
type FileStore struct {
        Folder string
}

// Write writes a payload to the file store
func (fs *FileStore) Write(payload []byte) (string, error) {
        return "", nil
```

```
}

// Read retrieves an item of data from the store by id
func (fs *FileStore) Read(id string) ([]byte, error) {
        return []byte{}, nil
}

// Delete removes an item of data from the store by id
func (fs *FileStore) Delete(id string) error {
        return nil
}
```

The last thing we need to do is amend the code that creates and uses the store. We'll use the factory to create the store, and then work with the interface value's *Read*, *Write* and *Delete* functionality in preference to a specific store type. The changes are minor.

**Example 98 - Refactoring to use the interface**

```
package main

import (
        "store/store"
)

func main() {
        // new store, currently a file store
        st := store.NewStore("file")

        // write to it
        id, err := st.Write([]byte("some data"))
        _, _ = id, err

        // read an item
        data, err := st.Read("xyz")
        _, _ = data, err

        // delete an item
        err = st.Delete("xyz")
        _ = err
}
```

That's it we're done. Our program is now working with an interface and not a specific store implementation.

To add a new store type, say a database, we'd create an implementation for the database, which also satisfies the *store.ReadWriteDeleter* interface, and add a case statement in the `store.NewStore()` factory to create a database store.

The only change we'd need to make in `main()` would be to pass "database" as an argument to `store.NewStore()` on line 9, so the factory function creates a database store instead of a file store.

To see how flexible our code is now that it uses interfaces, why don't you go ahead and add a new store for yourself? It should only take minutes.

### 9.4.3 Using interfaces in testing

When testing our software, often, the code we need to test has external dependencies such as third party libraries, APIs or databases, to name just a few examples.

Whether these components are available during testing may depend on the type of testing undertaken. If we're conducting *integration testing*, we're testing the system as a whole so it could make sense to perform those tests using real implementations. If we're conducting *unit testing*, which focuses on testing the pieces of code which comprise the program, it's often inconvenient to test using real implementations, and in some cases it's impossible.

To avoid needing real implementations, we can create *mock* or *fake* implementations of our external dependencies, and use those in our testing instead.

One way to do this in Go is with interfaces. By creating an interface for the external dependency, and refactoring our code to use the interface and not the dependency directly, we can create a mock/fake implementation of the dependency, one which satisfies the same interface. This is ideal when unit testing.

In the next example, we'll walk step-by-step through this process. We'll create an abstraction for a notification service using an interface, then amend our code to use that interface. We'll modify the notification service implementation so that it implements the interface, and finally, create a mock implementation of the service which we can use during testing.

Let's begin.

In our current implementation `SendNotification()` calls a helper function `makeCallToApi()` to send using the external notifications API. The code inside `makeCallToApi()` is not relevant for the example, and we're discarding any errors.

**Example 99 - SendNotification function**

```
package main

// SendNotification sends a notification using the external notification API.
func SendNotification(to, notification string) error {
    if err := makeCallToAPI(to, notification); err != nil {
        return err
    }
    return nil
}

// helper to make the REST call to API to send notification
func makeCallToAPI(to, notification string) error {
    return nil
}

func main() {
    _ = SendNotification("Joe Blogs", "Hello there!")
}
```

Let's quickly inspect the current unit test for the `SendNotification()` function.

**Example 100 - A basic unit test**

```
func TestSendNotification(t *testing.T) {
    err := SendNotification("Test Joe Blogs", "Test message")
    if err != nil {
        t.Errorf("expected nil, got %v", err)
    }
}
```

At the moment the unit test would make calls to the notifications API. If it has a valid *API key* or *access token* testing this function could send real notifications. If it doesn't have access - which is more likely - it will always error causing the test to fail.

We can't really test it properly at the moment, but, with a little work we can do better.

The first thing to do is to create an interface. We've called it *Notifier* and it defines a single *Notify* receiver.

**Example 101 - Notifier service interface**

```
package service

// Notifier must be implemented for any service we use to send notifications.
type Notifier interface {
    Notify(to string, notification string) error
}
```

Painless. Next, lets modify our existing service to use this interface. We'll create a struct, and bind a *Notify* receiver to it, which will implement *Notifier*. Inside `Notify()` we'll make the call to the `makeCallToAPI()` function.

**Example 102 - Modifying the notification code**

```
package notifications

// NotifyService provides for sending notifications using external API
type NotifyService struct {}

// Notify sends notification
func (n *NotifyService) Notify (to, notification string) error {
    if err := makeCallToAPI(to, notification); err != nil {
        return err
    }
    return nil
}

// helper to make the REST call to API to send notification
func makeCallToAPI(to, notification string) error {
    return nil
}
```

The final step is to modify `main()` so it uses the interface in place of a concrete implementation.

**Example 103 - Modify main()**

```
package main

import "notifications"

// SendNotification sends a notification using a Notifier service
func SendNotification(svc service.Notifier, to, notification string) error {
    return svc.Notify(to, notification)
}

func main() {
    // create a service
    notifySvc := &notifications.NotifyService{}
    _ = SendNotification(notifySvc, "Joe Blogs", "Hello there!")
}
```

That process should feel familiar. It's mostly the same as in the last section. We've already improved our code by making it more flexible. For example, should the business decide to use an alternative notifications service it would be straightforward to add.

But, that's not our goal here. We want to test the code, and the changes we've just made, mean it's significantly easier to do this too.

Rather than add an additional *real* notifications service, we're going to add a mock notifications service. We'll create it in the same notifications package we used for the real implementation.

**Example 104 - Creating a mock notifications service**

```
package notifications

// MockNotifyService provides for sending test notifications
type MockNotifyService struct {
    sent [][]string
}

// Notify mocks sending a notification
func (mn *MockNotifyService) Notify (to string, notification string) error {
    mn.sent = append(mn.sent, []string{to, notification})
    return nil
}
```

The mock service doesn't make a call to the `makeCallToAPI()` helper function. Instead, it uses a slice to store any notifications which would have been sent.

Now we can update the unit test. We create a pointer to a mock service and pass it as an argument to `SendNotification()` instead of the real service.

To verify that we did indeed send a test message, we check that an element has been added to the `sent` slice.

**Example 105 - A better unit test**

```
package main

func TestSendNotification(t *testing.T) {
        tstSvc := &notifications.MockNotifyService{}
        if err := SendNotification(tstSvc, "Test Joe Blogs", "Test message"); err != nil {
                t.Fatalf("Something went wrong, unexpected error: %v", err)
        }
        if len(tstSvc.sent) != 1 {
                t.Errorf("expected 1 notification, got %d", len(tstSvc.sent))
        }

        // check that we stored the expected data
}
```

Don't worry if the syntax of the unit test is confusing, we'll come back to testing later when we discuss quality assurance.

We could go further with the test, and perhaps verify that the data stored in `sent` matches what was used in the test. This would confirm that we're passing and handling the correct parameters in the service. You can add that piece to the test yourself if you wish.

## 9.4.4 When to add interfaces

Unless we are certain that interfaces will improve our application, they shouldn't be our starting point. Notice, how in both the storage and unit test scenarios, the usecase for interfaces was identified after the application already existed. Consequently, the behaviour we needed to abstract was *fairly* obvious to us.

If we attempt to build our application around interfaces from the start, there a risk we add complexity which isn't ever justified - what if the business never asks for another storage type, or we could pass a different API key and use a test notifications API?

Did we *really* need an interface to provide a mock? The modifications we made were as much about being able to inject our dependencies so that we can swap them, as interfaces specifically.

Why not leverage the typed nature of functions and inject the `makeCallToAPI()` function? Then simply mock that as in the examples below?

**Example 106 - Injecting makeCallToAPI()**

```
package main

import "fmt"

// SendNotification sends a notification using the external notification API.
func SendNotification(provider func(string, string) error, to, notification string) \
error {
        if err := provider(to, notification); err != nil {
                return err
        }
        return nil
}

// helper to make the REST call to API to send notification
func makeCallToAPI(to, notification string) error {
        fmt.Println(to, notification)
        return nil
}

func main() {
        _ = SendNotification(makeCallToAPI, "Joe Blogs", "Hello there!")
}
```

**Example 107 - An alternative mock and unit test**

```
package main

import "testing"

var sent [][]string

// Notify mocks sending a notification
func mockCallToAPI (to string, notification string) error {
        sent = make([][]string, 0, 0) // reset the slice
        sent = append(sent, []string{to, notification})
        return nil
}

func TestSendNotification(t *testing.T) {
        if err := SendNotification(mockCallToAPI, "Test Joe Blogs", "Test message"); err !=\
 nil {
```

```
17                  t.Fatalf("Something went wrong, unexpected error: %v", err)
18          }
19          if len(sent) != 1 {
20                  t.Errorf("expected 1 notification, got %d", len(sent))
21          }
22  }
```

By attempting to build using interfaces too soon, we also risk creating the wrong abstractions, because there is no existing concrete implementation to use for reference.

We *could* end up rewriting code if we make early assumptions about the abstractions we need, which then later prove to be incorrect.

### 9.4.5 Summary

That concludes our discussion on interfaces though we will touch on them again shortly when discussing both *type assertion* and *generics*.

In this section, we've compounded our knowledge of interfaces. We developed two solid examples which demonstrate how we refactor concrete implementations into abstractions using interfaces, and we've shown the value of interfaces in general.

But, we also know the risks of trying to implement interfaces prematurely, and, though the testing scenario was an excellent use case for interfaces, we saw it wasn't the *interface* itself that made the code simple to test, but the dependency injection pattern we applied, which enabled us to substitute mock dependencies *during* testing.

## 9.5 Type assertion and reflection

During program execution, we may need to inspect and manipulate values of unknown type or work with concrete types represented by interface values. To do this, we can use techniques like *type assertion* and *reflection*.

### 9.5.1 Type assertion

Type assertion is a method for verifying the underlying type of an interface value. It allows for checking if a variable is of a specific concrete type, and if it is, obtaining a copy of its value as that type. We're then able to work with the concrete value, not the interface value.

In the example below, we have a simple struct with one `Greeting` field. It satisfies `MyInterface`, so we can assign the struct value to the interface.

**Example 108 - Type assertion provides the concrete type**

```
package main

import "fmt"

type MyInterface interface {
        Do() error
}

type MyStruct struct {
        Greeting string
}

func (ms MyStruct) Do() error {
        return nil
}

func main() {
        var x MyInterface
        x = MyStruct{Greeting: "Hello"}

        // cannot access the concrete type, `Greeting` field is not visible
        fmt.Printf("This will generate compilation error", x.Greeting)

        // perform type assertion
        y := x.(MyStruct)

        // we can now access the structs fields
        fmt.Printf("'Greeting' set to: %s\n", y.Greeting)
}

// Go Playground: https://go.dev/play/p/n9njIiD7Kk1
```

Output:

```
./prog.go:22:56: x.Greeting undefined (type MyInterface has no field or method Greet\
ing)

Go build failed.
```

However, since interfaces only define behaviour and not state, we cannot access the `Greeting` field via the interface. We get a compilation error when we try.

Comment out line 22 and run the example again. We perform a type assertion, asserting the type is `MyStruct`. The assertion is correct, so variable `y` receives a copy of the `MyStruct` value. With this concrete value, we can now access its receivers and fields.

An improperly constructed type assertion may introduce the risk of a runtime panic should the assertion fail - meaning the interface value does not hold the asserted concrete type.

For example, the following code will compile as expected but will panic when the program is run, since the type assertion cannot succeed. Type assertions are runtime, and not compile-time, checks.

**Example 109 - Invalid type assertion, compiles but will panic**

```
1  package main
2
3  func main() {
4          var i interface{}
5          i = "this is a string"
6
7          // we assert the interface contains a integer value
8          x := i.(int)
9          _ = x
10 }
11
12 // Go Playground: https://go.dev/play/p/j3egltwQKPo
```

Output:

```
1  panic: interface conversion: interface {} is string, not int
```

Fortunately, Go has us covered. A type assertion returns a second, *boolean* value, which indicates the success or failure of the operation. We should always handle this second value in our code to avoid a panic.

In the next example, the previous code has been rewritten to avoid the panic risk. Now when the type assertion fails, `x` contains its initialised *zero value*, which is `0`. Of course, in production, this may not be ideal either, and we may still choose to log the error and shut down the program rather than allow execution to continue.

**Example 110 - Safely performing a type assertion**

```
package main

import "fmt"

func main() {
        var i interface{}
        i = "this is a string"

        // we assert that it is in interface contains an integer value
        x, ok := i.(int)
        if ok {
                fmt.Printf("We have an integer of: %d", x)
        } else {
                fmt.Printf("We don't have an integer, setting to zero value: %d", x)
        }
}

// Go Playground: https://go.dev/play/p/3jfW7_hkRjM
```

Since interfaces can be satisfied by many types, we often can't assert that the interface value holds a single type. So, in this situation we can use a special *type switch* assertion to determine which, of any number of concrete types, is contained in the interface value.

A *type switch* style of assertion also allows us to provide a *default* case which is useful should the interface hold an unexpected type. Note, that a type switch assertion will not panic, even if no default case statement is supplied.

The code in the example below demonstrates a *type switch* style assertion.

**Example 111 - Type switch style assertion with default case**

```
package main

import "fmt"

func main() {
        var i interface{} = "hello"

        switch v := i.(type) {
        case int:
                fmt.Printf("i is an int with value %v\n", v)
        case string:
                fmt.Printf("i is a string with value %v\n", v)
```

```
13          default:
14                  fmt.Printf("i is of an unknown type with value %v\n", v)
15          }
16  }
17
18  // Go Playground: https://go.dev/play/p/0ovjw0QFJtL
```

So, in summary, *type assertion* is used when we know the concrete type stored in the interface value at compile time, and we want to get and use the underlying value during runtime. It could be a single type in the interface or one of several types which satisfy that interface. Correctly used, type assertion can provide a safe and performant conversion of the abstract interface value into its concrete type.

## 9.5.2 Reflection

Reflection allows us to inspect the *type* and *value* of a variable at runtime.

We use the standard library's *reflect* package to obtain information about a variable. We're also able to manipulate the variable, for instance calling its receivers and updating its fields, using the same *reflect* package.

Note, only exported struct fields are accessible using reflection. Attempting to access an unexported field will result in a panic.

Let's now look at a few common applications of the reflect package. In this first example we use reflection to discover the type of a variable.

**Example 112 - Discover the type of a variable**

```
1   package main
2
3   import (
4           "fmt"
5           "reflect"
6   )
7
8   func main() {
9           x := "hello"
10          t := reflect.TypeOf(x)
11          fmt.Println(t)
12  }
13
14  // Go Playground: https://go.dev/play/p/HFJswkAEnB5
```

Output:

```
string

Program exited.
```

Next, we'll use reflection to inspect a variable to determine if it holds a value or a pointer.

**Example 113 - Does a variable hold a value or a pointer?**

```
package main

import (
        "fmt"
        "reflect"
)

func main() {
        x := "hello" // value
        y := &x      // pointer

        v1 := reflect.ValueOf(x)
        fmt.Println("x is pointer:", v1.Kind() == reflect.Ptr)

        v2 := reflect.ValueOf(y)
        fmt.Println("y is a pointer:", v2.Kind() == reflect.Ptr)
}

// Go Playground: https://go.dev/play/p/Y7p3BSuVGjN
```

Output:

```
x is pointer: false
y is a pointer: true

Program exited.
```

In this third example, we're going to use reflection to work with both the exported fields and receivers of a struct.

**Example 114 - Working with struct fields and receivers**

```
package main

import (
        "fmt"
        "reflect"
)

type Person struct {
        Age int
}

func (m *Person) SayHello(name string) {
        fmt.Printf("Hello %s\n", name)
}

func main() {
        p := &Person{Age: 22}
        v := reflect.ValueOf(p)

        // call the receiver/method with argument
        sayHello := v.MethodByName("SayHello")
        name := reflect.ValueOf("Joe Blogs")
        sayHello.Call([]reflect.Value{reflect.Value(name)})

        // access the Age field, and change it
        age := v.Elem().FieldByName("Age")
        fmt.Println("Current age:", age.Int())
        age.SetInt(42)
        fmt.Println("New age:", age.Int())
}

// Go Playground: https://go.dev/play/p/D1GKlMeHs0V
```

Output:

```
Hello Joe Blogs
Current age: 22
New age: 42

Program exited.
```

As we can see, reflection is a powerful tool. Some tasks would not be possible without the dynamic type checking provided by reflection, but remember, with great power comes great responsibility.

Reflection adds significant complexity, making our code harder to read and understand, but it also has a performance cost.

Reflection is slower than direct access because it involves checking and converting between different types. Reflection uses heap memory, and allocations on the heap must be managed by the garbage collector. In tight or high iteration loops, the impact of reflection on speed and memory may be significant.

The following simple example compares the speed of direct variable access, with access via reflection.

**Example 115 - Speed of direct access vs reflection**

```
package main

import (
        "fmt"
        "reflect"
        "time"
)

var i int = 42

func DirectAccess() time.Duration {
        start := time.Now()
        for j := 0; j < 100000; j++ {
                val := i
                _ = val
        }
        elapsed := time.Since(start)
        fmt.Printf("Direct variable access: %s\n", elapsed)
        return elapsed
}

func ReflectAccess() time.Duration {
        start := time.Now()
        for j := 0; j < 100000; j++ {
```

```
                val := reflect.ValueOf(i)
                _ = val
        }
        elapsed := time.Since(start)
        fmt.Printf("Access using reflection: %s\n", elapsed)
        return elapsed
}

func main() {
        elapsed1 := DirectAccess()
        elapsed2 := ReflectAccess()
        inc := ((elapsed2 - elapsed1) / elapsed1) * 100
        fmt.Printf("Reflection takes %d%% longer than direct access", inc)
}
```

Output:

```
Direct variable access: 34.685µs
Access using reflection: 314.971µs
Reflection takes 800% longer than direct access%
```

# 9.6 Introducing Generics

*Generics* are a feature in most statically typed programming languages. Generics, or generic programming, allows us to write more reusable code by abstracting types away. It provides a way to reduce duplication, which would otherwise arise when writing functionality compatible with more than just a single type.

Go wasn't designed with support for generics but after much debate over several years, generics support was added to the language in version 1.18.

Mechanically, generics enable us to write code with values whose type can be specified later at the point of use, while maintaining type safety using syntax to place constraints on which types may be accepted.

In a later section we will examine generics in detail, and through several examples, we'll learn how to do generic programming in Go.

But first we need to understand the problem, which is essentially a side-effect of *static typing*.

## 9.6.1 Before generics

Consider the simple `sumInt64()` function in the following example. It's designed to add two integers of the *int64* type. An unremarkable, trivial function.

**Example 116 - Adding two *int64* integers**

```
package main

import "fmt"

func sumInt64(x, y int64) int64 {
        return x + y
}

func main() {
        fmt.Println(sumInt64(1, 2))
}

// Go Playground: https://go.dev/play/p/XDU49bgxRs_j
```

The problem is that its parameter and return typing mean it can only add *int64* types.

But, we can't count on all integers being of this type, so what do we do when we need to add *int8*, *int16*, or *int32* typed integers?

One obvious approach is to duplicate the same function three more times, one for each integer type that we'll need to add.

In the following example, we implement those new functions. Notice, that the function body is always the same, only the *parameter* and *return* types are different.

**Example 117 - Adding other integer types**

```
package main

import "fmt"

func sumInt8(x, y int8) int8 {
        return x + y
}

func sumInt16(x, y int16) int16 {
        return x + y
}

func sumInt32(x, y int32) int32 {
        return x + y
}

func sumInt64(x, y int64) int64 {
```

```
18          return x + y
19  }
20
21  func main() {
22          fmt.Println(sumInt8(1, 2))
23          fmt.Println(sumInt16(1, 2))
24          fmt.Println(sumInt32(1, 2))
25          fmt.Println(sumInt64(1, 2))
26  }
27
28  // Go Playground: https://go.dev/play/p/Q3SzagyF-CS
```

With the function duplicated, we can now add any integer type, which is great, but it does feel a bit awkward. All that duplication just to add two numbers?

Perhaps there is another way to solve the problem.

Well, there is/was. Before Go 1.18 we could already use a *form* of generic programming with *interfaces*. By using the empty interface specifically, rather than duplicating the function as we did in the previous example, we could write a single implementation that accepts the `interface{}` type as parameters and then performs a *type switch* on the arguments to determine how to handle them.

This approach has drawbacks too, which we will come to, but first, let's look at how we could write a single function that could add any pair of same typed integers.

**Example 118 - A single sumIntAny implementation**

```
1   package main
2
3   import (
4           "errors"
5           "fmt"
6           "reflect"
7   )
8
9   func sumIntAny(x, y interface{}) (interface{}, error) {
10          if reflect.TypeOf(x) != reflect.TypeOf(y) {
11                  return nil, errors.New("mismatched  types")
12          }
13
14          switch v := x.(type) {
15          case int8:
16                  return v + y.(int8), nil
17          case int16:
```

```
18                  return v + y.(int16), nil
19          case int32:
20                  return v + y.(int32), nil
21          case int64:
22                  return v + y.(int64), nil
23          }
24
25          return nil, nil
26  }
27
28  func main() {
29          res, err := sumIntAny(int8(1), int8(2))
30          if err != nil {
31                  fmt.Println(err)
32          }
33
34          fmt.Println("Result:", res)
35          //fmt.Println("Addition:", res+res)
36  }
37
38  // Go Playground: https://go.dev/play/p/GZ0JOBSwu9_n
```

One problem is evident immediately. The function is now significantly more complex.

We need to call it by explicitly stating which integer types we are passing. We need to check that whatever values are passed are the same concrete type so we can add them, and we have an error return value in case they are not.

We're using a *type switch* to handle the different types that could be represented by `interface{}` before performing the integer addition. But what if a string is passed, we don't handle strings currently, and we should probably return an error?

Another problem is the return type. Without performing a type conversion, we have to return the same type as was passed. Since this can be one of four integer types, we can only represent this with the empty interface.

We could choose to convert to an arbitrary integer type and return that type, but which one? If the caller passes *int8* values, does it make sense to return an *int64* result?

So we stick with the empty interface return, but that creates friction for the caller. The calling code, must introspect the interface, probably with another *type switch*.

The `fmt.Println()` performs that type switch internally, which is why it can print the value held in the interface, but try uncommenting line 35 to perform the simple addition. The program won't compile.

```
./prog.go:35:27: invalid operation: operator + not defined on res (variable of type \
interface{})

Go build failed.
```

So both solutions are clunky when you stop and think, but as Go developers, we've never known any different. This is just how it is - the price of type safety!

But, for other developers coming from languages which support generic programming, this is not how we should do it at all. They may find the above approaches quite naive compared to a solution based on generics.

Their frustration is genuine. Perhaps there is a better way?

## 9.6.2 Solving the problem with generics

In this section, we're going to tackle the same problem using generics.

We'll start with a simple but naive example, and build on that over several more examples until we understand everything that the current implementation of generics gives us, plus what might come in later versions of Go.

To work with generics examples locally, you'll need a Go version of 1.18 or above. At the time of writing the Go Playground has Go 1.19 available, so all the examples included here, will run over there.

One way a generic function differs from a normal function is that we can pass additional type parameters with constraints.

Think of these as placeholders for type. The concrete type will be assigned by the compiler when the function is called. The constraint provides some hints to the compiler about what that type is allowed to be. This means some checks are possible during compilation, and we'll see this in action shortly.

The type parameters together with their constraints are passed in square braces which must immediately follow the function's name.

In the snippet below we illustrate with a single type parameter and constraint. This is sufficient for our solution, but we're not restricted to a single type parameter/constraint, and we'll show an example of that later on.

```
func SumAny[T any] (x, y T) T {
    ...
}
```

So how do we reason about this unusual syntax? Let's walk through it.

`T`, is a placeholder or *alias* for a *type* which has the constraint `any` which means it can be any type. We could have passed the empty interface type, `interface{}` instead.

Both `x` and `y` parameters use the type *alias* for their type. The single return value is also using the alias. So, whatever type is passed for `x` and `y` - which must be the same - will also be the return type.

It's like a template, with real types being substituted for the placeholders when they become known.

The capitalization of the type parameter `T` is not a requirement, but things get a little (very) confusing if we mix lowercase type parameters with variable parameters. We're adopting the capitalization convention to improve readability.

Finally, `T` has no special significance, we could have used any name for the type parameter. Single letter or a semantic name. Once again, convention is good. We'll use the single letter, just as we'll use capitalisation.

So, now that we understand the semantics a little better, let's implement that function body of the `SumAny()` function.

It should be straightforward enough.

**Example 119 - A generic implementation of SumAny**

```
1  package main
2
3  import (
4          "fmt"
5          "reflect"
6  )
7
8  func SumAny[T any](x, y T) T {
9          return x + y
10 }
11
12 func main() {
13         res := SumAny(int16(1), int16(2))
14         fmt.Println(res)
15         fmt.Println(reflect.TypeOf(res))
16 }
17
18 // Go Playground: https://go.dev/play/p/jbzggpV7uM8
```

Output:

```
./prog.go:9:9: invalid operation: operator + not defined on x (variable of type T co\
nstrained by any)

Go build failed.
```

It doesn't compile.

The constraint we specified allows this function to accept *any* type, but not all types can be summed with the addition operator, since not all types are numbers.

The compiler recognises this during compilation, so this does't introduce a panic risk fortunately. It looks like what we need instead is a constraint that accepts only *any* integer type?

There are two ways we can achieve this. We can use a type parameter constraint list or we can create our own constraint using an interface.

We'll implement the same function again using both approaches.

First, let's use a type constraint list. We need to specify all valid types for our function and we do this using a *pipe* delimited list.

```
package main

import (
        "fmt"
        "reflect"
)

func SumAny[T int8 | int32 | int64 | int](x, y T) T {
        return x + y
}

func main() {
        res := SumAny(int16(1), int16(2))
        fmt.Println(res)
        fmt.Println(reflect.TypeOf(res))
}

// Go Playground: https://go.dev/play/p/yZemwfvY6ST
```

Output:

```
./prog.go:13:15: int16 does not implement int8|int32|int64|int (int16 missing in int\
8 | int32 | int64 | int)

Go build failed.
```

Again it doesn't build. This time, because we are passing `int16` types to the function, and we deliberately didn't include that type in the constraint list.

Add it to the list and run the example again - don't forget the pipe separator!

You should now see this output, and notice the return type used was `int16` too, because the type placeholder `T` became `int16` when the function was called.

```
3
int16

Program exited.
```

We've just written our first generic function in Go! The function signature is *bit* harder to read, but it's avoided us duplicating the function three or four times. Neither did we need to perform complex type switch style assertions in this single implementation. In fact the function body is the same as we started with.

However, we can do better. We can create our own constraints, which are interfaces basically, and specify the types allowed for that constraint. This single act will restore order in our codebase.

Go has two built-in constraints *any*, which we've seen, and *comparable*, which allows any type which supports equality comparisons with the == and != operators.

In later releases it's probable that the standard library will include a new constraints package[15], which could include an `Integer` constraint, but for now, if we need to go further than *comparable* we must build the constraint ourselves.

In this next example, we'll a create custom constraint *Numeric* by moving the constraint list into a new interface type.

[15] https://cs.opensource.google/go/x/exp/+/a68e582f:constraints/constraints.go

**Example 121 - Creating a custom constraint**

```
package main

import (
        "fmt"
        "reflect"
)

type Numeric interface {
        int8 | int16 | int32 | int64 | int
}

func SumAny[T Numeric](x, y T) T {
        return x + y
}

func main() {
        res := SumAny(int16(1), int16(2))
        fmt.Println(res)
        fmt.Println(reflect.TypeOf(res))
}

// Go Playground: https://go.dev/play/p/IxpNJBWPkZo
```

The output is identical to before, and the `SumAny()` function signature is simple to read again. As a bonus *Numeric* is reusable, we can avoid the constraint list in the next generic function that performs addition or subtraction, and instead use the *Numeric* constraint.

As you might expect, custom constraints, since they are interfaces, are composable. This allows us to build less restrictive constraints by composing from restrictive constraints.

In the next example, we'll amend the code so that `SumAny()` can also perform addition on unsigned integers, a glaring omission. But, rather than use one big constraint list, we will use composition to create the *Numeric* constraint, by creating and embedding a *Signed* and *Unsigned* constraint.

**Example 122 - Using composition with constraints**

```
package main

import (
        "fmt"
        "reflect"
)

type Numeric interface {
        Signed | Unsigned
}

type Unsigned interface {
        uint8 | uint16 | uint32 | uint64 | uint
}

type Signed interface {
        int8 | int16 | int32 | int64 | int
}

func SumAny[T Numeric](x, y T) T {
        return x + y
}

func main() {
        res := SumAny(int16(1), int16(2))
        fmt.Println(res)
        fmt.Println(reflect.TypeOf(res))
}

// Go Playground: https://go.dev/play/p/2yARr7xNBG8
```

Kudos to the Go development team! That is incredibly elegant in its simplicity, and the way it builds on what we already know.

But what about custom types?

We can use custom types in constraint lists and constraints, just like built-in types, but Go provides a modifier we can apply to built-in types, so that any user-defined type, which is based on one of the built-in types can be allowed.

Let's take our example in a slightly different direction to illustrate this.

**Example 123 - Allow any type with underlying type *int***

```
1  package main
2
3  import (
4          "fmt"
5          "reflect"
6  )
7
8  type OnlyInt interface {
9          int
10         // ~int
11 }
12
13 type myInt int
14
15 func SumAny[T OnlyInt](x, y T) T {
16         return x + y
17 }
18
19 func main() {
20         var myInt myInt = 1
21         res := SumAny(myInt, 2)
22         fmt.Println(res)
23         fmt.Println(reflect.TypeOf(res))
24 }
25
26 // Go Playground: https://go.dev/play/p/b8jNMrEtUcL
```

Output:

```
1  ./prog.go:21:15: myInt does not implement OnlyInt (possibly missing ~ for int in con\
2  straint OnlyInt)
3
4  Go build failed.
```

Once again, this code fails to compile, but the compiler message is very helpful. We are trying to pass a value of type *myInt*, a user-defined type which has the underlying type of *int*.

If you uncomment line 10, and comment out line 9, this tilde modifier tells the compiler that we will accept any type which has the underlying type of *int*.

The code should now compile and run, producing the following output.

```
3
main.myInt

Program exited.
```

The last example we're going to look at demonstrates how we can use multiple type parameters each with its own constraint. It's quite contrived but serves to illustrate the point.

**Example 124 - Multiple type parameters**

```
package main

import (
        "fmt"
        "reflect"
)

func MakeAddToMap[K comparable, V any](k K, v V) map[K]V {
        mp := make(map[K]V)
        mp[k] = v
        return mp
}

func main() {
        mp := MakeAddToMap("key", 1)
        fmt.Println(reflect.TypeOf(mp))
        fmt.Println(mp)
}

// Go Playground: https://go.dev/play/p/DFeMUaFqBzG
```

Output:

```
map[string]int
map[key:1]

Program exited.
```

Notice that we use the built-in *comparable* constraint on the `K` type parameter, as not every type can be used for a map key. If we had specified *any* the program would have failed to build.

The syntax is again a little confusing, but I suspect as we start to see more generic programming in Go, we'll get used to it.

### 9.6.3 Summary

We should now have a fairly good grasp of generics. Let's recap on what we've learned.

We know how to apply generic programming to solve some of the types of problems where traditionally we've just written more code, or settled for some type assertion.

We've seen the complexity generics can introduce into our syntax, and, while accepting that novelty and our unfamiliarity are factors, we've discovered ways we can mitigate this with custom constraints and composition, for example.

And, future versions of Go will likely add further features, and it's likely that more built-in constraints will be introduced which will reduce the footprint of generics further still.

But of course, not everything requires us to apply generics. Often a little duplication is absolutely fine. Such code is often simple to read, and simple to reason about. We must recognise that maintaining code is often the responsibility of a team of developers so we should be considerate of them, and their abilities, especially when applying features which are relatively new, and that not everybody has experience with.

# Chapter 10 - Concurrency

One of the most attractive features of Go is its support for *concurrency* which provides the ability to run multiple tasks at once.

In this chapter, we're going to explore Go's concurrency features, including *goroutines*, *waitgroups*, and *channels*. We'll learn how to use them, why they work well together, and how *context* also contributes to writing safer concurrent programs.

We'll also look at another more traditional style of concurrent programming, one where we share memory but use *mutexes* to lock memory for use by a single process at a time.

By the end of this chapter, we'll have the ability to identify opportunities for using concurrency in our programs. We'll know how to use the appropriate tools and, understand the risks associated with introducing asynchronous code blocks into our codebase.

*Concurrency* is not the same as *parallelism*. With concurrency, Go's included *scheduler* manages tasks which are in a *runnable* state, starting and stopping them based on several factors. The result is that *usually*, tasks complete much faster than if they were run sequentially, **but**, they are not all running at the same time.

## 10.1 Goroutines

Goroutines are the foundations of asynchronous programming in Go. They allow multiple functions to run concurrently within a single program by scheduling them as lightweight operations, on top of threads. Each goroutine is created with an initial stack size of 2KB which can be grown as required.

Goroutines are *non-blocking*. When a goroutine is started it returns immediately, leaving the function to be handled asynchronously by the Go scheduler.

The Go scheduler starts and stops goroutines based on several factors, which include:

- Goroutine priority
- Channel communication
- System load
- Available processors
- Goroutine stack size
- Number of existing goroutines.

Regarding *priority*, the Go scheduler dynamically assigns priority based on the goroutine's state, how long it has been blocked, and how many other goroutines are blocked and waiting to run.

*Channel communication* will often block goroutines, so the scheduler may start a goroutine while another is blocked, waiting to send or receive on the channel.

Another consideration for the Go scheduler is *stack size*. Goroutines with a large stack size may put pressure on system resources so *may* be scheduled less frequently.

We don't usually care about goroutine scheduling which takes place in the background - with possibly one exception. By default, the Go scheduler will schedule goroutines on all available CPU cores.

In most situations this is appropriate, but we may find our Go application is overutilising resources, limiting memory and CPU available to other applications on the same hardware. In such circumstances we can limit the CPU cores the scheduler can utilise using `GOMAXPROCS`.

This value can be set either as an environment variable of the same name or within the program itself using the *runtime* package, which means the CPU cores available to the program can be restricted for specific execution paths.

**Example 125 - Setting GOMAXPROCS during runtime**

```
package main

import (
        "fmt"
        "runtime"
)

func main() {
        fmt.Printf("Setting GOMAXPROCS to 5, previously was %d\n", runtime.GOMAXPROCS(5))
        fmt.Printf("Setting GOMAXPROCS to 2, previously was %d\n", runtime.GOMAXPROCS(2))
}


// Go Playground: https://go.dev/play/p/ptFRl7NAoT_6
```

Output:

```
Setting GOMAXPROCS to 5, previous was 8
Setting GOMAXPROCS to 2, previous was 5

Program exited.
```

The return value of `runtime.GOMAXPROCS()` is the previously set value, not the value just set. We ran the function twice to demonstrate that.

### 10.1.1 The *go* keyword

Although goroutines are very easy to create, it's rare that we can simply create a goroutine and forget about it. However, writing code that runs asynchronously in a goroutine is very simple, only requiring that we prefix a function invocation with the `go` keyword.

```
1  // calling a named function as a goroutine
2  go sendEmail(...)
3
4  // wrapping code in an anonymous concurrent function
5  go func(){
6      ...
7  }()
```

In practice, we usually at least want to be sure the function completes and using goroutines alone as in the previous example isn't sufficient to guarantee that. To illustrate this, check out the gotcha below, in which the expected output never prints.

**Example 126 - Where is the output?**

```
1  package main
2
3  import (
4          "fmt"
5  )
6
7  func main() {
8          go func() {
9                  fmt.Println("Will this print?")
10         }()
11
12         //time.Sleep(1 * time.Second)
13 }
14
15 // Go Playground: https://go.dev/play/p/nd5nwv5b3ok
```

Because the goroutine returns immediately, program execution continues. In this case, the program completes and exits before the body of the goroutine is scheduled and executed.

Uncomment line 12 which pauses the program, preventing it from terminating, for long enough that we see the expected output.

As well as completion, we often want to know the result of an operation performed by a goroutine, and, if anything went wrong. But since goroutines return immediately we can't use traditional

function return values. Even if we could, it would be unwise since it could easily lead to race conditions and other synchronization issues.

So, usually, we don't use goroutines in insolation. Instead, we combine them with *waitgroups* and *channels* which we'll cover shortly.

# 10.2 Context

Go introduced *Context* in version 1.7 in 2016. Context solves one important problem for us: *it provides a way to maintain control across processes.*

Its application is not limited to use with goroutines, but it fits well with our discussion of concurrency, and the problems of communication and control we may encounter when writing asynchronous code.

We've already seen how simple it is to run something concurrently using a goroutine and we'll look at the different approaches we can employ for signalling and messaging, **from** goroutines, back to our main program, later in this chapter.

But what about the other way? How does our main program communicate with a goroutine once it has started? How does it cancel a goroutine, for example, if the program must exit, allowing the goroutine the opportunity to clean up after itself and end gracefully?

We also need to manage goroutines during execution and not just when closing down our program. For example, if we are blocking while waiting for goroutines to finish their work, a single goroutine that can't complete its work could result in allocated memory which we can't reclaim and a slow or unresponsive program.

So, while it's easy to run a function concurrently by starting it in a new goroutine, managing that goroutine once it has started, is not as straightforward.

The context package solves this problem. Using a context, we can communicate *timeouts*, *deadlines*, *cancellations*, and other request-scoped values across API boundaries, processes and goroutines.

We still can't ensure a goroutine finishes its work - many factors will be beyond our control - but the context package allows us to limit the execution window for that work.

## 10.2.1 Conventions

There are two conventions we need to be aware of. We should use *ctx* as the idiomatic name for a variable which holds a context and, any function or receiver which accepts a context value, should take it as the first parameter.

## 10.2.2 The context.Background() empty context

The function `context.Background()` returns an empty context or `emptyCtx`. Think of it as the parent or root context, it has no cancellation rules, and no values or deadline.

Since context can be chained: a child can inherit a parent context and can add to the context, so cancellation rules, values, and deadlines can be added later.

Although its application is normally limited to `main()`, `context.Background()` is often used in testing of functions which require a context.

### 10.2.3 The context.TODO() empty context

This is essentially a placeholder. In fact, `context.TODO()` returns the same empty context as `context.Background()`, but the intended purpose of `context.TODO()` is different.

Since not all Go code uses context and, when developing or refactoring our code to use context it may not be immediately obvious how the context will be made available, we have `TODO`, a placeholder. We're saying we know this function will accept a context, but we don't know much more than that at this time.

`context.TODO()` should be used in function arguments in preference to `context.Background()`, the compiler differentiates between them when checking for correct context propagation.

### 10.2.4 Context with cancellation

Cancellation allows us to manually cancel those goroutines which have been written to handle and honour the context contract.

We have to write code in the goroutine to accomplish this, it isn't an automatic process. Goroutines which don't honour the contract will not terminate, despite a cancellation signal.

To implement cancellation on the current `context` we use the `context.WithCancel()` function. This returns a *cancelFunc* type, a function, which we can defer, or call explicitly in response to an event such as a control signal to terminate our program.

In the example below, we call the `cancel()` function after a short delay. One goroutine has been written to handle the context and returns on the cancellation signal. The second goroutine doesn't handle context, so carries on regardless, or at least until the timer unblocks and the program exits.

**Example 127 - Using context.WithCancel()**

```
1  package main
2
3  import (
4          "context"
5          "fmt"
6          "time"
7  )
8
9  func main() {
10         ctx := context.Background()
```

```
11          ctx, cancel := context.WithCancel(ctx)
12
13          // wait 5 seconds then call the cancelFunc
14          // which will stop any running goroutines which honour context
15          go func() {
16                  time.Sleep(5 * time.Second)
17                  cancel()
18          }()
19
20          // no context
21          go func() {
22                  for {
23                          time.Sleep(2 * time.Second)
24                          fmt.Println("I'm Dave Blogs and I AM unstoppable")
25                  }
26          }()
27
28          // with context
29          go func() {
30                  for {
31                          select {
32                          case <-time.After(1 * time.Second):
33                                  fmt.Println("I'm Joe Blogs I'm unstoppable!")
34                          case <-ctx.Done():
35                                  err := ctx.Err()
36                                  fmt.Printf("I SPOKE TOO SOON... '%v' OH NO!\n", err)
37                                  fmt.Println("Cleanup and graceful shutdown type stuff performed here
38                                  return
39                          }
40                  }
41          }()
42
43          // exit after 20 seconds
44          time.Sleep(20 * time.Second)
45  }
46
47  // Go Playground: https://go.dev/play/p/aAbmie_rtVV
```

See how we extended the parent context, building on the empty context returned by `context.Background()` and adding the cancellation to it?

Note also how one of the goroutines has been written to honour context? We're using a select statement and listening for signals on both the `time.After()` and `ctx.Done()` channels. When we

receive on the latter channel, we have the opportunity to stop what we are doing gracefully and generate logs before returning.

Finally, observe how `ctx.Err()` provides specific information about why the context ended, in this case its value was "context canceled".

### 10.2.5 Context with timeout

In the previous example, we created a context which could send cancellation signals. Although we cheated by running the cancellation function after a short delay, we could have called it conditionally.

In this section, we're going to create a context which has a *timeout*. Timeout specifies a duration of time. When we create a context with a timeout we are stating how long we are prepared to wait for the results of the goroutine. The context will expire when that duration of time has elapsed.

In this example and the examples which follow, the way we handle the context contract inside our goroutines is identical to the way we did it for cancellation, it's only the way we set up the context which differs.

In the example below, we've simplified the program so it only includes one goroutine, which handles context.

**Example 128 - Using context.WithTimeout()**

```
1  package main
2
3  import (
4          "context"
5          "fmt"
6          "time"
7  )
8
9  func main() {
10         ctx := context.Background()
11         timeout := 10 * time.Second
12         ctx, cancel := context.WithTimeout(ctx, timeout)
13         defer cancel()
14
15         // with context
16         go func() {
17                 for {
18                         select {
19                         case <-time.After(1 * time.Second):
```

```
                                fmt.Println("I'm Joe Blogs I've got 10 seconds to talk")
                        case <-ctx.Done():
                                err := ctx.Err()
                                fmt.Printf("Got to go now... '%v'\n", err)
                                fmt.Println("Cleanup e.t.c")
                                return
                        }
                }
        }()

        time.Sleep(12 * time.Second)
        fmt.Println("Exiting")
}

// Go Playground: https://go.dev/play/p/6y1hYI5pSdM
```

Observe the value of `context.Err()` when duration elapses versus manual cancellation.

```
Got to go now... 'context deadline exceeded'
```

Note that `context.WithTimeout()` also returns a `cancel` function and we're using *defer* to call it. This ensures that context is cancelled regardless of the time elapsed should the program exit for another reason such as a *panic*. This avoids what the compiler terms, "context leak".

## 10.2.6 Context with deadline

A *deadline* on a context is similar to a *timeout*, but, rather than stating a duration in which an operation must complete as we do with `context.WithTimeout()` when using `context.WithDeadline()` we instead set an actual time in the future that the operation may not go beyond.

Configuring a context with a deadline is, as you might expect, very similar to setting a timeout. Note that the printed time output will be incorrect if you run this on the Go Playground.

**Example 129 - Using context.WithDeadline()**

```
package main

import (
        "context"
        "fmt"
        "time"
)

func main() {
        ctx := context.Background()
        deadline := time.Now().Add(10 * time.Second) // 10 seconds in future
        ctx, cancel := context.WithDeadline(ctx, deadline)
        defer cancel()

        // with context
        go func() {
                for {
                        select {
                        case <-time.After(1 * time.Second):
                                fmt.Printf("I'm Joe Blogs I can talk until %s\n", deadline)
                        case <-ctx.Done():
                                err := ctx.Err()
                                fmt.Printf("Got to go now... '%v'\n", err)
                                fmt.Println("Cleanup e.t.c")
                                return
                        }
                }
        }()

        time.Sleep(12 * time.Second)
        fmt.Println("Exiting")
}

// Go Playground: https://go.dev/play/p/MyjT3EVGkus
```

## 10.2.7 Sharing data via context

As well as carrying *cancellations*, *deadlines*, and *timeouts*, we can also use context to share data.

Sharing data via context is useful when passing certain kinds of data from one function to another, or multiple goroutines, without explicitly passing it through function parameters or sharing as a global variable.

The focus there was on specific types of data. Incidental data, not central to the main purpose of the application is ideal for sharing via context. Examples of such data include API keys, JWT tokens, user credentials, or session tokens, which are request-scoped, relatively small in size, and not essential to the application's business logic.

We should avoid using context to share large values, and, for the sake of readability, data which is part of our program's business logic should be shared explicitly rather than within the context.

In the next example, we demonstrate how to share values using context. We first create a context using `context.WithValue()`, and to retrieve values from the context inside of our goroutines we simply use `context.Value()`.

The *key* and *value* parameters in `context.WithValue()` are `interface{}` types. On the value, that's useful, since we can use aggregate types here. For example, we could use a struct to represent multiple request-scoped data within its fields.

But why is *key* of type `interface{}`? Why not use a basic type like *string*? The answer is that keys should be custom types (actually custom comparable types) to avoid naming collisions between packages that might use context and have the same key name. By using a custom type, keys - even those of the same name - will have a different type.

**Example 130 - Sharing data via context**

```
package main

import (
        "context"
        "fmt"
        "time"
)

type ctxKey string

func main() {
        var name ctxKey = "name"
        ctx := context.WithValue(context.Background(), name, "Joe Blogs")

        // with context.Value()
        go func() {
                for {
                        time.Sleep(1 * time.Second)
                        fmt.Printf("My name is %s\n", ctx.Value(ctxKey("name")))
                }
        }()

```

```
        time.Sleep(12 * time.Second)
        fmt.Println("Exiting")
}

// Go Playground: https://go.dev/play/p/7p504omEWsq
```

There's no deadline or timeout set on the context in the previous example, we used context only to share a single value. We could have used *chaining* to add timeouts and deadlines, and share additional values.

Observe also how we created a custom type, based on *string*, and used a variable of that type for the context key.

## 10.3 Blocking execution with waitgroups

Waitgroups provide a simple way to wait for multiple goroutines to complete, before continuing program execution. The alternative would be a naive pause for some arbitrary amount of time as we saw in the previous examples, **which is not recommended**, or implementing more complex communication over channels, purely to signal completion.

Waitgroups are most useful where there is no requirement to communicate results or data between goroutines, something which would mandate the use of channels.

Goroutines managed by a waitgroup are capable of updating simple shared state, for example, a counter, or an array of known size.

In the next example, we use a waitgroup to hold execution until goroutines have completed, each populating an element in the `results` array. The example demonstrates how to orchestrate a waitgroup, and how safe state updates are possible in some circumstances.

Beyond a demonstration, this would be a poor application for concurrency. The code itself is very simple, and our inference *should* be that is unlikely to be faster than it's synchronous counterpart. In my testing it was between two and three times slower to schedule goroutines, than to populate the array directly.

**Example 131 - WaitGroup updating shared state safely**

```
package main

import (
        "fmt"
        "sync"
        "time"
)

func main() {
        t := time.Now()
        var results [30]int

        var wg sync.WaitGroup
        //wg.Add(29)

        for i := 1; i <= 30; i++ {
                wg.Add(1)
                go func(i int) {
                        results[i-1] = i * 10
                        wg.Done()
                }(i)
        }

        wg.Wait()
        fmt.Println(results)
        fmt.Println("Completed in", time.Since(t))
}

// Go Playground: https://go.dev/play/p/Z3Rh0ufgiVr
```

Running through the code. The `wg` variable is of type *sync.WaitGroup*. We're making use of closures so not providing the `wg` argument as a parameter to the anonymous function. When a *sync.Waitgroup* typed variable is passed as an argument to a named function it should be passed by reference and not by value.

We add `1` to the waitgroup every time we invoke a new goroutine with `wg.Add(1)` and when each function has completed its work we call the `wg.Done()` receiver.

`wg.Wait()` blocks execution until all goroutines report they have completed via `wg.Done()`.

Observe `wg.Add(29)` on line 14. It is generally better to add goroutines to the waitgroup at the point of use - for example in the loop as we do. If there is a mismatch between the number of goroutines we add, and the number which call `wg.Done()` our program will panic at runtime. By calling `wg.Add(1)` in the loop which invokes the goroutine we prevent this possibility. Uncomment line 14, and comment out line 17 to experiment for yourself.

To summarise, waitgroups are simple to reason about. The waitgroup understands how many completion notifications it should expect, and each goroutine informs the waitgroup when it has completed its work. When all goroutines are accounted for, the waitgroup unblocks and program execution continues.

## 10.4 Sharing variables using mutexes

In the previous example, we used goroutines to populate the elements of an array which had a predefined size and capacity. It's safe to do this, because, as the code is written, each goroutine mutates just one element in the array. We are sharing the array, but we're not sharing the elements which comprise the array.

But few values can be safely shared in this way. Usually, unguarded concurrent access is unsafe, potentially giving rise to a race condition. Certainly any concurrent operation which reads a shared value and then changes that value is unsafe.

Consider the following example, which creates 10,000 goroutines, each incrementing the `myCounter` variable by a value of `1`. There is a waitgroup in place, so the program blocks until all goroutines have completed.

We print the `myCounter` variable once the work is done and expect the printed output to be `10000`.

**Example 132 - WaitGroup updating shared state unsafely**

```
1  package main
2
3  import (
4          "fmt"
5          "sync"
6  )
7
8  func main() {
9          var myCounter int
10
11         var wg sync.WaitGroup
12         for i := 0; i < 10000; i++ {
13                 wg.Add(1)
14                 go func() {
```

```
                        myCounter += 1
                        wg.Done()
                }()
        }

        wg.Wait()
        fmt.Printf("Counter total is: %d\n", myCounter)
}

// Go Playground: https://go.dev/play/p/dW1kdCrqmCu
```

Output:

```
Counter total is: 9764

Program exited.
```

But it isn't 10,000. In fact every time we run the code, we get a different result, and it's never 10,000. Our code has a problem since there is unguarded shared access to the `myCounter` variable. Multiple goroutines can read the value in that memory and increment it, simultaneously.

The actual value printed for `myCounter` depends on how many goroutines manage to read and write to `myCounter` atomically before another goroutine reads the variable. There's no way we can predict that access which is why we get a different result each time.

In short, we have a *race condition* or *data race*.

### 10.4.1 Implementing mutually exclusive access

Mutexes are guard mechanisms which guarantee exclusive access to a piece of memory. They are part of the `sync` package along with waitgroups. To declare a variable (or struct field) which holds a value of type `sync.Mutex` we generally use the naming convention `mu`.

We're going to improve the previous code and remove the race condition. We'll use a *mutex* in the example which follows, but in section 10.3 we'll solve the same problem using a *channel*.

**Example 133 - Mutex to guarantee exclusivity of access**

```
package main

import (
        "fmt"
        "sync"
)

func main() {
        var myCounter int
        var wg sync.WaitGroup

        var mu sync.Mutex

        for i := 0; i < 10000; i++ {
                wg.Add(1)
                go func() {
                        mu.Lock()
                        defer mu.Unlock()

                        myCounter += 1
                        wg.Done()
                }()
        }

        wg.Wait()
        fmt.Printf("Counter total is: %d\n", myCounter)
}

// Go Playground: https://go.dev/play/p/ZObOF_af_W9
```

Output:

```
Counter total is: 10000

Program exited.
```

We added three lines of code to create a *mutex* typed variable `mu`, apply a lock with the `mu.Lock()` statement, and create a deferred call to `mu.Unlock()`.

When we run the program, `myCounter` outputs the expected value of `10000`.

By using a mutex, each goroutine is granted exclusive access to the variable. When one goroutine locks the variable for reading and writing, others have to wait until it is unlocked before they can access it.

We used the *defer* keyword immediately after the statement which created the lock, ensuring we didn't forget to add it later.

Omitting `mu.Unlock()` would cause a deadlock at runtime, a type of panic which occurs when all goroutines dependent on a channel or mutex are blocked and cannot proceed. The runtime issues this error `fatal error: all goroutines are asleep - deadlock!` followed by a stack trace.

The deferred statement is executed just before the function exits. If the goroutine performs additional tasks after incrementing `myCounter`, it may result in the variable being locked for an extended period and cause other goroutines to be blocked for longer than necessary. This can lead to a longer program execution time. Should this be identified as a problem, to optimize this, it may be advisable to call `mu.Unlock()` immediately after the incrementing `myCounter`.

## 10.4.2 Write-only lock

In many cases it will be safe to read from a shared variable as long as that variable is not being written to and Go includes `sync.RWMutex` specifically for this purpose.

Read operations can be performed by many goroutines simultaneously with a readers lock, but a single write operation locks the shared variable for both *read* and *write* access.

The shared variable remains in a consistent state, race conditions are still avoided, but, removing the majority of the read locks helps limit blocking and improve overall application performance.

In the writing goroutine, we use the same syntax as previously which creates a full lock.

```
1 mu.Lock()
2 defer mu.Unlock()
3 myCounter++
```

In the reading goroutine, we implement a *readers lock* like so. This will only be blocked when there is a concurrent write operation.

```
1 mu.RLock()
2 defer mu.RUnlock()
3 fmt.Println("current value is:", myCounter)
```

### 10.4.3 The Race Detector

Spotting unsafe shared access is not always easy. In large programs with many lines of code, manual review may not be enough to spot potential race conditions. Fortunately, Go provides a *race detector* which can identify unsafe concurrent sharing of variables on our behalf.

To enable it, we simply add the `-race` flag to any `go run`, `go build` or `go test` command.

```
1 go run -race main.go
```

With the `-race` flag set, the Go compiler creates a modified version of the application that can track variable access and report any unguarded shared access which could result in a data race. It does this throughout the lifetime of the program, but only over paths that are executed.

It's a good idea to use this flag with *test* builds, however, a good test suite with high coverage is essential to ensure that as many execution paths as possible are checked for data races. The race detector can only identify race conditions it encounters, it does not analyse the code to find or predict them.

An executable compiled to include the race detector will use slightly more memory and be slightly slower. Although the difference is unlikely to be meaningful for most applications, it is worth stating.

## 10.5 Communicating with channels

We discussed channels briefly in Chapter 7 and we've already seen channel mechanics employed earlier in this chapter when we considered *context*, which uses channels in its implementation to send cancellation signals.

For the remainder of this chapter, we'll look at how channels provide a mechanism for messaging and signalling between concurrent goroutines. There's a lot to cover, but we'll progress slowly, and hopefully logically. By the end of this chapter, we'll have channels down.

Let's get started!

### 10.5.1 Signalling versus messaging

Messaging over channels speaks for itself. We pass messages, or data, which could include work to be performed, results of work and possibly errors too, between goroutines, or from goroutines back to our *main* execution.

But what about *signalling*? When signalling over channels, we may still send data, but the data is *signal* rather than *information*. Channels for signalling, provide a way for goroutines to inform each other - and main - of their status.

The idiomatic way to create a signalling channel is to use a channel of type empty struct. Although we'll often see types of *bool* or *int* used too, the empty struct is better for a few reasons.

An empty struct is 0 bytes in size, so it occupies no storage. It **cannot** carry any data, so it is clear the channel is for signalling only.

If we use *bool* or *int* types, the intent is less clear. Is the channel being used to share state which may fluctuate between `true` and `false`, or carry a number result, or is it for signalling? We can't be sure just by looking at the channel declaration alone. With an empty struct, there is zero ambiguity.

Often, we'll use a signalling channel so goroutines may inform *main* when they have completed their work, similar to how they would call `wg.Done()` if we implemented a *waitgroup*. That's probably one of the most common applications of signalling channels, but we can do more.

By way of demonstration, in this next example, we'll solve the race condition problem we identified in the previous section. But we won't use a mutex this time, instead, we'll leverage the signalling (and blocking) nature of channels, to implement a guard around the shared `myCounter` variable.

**Example 134 - Signalling channel for mutual exclusion**

```
1  package main
2
3  import (
4          "fmt"
5          "sync"
6  )
7
8  func main() {
9          var myCounter int
10         var wg sync.WaitGroup
11
12         var lock = make(chan struct{}, 1)
13
14         for i := 0; i < 10000; i++ {
15                 wg.Add(1)
16                 go func() {
17                         lock <- struct{}{}
18                         myCounter += 1
19                         <-lock
20
21                         wg.Done()
22                 }()
23         }
24
25         wg.Wait()
26         fmt.Printf("Counter total is: %d\n", myCounter)
```

```
}

// Go Playground: https://go.dev/play/p/JHiaz5Rjvii
```

Output:

```
Counter total is: 10000

Program exited.
```

The result is correct, but we didn't need a mutex. Instead, we declared a channel value of type `struct{}` as the `lock` variable. The first goroutine to place a `struct{}` value on the channel, fills the channel, meaning no other goroutines can put a value on the channel as there is no capacity.

Once the first goroutine has completed its work and incremented `myCounter`, it reads back from the channel, discarding the value read. The channel now has capacity again.

Another blocked goroutine can write a `struct{}` to the channel, which again locks the remaining goroutines out until it completes its work and reads back from the channel.

This process of *locking* and *unlocking* continues until all goroutines have performed their work.

The channel has a buffer size of `1` which is important. Try removing the buffer and see what happens. The program panics with a deadlock error. We'll discuss unbuffered and buffered channels, as well as why the buffer is crucial for this example, in the next section.

We can see the syntax employed to place a value on the channel `lock <- struct{}{}` and read back from the channel `<- lock`. As mentioned, we discarded the value we read, since it was just a signalling channel in this example.

Had we been sharing data - data which needed for later use - we could have captured it in a variable like so.

```
channelValue := <- dataChan
```

## 10.5.2 Buffered or unbuffered channels?

We can create both unbuffered and buffered channels in Go. A buffered channel can have a capacity of anything from a value of `1` upwards and the buffer size is considered to be a part of the channel's type declaration. Buffered and unbuffered channels behave differently, and we need a way to reason about that different behaviour.

Think of a buffered channel as a mailbox which can hold as many letters as the buffer size allows. Because it is a mailbox, users of the mailbox - the courier and recipient - can access it independently of each other.

As long as there is space in the mailbox, the courier can leave an item of mail in the mailbox and then continue about their business. As long as there is some mail in the mailbox the recipient can retrieve it one piece at a time.

Even if the mailbox has the capacity to hold only one item of mail the courier can make a delivery, and then continue about their business, and, as long as the recipient collects that item before the courier next arrives with another item, a mailbox which holds just one item of mail is sufficiently sized.

If there is no space in the mailbox, the courier must wait for the recipient to arrive and remove an item. If there is nothing in the mailbox when the recipient arrives, the recipient has to wait until the courier arrives to put an item in the mailbox before leaving with that item

So how does that compare to an unbuffered channel? Think of that as signed-for hand delivery instead.

When the courier arrives with an item, they can't just deposit it in the mailbox and go about their business. They must wait for the recipient to meet them before handing over the item.

Similarly, if the recipient is waiting for a delivery, they must wait for the courier to arrive with their item before they can do anything else.

So that's our courier analogy, let's summarise that specifically in relation to channels.

An unbuffered channel blocks either the send or receive operation until both can take place simultaneously. Like hand delivery. Unbuffered channels are often referred to as *synchronous channels*.

A buffered channel which has contents *and* spare capacity does not block either the send or receive operation. When full, the send operation is blocked until capacity becomes available, and when empty, the receive operation is blocked until the channel contains something in the buffer to take. They're a *bit* like a mailbox, and are often referred to as *asynchronous channels*.

How the buffer fills and depletes depends on the workloads being performed in the goroutines which send and receive on the channel but in many cases a buffer of 1 will be sufficient to ensure neither send nor receive operations get blocked.

Before we move on, let's apply this new knowledge to Example 134. Why was the buffer important in that code, and why did the code panic if we used an unbuffered channel instead?

With a buffer size of one, the first goroutine could add a value to the channel and then continue its work. That work entailed incrementing the `myCounter` variable and then reading back from the channel. With the first value on the channel there was no capacity for any more values, so the rest of the goroutines had to wait until there was capacity again.

The problem with using an unbuffered channel here is that no goroutine is waiting to receive on the channel. The first goroutine to use it, which wants to send, has to wait until another goroutine is ready to receive before it can send. It is blocked.

But, no goroutine ever arrives to receive on the channel. They are all send first goroutines, which in

any case, are also blocked. We have a deadlock. None of our goroutines can be scheduled by the Go scheduler. The program panics.

It's a simple example but hopefully serves to illustrate how we should reason about channel communication. The second takeaway is that it is very easy to introduce bugs when working with channel communication, in this case, by simply using a channel without a buffer.

Where there is a known amount of work to perform, the channel buffer can be sized to store the results of that work. A batch of worker goroutines can perform the work and place the results on the channel. Another batch can then process the results by reading from the channel buffer.

A real-life example could be a series of HTTP requests for data. The data is fetched and stored in the channel buffer, and then processed later.

The example below illustrates the mechanics. Operations of this kind are one of the few times we will create a channel with a buffer larger than `1`.

**Example 135 - Sizing the buffer to store results**

```
1  package main
2
3  import (
4          "fmt"
5          "sync"
6  )
7
8  func main() {
9          var wg sync.WaitGroup
10         store := make(chan int, 3)
11
12         // fill the channel buffer
13         fmt.Println("Filling the buffer")
14         for i := 0; i < 3; i++ {
15                 wg.Add(1)
16                 go func(i int) {
17                         store <- i
18                         fmt.Printf("Channel buffer contains %d values\n", len(store))
19                         wg.Done()
20                 }(i)
21         }
22
23         wg.Wait()
24
25         // read back from the channel buffer
26         fmt.Println("Reading the buffer")
27         for i := 0; i < 3; i++ {
```

```
28                  wg.Add(1)
29                  go func() {
30                          fmt.Printf("Reading value %d from channel buffer\n", <-store)
31                          wg.Done()
32                  }()
33          }
34
35          wg.Wait()
36
37          fmt.Println("Finished")
38  }
39
40  // Go Playground: https://go.dev/play/p/YVTWPzUYht9
```

Output:

```
1  Filling the buffer
2  Channel buffer contains 1 values
3  Channel buffer contains 2 values
4  Channel buffer contains 3 values
5  Reading the buffer
6  Reading value 2 from channel buffer
7  Reading value 1 from channel buffer
8  Reading value 0 from channel buffer
9  Finished
10
11 Program exited.
```

## 10.5.3 Unidirectional channels

Channels themselves are bi-directional. They can receive data from goroutines which send on the channel, and they can hand over data to goroutines which receive on the channel.

However, the Go type system supports *unidirectional* channel typing: channel types on which it is only possible to send, and channel types on which it is only possible to receive.

By specifying a unidirectional type, when we share a channel around our program, although the channel itself is bi-directional, we can restrict its usage with a unidirectional type. In doing so, we add further type safety to our programs. The compiler will fail to build our program if we attempt to send on a receive-only unidirectional typed channel and vice versa.

Below we illustrate the syntax. Try changing the `chan` type on line 17 to either unidirectional type. Because the function `SenderReceiver()` performs both a *send* and a *receive* on the channel, restricting it to either direction will cause compilation to fail.

**Example 136 - Unidirectional channel types**

```
package main

import (
        "fmt"
)

func Sender(ch chan<- string, message string) {
    // chan<- unidirectional send
        ch <- message
}

func Receiver(ch <-chan string) {
    // <-chan unidirectional receive
        fmt.Println(<-ch)
}

func SenderReceiver(ch chan string, message string) {
    // chan bi-directional send or receive
        ch <- message
        fmt.Println(<-ch)
}

func main() {
        ch := make(chan string, 1)

        Sender(ch, "Hello World")
        Receiver(ch)
        SenderReceiver(ch, "Hello World again")
}

// Go Playground: https://go.dev/play/p/ztggMJsU0cj
```

## 10.5.4 Signalling using an unbuffered channel

So far we've used waitgroups to synchronise executing goroutines, but we can also utilise the blocking characteristics of an unbuffered channel in many cases.

In this next example, we show what is a commonly used pattern, which prevents the program from exiting until the goroutine has completed its work.

**Example 137 - Unbuffered channel for signalling**

```
package main

import (
        "fmt"
        "time"
)

func main() {
        done := make(chan struct{})
        fmt.Println("The receive operation on the channel is blocked")
        go func() {
                for i := 5; i > 0; i-- {
                        fmt.Printf("Signalling done in %d seconds\n", i)
                        time.Sleep(1 * time.Second)
                }

                done <- struct{}{}

        }()

        <-done

        fmt.Println("OK, exiting")
}

// Go Playground: https://go.dev/play/p/ro8OHTv4G4x
```

Output:

```
The receive operation on the channel is blocked
Signalling done in 5 seconds
Signalling done in 4 seconds
Signalling done in 3 seconds
Signalling done in 2 seconds
Signalling done in 1 seconds
OK, exiting

Program exited.
```

We've removed the waitgroup and instead use a single unbuffered channel which blocks on receive until the signal is sent on the channel. Once the receive happens, after printing some output, the program exits.

### 10.5.5 Closing a channel

There's a limitation with the previous example. Only one receive can take place, so only one goroutine gets the signal. When that goroutine is main, that's fine but if we had hundreds of goroutines blocked waiting to receive, a single *done* signal would not be very useful.

Fortunately, we've another way to signal without data, by closing the channel.

By explicitly closing the channel, we signal that no more data will be sent on the channel. When a channel is closed in this manner, all goroutines which are blocked waiting to receive will return immediately.

In the example which follows, to avoid needing to send an empty struct on the channel, we simply close the channel. It's a drop-in replacement for the previous syntax.

**Example 138 - Signalling by closing the channel**

```
1  package main
2
3  import (
4          "fmt"
5          "time"
6  )
7
8  func main() {
9          done := make(chan struct{})
10         go func() {
11                 for i := 5; i > 0; i-- {
12                         fmt.Printf("Signalling done in %d seconds\n", i)
13                         time.Sleep(1 * time.Second)
14                 }
15
16                 close(done)
17         }()
18
19         <-done
20
21         fmt.Println("OK, exiting")
22 }
23
24 // Go Playground: https://go.dev/play/p/ciX1rKK-XAV
```

In this example, we have five goroutines which are blocked and waiting to receive. After a short duration, we close the channel from within *main*. This unblocks all receive operations and enables the goroutines to return.

**Example 139 - Signalling multiple goroutines by closing the channel**

```
package main

import (
        "fmt"
        "time"
)

func main() {
        message := make(chan string)

        for i := 0; i < 5; i++ {
                go func(i int) {
                        fmt.Printf("G%d is blocked\n", i)
                        <-message
                        fmt.Printf("G%d is returning\n", i)
                }(i)
        }

        <-time.After(5 * time.Second)

        close(message)

        <-time.After(5 * time.Second)
}

// Go Playground: https://go.dev/play/p/vHFbaV5Tyg3
```

Output:

```
G1 is blocked
G0 is blocked
G2 is blocked
G3 is blocked
G4 is blocked
G4 is returning
G1 is returning
G3 is returning
G2 is returning
G0 is returning

Program exited.
```

We used another blocking receive operation in the previous example, with the `time.After()` function. This function waits for the specified duration and then sends the current time on an unbuffered channel. The receive operation blocks accordingly.

When we close a channel if there is data on the channel, in a buffer, it is possible to continue to read the channel buffer even though the channel is closed. But, if we attempt to send on the channel after it has been closed, our program will panic. To avoid this risk, only the sending goroutine should close the channel, not the receiver.

In this final example, we demonstrate how data continues to be read from the channel after it has been closed. And, if you uncomment line 24 and run the program again, it will panic when attempting to send data on the closed channel.

**Example 140 - Reading from a closed channel**

```
1  package main
2
3  import (
4          "fmt"
5  )
6
7  func main() {
8          data := make(chan int, 10)
9
10         fmt.Println("Sending to channel")
11         for i := 0; i < 10; i++ {
12                 data <- i
13                 if i == 9 {
14                         fmt.Println("Closing channel")
15                         close(data)
16                 }
17         }
18
19         fmt.Println("Reading from closed channel")
20         for i := 0; i < 10; i++ {
21                 fmt.Println("Received:", <-data)
22         }
23
24         //data <- 10
25 }
26
27 // Go Playground: https://go.dev/play/p/iedP9__fCzR
```

Output:

```
Sending to channel
Closing channel
Reading from closed channel
Received: 0
Received: 1
Received: 2
Received: 3
Received: 4
Received: 5
Received: 6
Received: 7
Received: 8
Received: 9

Program exited.
```

When reading from a channel, the receiving goroutine can optionally also inspect a second return value which will inform it when the channel has been closed.

```
//ok is false when the channel is closed
d, ok := <-data
```

### 10.5.6 Using range with channels

We can use a *for range* loop to receive from a channel indefinitely. When the channel is closed by the sender, any buffered data is read from the channel before program execution continues.

This approach is useful since in real-world applications we often won't know how much data to expect.

**Example 141 - Reading from a channel with range**

```
package main

import (
        "fmt"
)

func main() {
        data := make(chan int)

        // worker
        go func() {
                for i := 0; i < 10; i++ {
```

```
13                          data <- i
14                  }
15                  close(data)
16                  fmt.Println("Channel closed")
17          }()
18
19          // handler
20          for d := range data {
21                  fmt.Printf("Receiving %d\n", d)
22          }
23  }
24
25  // Go Playground: https://go.dev/play/p/Qdo3mWvKmmg
```

We used an unbuffered channel which means the communication between the worker and the handler is synchronous. The channel wasn't closed until all values had been sent and received.

We could have used a buffered channel instead. An appropriately sized buffer would enable the worker to load the channel with all the values and then close it. Range would still have read all the data from the closed channel before continuing. This modified code example demonstrates.

**Example 142 - Reading from a closed channel with range**

```
1   package main
2
3   import (
4           "fmt"
5           "time"
6   )
7
8   func main() {
9           data := make(chan int, 10)
10
11          // worker
12          go func() {
13                  for i := 0; i < 10; i++ {
14                          fmt.Println("Sending data")
15                          data <- i
16                  }
17                  close(data)
18                  fmt.Println("Channel closed")
19          }()
20
21          // handler
```

```
22          for d := range data {
23                  fmt.Printf("Receiving %d\n", d)
24                  time.Sleep(1 * time.Second)
25          }
26  }
27
28  // Go Playground: https://go.dev/play/p/X5kcrsmOpQW
```

## 10.5.7 Monitoring several channels with select

Let's wrap up channels by talking about *select*. In many applications, it's necessary to handle data from more than one channel. Indeed, we've already encountered such a scenario in our discussion of *context*, where we monitored two channels, one for a ticker, and the other for the cancellation signal.

*Select* is used to listen for communication on several channels at the same time. Its syntax is similar to a *switch* statement, but it's designed specifically for channel operations.

The *select* statement blocks indefinitely until one of its cases can run, and then it executes that case. Each case statement is waiting to read on a specific channel, so when it executes, it means we have received on that channel.

It's a common pattern to use *select* inside of a *for* loop so it executes repeatedly to read multiple values from multiple channels.

We can add a *default* case to the select to allow execution to continue even if there is nothing to receive on any of the channels, meaning the select no longer blocks.

If multiple cases can receive on their channels, only one case body will be executed, which, is chosen randomly.

In our final example, we utilise a select statement within a *for* loop to monitor two signalling channels. All cases in the select statement are executed once. Since the pong case body does not put a value back on either channel, there is nothing further to receive so the default case is executed at the next iteration. Without data to receive on either channel, the default case would be executed indefinitely, so we *return* from main, to exit the program, ensuring it only runs once in our example.

**Example 143 - Monitoring multiple channels with select**

```
package main

import (
        "fmt"
)

func main() {
        ping := make(chan struct{}, 1)
        pong := make(chan struct{}, 1)

        ping <- struct{}{}

        for {
                select {
                case <-ping:
                        fmt.Println("Ping received")
                        pong <- struct{}{}
                case <-pong:
                        fmt.Println("Pong received")
                default:
                        fmt.Println("Continuing")
                        return
                }
        }
}

// Go Playground: https://go.dev/play/p/L8d4dFaVta3
```

Output:

```
Ping received
Pong received
Continuing

Program exited.
```

# 10.6 Summary

Concurrency in Go is a powerful tool which enables us to write scalable and efficient applications. Goroutines and channels make it relatively easy for us to take advantage of multiple processors and handle multiple tasks simultaneously.

In this chapter we've covered most of Go's concurrency mechanisms and, using several simple examples, we've illustrated the principles being applied and key points of learning.

However, asynchronous code is more complicated code and programs which leverage concurrency are undoubtedly more difficult to reason about. Additionally, when we choose to use concurrency, we increase the likelihood of bugs, some of which may not be discovered until runtime. So, before utilising concurrent programming techniques, we must understand trade-offs versus benefits.

# Chapter 11 - Quality Assurance

Next, we're going to discuss *quality assurance*, a catchall term for ensuring the software we create is fit for purpose today and stays that way in the future.

At this point in our learning journey, we should now have the skills and knowledge to write good quality *Go* code. Not just a knowledge of syntax and general concepts, but an understanding of what is happening on the machine itself: knowledge which will help us create more robust Go programs by avoiding some of the gotchas.

But, since we're human, we can't guarantee we won't make mistakes, just like we can't guarantee we'll always write code which performs well enough.

So quality assurance is a toolset - a framework if you like - for us to use, which can help us improve the quality of the software we create, hopefully reducing the incidence of errors and bugs in our code, while ensuring performance is sufficient for the purpose.

Quality assurance starts with softer practices, such as *buddying* or *code-pairing*, together with peer or senior review of code submissions. If we are writing readable and clear Go code - which by now we should be - these simple practices can improve application quality as well as present learning and knowledge-sharing opportunities.

But quality is improved also through the use of tooling and automation. Systems for software version control, for example. Automation for build, integration and deployment of code through test environments and into production, also helps.

In fact, anything that can reduce manual effort and cognitive load, and improve repeatability, adds value from a perspective of software quality, not to mention productivity, but discussion of these techniques take us well beyond the scope of this book.

So, in what is the final chapter of Go Faster, our discussion of quality assurance, keeps the focus on Go and the tooling which it includes, to assist us with *testing*, *benchmarking* and *profiling*.

## 11.1 Testing

We use *testing* to ensure our software does what it should do, and continues to do so, over time, through initial and ongoing development.

Some developers like to develop against tests adopting a *test-driven development (TDD)* approach as they create the feature, while others like to develop the feature and then add tests.

Whatever our preference, we'll probably find ourselves using a test first approach at some point. For example, in the absence of a real-world dependency such as a database, it can make sense to

write the implementation by coding against a unit test which uses a mock database in place of the dependency.

Ultimately, when we add tests, is a matter of preference. Our concern is only that tests should exist before we mark a feature as *complete*, to lock in the expected behaviour of that feature and that the tests are of sufficient quality and coverage, so that if that behaviour should change, our test suite would alert us.

Go's included test tooling is quite minimalist, but it has everything we need. There are several third-party testing packages available which build on top of the standard library, many of which replicate syntax and functionality from other languages. Of course, you should feel free to explore any of these packages, but here we're going to focus solely on the standard library's *testing* package.

### 11.1.1 Test file setup

Testing can be approached from two perspectives, *black-box* and *white-box* and the approach chosen will determine how we set up our test suite.

But what do we mean by *black-box* and *white-box*?

The former only tests exported functionality and behaves like a real user, while the latter *can* test all functionality, including unexported functions - the implementation details - by creating a test suite as part of the same package as the functionality under test.

Testing behaviour *should* cover all the implementations that support the behaviour, which helps achieve similar coverage in the majority of cases.

However, when testing using the white-box approach, it's often easier to replace dependencies with mock implementations, as the functions being tested may directly accept them. Additionally, white-box testing is often better for creating targeted tests, especially for intricate implementation details that require individual test functions to affirm their quality.

Assume we have a package to test which is named `something`. The code sits in a single file, `something.go`. For both testing approaches, we would add a second file for the test functions named, `something_test.go`. The suffix informs the Go tool that it should run the functions in the file during tests.

```
something/
    something.go
    something_test.go
```

Using a white-box testing approach, the test file would be a part of the same package namespace. We can write tests for exported and unexported functionality, and because of this, we can often inject test dependencies directly into the functions under test.

```
package something
```

Conversely, by adopting a black-box testing approach, our test file would sit in a different package namespace, ensuring we can only import, and so write tests for, exported functionality. In many cases, we need to create specific setup and teardown functions to create values using test dependencies, before running the tests themselves, because the functions under test do not accept them directly.

```
package something_test
```

As the codebase expands, black-box testing is typically easier to maintain. Because it concentrates solely on the outcome and not the implementation, it is not affected by any changes to the implementation.

White-box tests that test the implementation may require more maintenance. As development progresses and implementation details are refactored, the tests will often need revision too.

### 11.1.2 Test functions

Regardless of the approach chosen, all `*_test.go` files need to import the standard library's *testing* package and all test functions should be prefixed with *Test* to mark them as tests.

To access the testing tools each function should accept the testing package's type `T`, a struct, which is used to manage the test state and exposes receivers for logging to the screen and signalling failures.

```
package something_test

import "test"

func TestFunctionToTest(t *testing.T) {

}
```

Test functions can be run from the command line individually or as a matching group using a regex search.

```
go test -run=TestFunction
```

`T` implements the `testing.TB` interface so includes all of these receivers. The documentation can explain each throroughly.

```
type TB interface {
        Cleanup(func())
        Error(args ...any)
        Errorf(format string, args ...any)
        Fail()
        FailNow()
        Failed() bool
        Fatal(args ...any)
        Fatalf(format string, args ...any)
        Helper()
        Log(args ...any)
        Logf(format string, args ...any)
        Name() string
        Setenv(key, value string)
        Skip(args ...any)
        SkipNow()
        Skipf(format string, args ...any)
        Skipped() bool
        TempDir() string
}
```

A test ends when the `Test*` function returns. Failure can be signalled in several ways, either with logging or without. Able to continue to run remaining test assertions or a hard stop.

The receivers we'll use frequently are listed below. Some signal failures, others produce output only, and one affects how the tests are run. The included comment briefly explains each.

```
// log to screen, mark the test as failed and continue
t.Errorf()

// log to screen, mark the test as failed, and return
t.Fatalf()

// mark as failed and continue
t.Fail()

// mark as failed and return
t.FailNow()

// log to screen, printed only if the test fails
// or the verbose flag is set `-v`
t.LogF()

```

```
// Run test in parallel with other tests allowed to run in parallel
// to reduce test run time
t.Parallel()
```

### 11.1.3 Test coverage

The Go tool can provide an indicator of test coverage, a percentage indicator of the amount of our code that was executed when running the tests. We can obtain a coverage metric with the addition of the `-cover` flag.

```
$ go test -cover
```

Output:

```
PASS
coverage: 67.0% of statements
ok          github.com/golangatspeed        0.010s
```

A coverage report, with syntax highlighting to indicate which execution paths are covered by tests, and those which are not can be generated with the commands below.

```
// generate the coverage report
go test -cover -coverprofile=cover.out

// opens a web browser with the syntax highlighted coverage
go tool cover -html=cover.out
```

### 11.1.4 Subtests

It's common to write what is known as *table-driven* tests, where the same function is tested in repetition using several test cases, each a specific set of test inputs and expectations.

Since Go 1.7, we've been able to create subtests, by calling `t.Run()` for each test case and there are several advantages to using subtests for multiple test cases.

Each test case is isolated. A call to `t.Fatalf()`, or `t.FailNow()` will not prevent the next test case from being run.

Setup and teardown code can be used to wrap a group of subtests, reducing the time spent on these operations, which are executed if any single one of the included subtests is run.

Subtests can also be run from the command line, either individually or as a group using the regex search filter mentioned already, with the addition of search text representing the contents of the test case.

```
go test -run=TestTime/"in Europe"
```

We'll leave the section on testing with an example of table-driven testing which leverage subtests.

Observe that we're testing using a white-box approach in the example, and we include the test and the function under test in the same file on the Go Playground simply for convenience. In the Github repository we split the code into the appropriate files.

**Example 144 - Example test with subtests**

```
package main

import (
        "errors"
        "testing"
)

var ERR_WOULD_OVERFLOW = errors.New("would overflow")

func Adder(list ...uint8) (uint8, error) {
        var t uint8
        for i := 0; i < len(list); i++ {
                if int(t)+int(list[i]) > 255 {
                        return 0, ERR_WOULD_OVERFLOW
                }
                t += list[i]
        }

        return t, nil
}

func TestAdder(t *testing.T) {
        testCases := []struct {
                name    string
                inputs  []uint8
                wantRes uint8
                wantErr error
        }{
                {
                        "simple total",
                        []uint8{1, 1, 7},
                        9,
                        nil,
```

```
                },
                {
                        "overflow prevented",
                        []uint8{255, 1},
                        0,
                        ERR_WOULD_OVERFLOW,
                },
        }

        for _, tc := range testCases {
                t.Run(tc.name, func(t *testing.T) {
                        gotRes, gotErr := Adder(tc.inputs...)
                        if gotRes != tc.wantRes {
                                t.Errorf("wanted %v got %v", tc.wantRes, gotRes)
                        }
                        if gotErr != tc.wantErr {
                                t.Errorf("wanted error %v got %v", tc.wantErr, gotErr)
                        }
                })
        }
}

// Go Playground: https://go.dev/play/p/2NNWfknTbXC
```

Output:

```
=== RUN   TestAdder
=== RUN   TestAdder/simple_total
=== RUN   TestAdder/overflow_prevented
--- PASS: TestAdder (0.00s)
    --- PASS: TestAdder/simple_total (0.00s)
    --- PASS: TestAdder/overflow_prevented (0.00s)
PASS

Program exited.
```

### 11.1.5 Example tests

We've already mentioned *examples* in Chapter 1 when considering documentation. Examples enable us to include interactive code snippets for package functions and receivers, with the documentation for the package.

They don't contain specific assertions, and they don't use the *testing* package itself, but they're a useful complement to tests, offering us a way to perform comparisons of printed output, without having to capture that output first.

Example functions have an *Example* name prefix, and live in the same `*_test.go` files as tests.

We can test both *ordered* and *unordered* output, and we illustrate both approaches below. For unordered output, the example test passes as long as all the output is included, the order itself doesn't matter (ideal for map output). The second example test, requires an exact match, so the test fails since we omitted the square braces in the formatted slice output.

**Example 145 - Examples with ordered and unordered output**

```
package main

import "fmt"

func List(ceil int) []int {
        var res []int
        for i := 0; i < ceil; i++ {
                res = append(res, i)
        }
        return res
}

func PrintList(list []int) {
        fmt.Println(list)
}

func ExampleList() {
        for _, v := range List(5) {
                fmt.Println(v)
        }

        // Unordered output: 4
        // 2
        // 1
        // 3
        // 0
}

func ExamplePrintList() {
        PrintList([]int{1, 2, 3, 4})

        // Output: 1 2 3 4
```

```
}

// Go Playground: https://go.dev/play/p/RRE5gXK3A87
```

Output:

```
=== RUN   ExampleList
--- PASS: ExampleList (0.00s)
=== RUN   ExamplePrintList
--- FAIL: ExamplePrintList (0.00s)
got:
[1 2 3 4]
want:
1 2 3 4
FAIL

Program exited
```

## 11.2 Benchmarking

Benchmarks allow us to discover and quantify performance. Using benchmarks we can establish baselines, baselines which can be monitored as development progresses, or compared during efforts to improve the performance of a specific part of our application.

Benchmarking functionality is also provided by the standard library's *testing* package. Benchmark functions reside in the same `*_test.go` files as our tests but functions are named with a *Benchmark* prefix. They must accept the `testing.B` struct type which also implements the `testing.TB` interface.

Benchmarks do not run in the test suite by default, they are executed when the `-bench` flag is passed with the `go test` command. This flag accepts a valid regex.

```
go test -bench=.
```

Within each benchmark, the function to be evaluated must be wrapped in a *for* loop. The upper bound of this loop is denoted by `b.N`. This is not a value we set.

By default the go test tool will aim to run the benchmarked function for at least one second, if the function completes before the second has elapsed, `N` will be increased such that the function under test is run more times. This provides a statistically better view of performance by averaging the results of multiple executions.

For longer running functions that may take over a second to complete, it may be desirable to manually increase the benchmarking time so that performance can be averaged over more than a single execution. We can do this by setting the `-benchtime` flag.

```
go test -bench=longfunction -benchtime=10s
```

# 11.3 Profiling

Profiling is the automated process of measuring the performance characteristics of an application, to identify and remove bottlenecks and improve the performance of slow parts of the program.

Code review alone often isn't sufficient to identify performance issues, but fortunately, Go provides tools which we can use to gather information about the performance of the executing application - the profiles - and then analyse that information.

There are two types of profiling we can perform. We can gather data from code which is under test, or, we can capture profile data from a slightly modified application during its execution.

Both approaches allow us to obtain information on specific performance characteristics. Which we target will depend on the problems we need to address, but common targets are *CPU usage*, *heap allocations* and *blocked goroutines*.

## 11.3.1 Obtaining profiles with the test tool

For code which is covered by tests, it's very simple to obtain a profile by modifying the `go test` command. This, for example, would create a profile for heap allocations during the test run.

```
go test -memprofile=heap.out
```

## 11.3.2 Obtaining profiles from a running program

For non-test applications, we need to modify the code slightly so that profiles can be obtained.

In this first example, we can obtain CPU profile information when the program exits, the profile data is written to a file `cpu.out`.

**Example 146 - Modify code to obtain a CPU profile**

```
package main

import (
        "os"
        "runtime/pprof"
)

func main() {
        f, _ := os.Create("cpu.out")
        defer f.Close()
```

```
        pprof.StartCPUProfile(f)
        defer pprof.StopCPUProfile()

        // rest of the program
}
```

It's also possible to obtain profiling information during application execution by creating an HTTP server on a specific port and importing the *net/http/pprof* package. By adding a profiling server to our program, we can get real-time insights into the performance of our application, and identify and optimize any bottlenecks and performance issues as they occur.

**Example 147 - Realtime profiling during program execution**

```
package main

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

func main() {
    go func() {
        log.Println(http.ListenAndServe("localhost:8081", nil))
    }()

    // rest of the program
}
```

During application execution, the HTTP server exposes profiling data at several endpoints, three of which are listed here.

CPU Profile: http://localhost:8081/debug/pprof/profile
Memory profile: http://localhost:8081/debug/pprof/heap
Blocking profile: http://localhost:8081/debug/pprof/block

## 11.3.3 Analysing a profile with the pprof tool

Whichever method we use to obtain the profile, and regardless of profile type, to make sense of the information we need to use pprof.

The *pprof* tool is a command line tool that is used to analyse and visualise profiling data. Pprof offers several ways to visualise the information which include.

- **Text**. Provides function performance statistics in a simple text format.
- **Graphical**. The function call graph, with the box size used to represent execution time.
- **Flame**. The relative contribution of each function to execution time in a horizontal bar graph.

In the example below we specify we want text output, and pass both the profile file `cpu.out` and the compiled application to pprof. The latter is required since profile output omits function names, using their addresses instead. Having the compiled application pprof can cross reference the address to the function name.

```
1 go tool pprof -text ./myprogram.test cpu.out
```

When generating profiles using `go test` the compiled application will not be discarded after a test run. Instead, it is saved with the `.test` extension so it can be shared with the pprof tool.

# Glossary

API - Application Programming Interface
CLI - Command Line Interface
CPU - Central Processing Unit
CRUD - Create, Read, Update, Delete
FIFO - First In First Out
IDE - Integrated Development Environment
JSON - JavaScript Object Notation
LIFO - Last In First Out
NOOP - No Operation
OOP - Object-Oriented Programming
OS - Operating System
REST - Representational State Transfer
Stdout - Standard Out
URL - Uniform Resource Locator