sitepoint

LEVEL UP YOUR WEB APPS WITH GO

BY **MAL CURTIS**

PERFORMANCE, CONCURRENCY, SCALABILITY

Summary of Contents

LEVEL UP YOUR WEB APPS WITH GO

BY **MAL CURTIS**

Level Up Your Web Apps With Go

by Mal Curtis

Product Manager: Simon Mackie
English Editor: Kelly Steele
Technical Editor: Lionel Barrow
Cover Designer: Alex Walker

Notice of Rights

Notice of Liability

Trademark Notice

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: business@sitepoint.com

ISBN 978-0-9924612-9-4 (print)

ISBN 978-0-9941826-0-9 (ebook)
Printed and bound in the United States of America

About Mal Curtis

Mal Curtis is a Kiwi polyglot software engineer currently focussing on Go and JavaScript. He's the founder of transactional email management service Apostle.io, and is a Principal Engineer at Vend, where he helps make beautiful Point of Sale and Inventory Management software (yes, it can be beautiful). In the past he's helped launch Learnable.com, and worked for SitePoint in Melbourne. In his spare time you'll find him attempting geeky pursuits with varying levels of failure, such as quadcopters and sous vide cooking.

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, Ruby, mobile development, design, and more.

To my Dad — for surrounding me with technology from my earliest days, I can never thank you enough. I miss you.

And to Sarah — for your unrelenting support and patience. You're my rock.

Table of Contents

Preface

The Internet is a place of constant evolution and creation. Nearly every day, web developers have new tools available to add to their repertoire. In recent years, though, few have been as influential as the Go programming language. Originally created at Google to solve system administration problems, Go has evolved into a modern, powerful, and well-adopted language. To call Go just a language would be a disservice, however; Go is more than a language—it's an entire ecosystem. From the tools that come with it, to the community of developers that build on it, Go is a force majeure in the web development world and it is here to stay.

So what is Go? Well, it's a combination of a strongly typed programming language, and a collection of tools that make working with the language a pleasure. While many may consider a language to comprise merely the syntax, the tools provided to aid development are just as important—if not more so—as the language itself. I hope that as you work through this book you'll see this for yourself, and learn to love Go as I do.

The topics covered in this book are targeted at web development. While Go wasn't created as a "language for web development," it was produced with such a powerful and diverse standard library that web developers have taken to the language in droves, with many organizations now having Go applications powering their services. In this book, we won't be covering every aspect of the standard library; instead we'll be diving deeply into how to create fast, powerful, and maintainable web applications.

Who Should Read This Book

This book assumes at least a basic understanding of many programming and web development principles. If you're unfamiliar with programming, or lack an understanding of how to program for the Web, you may find the concepts discussed in this book hard to grasp. If, however, you're a seasoned web developer, I hope that by seeing Go in action you'll be inspired to try building "the next big thing" in Go.

Conventions Used

You'll notice that we've used certain typographic and layout styles throughout the book to signify different types of information. Look out for the following items:

Code Samples

Code in this book will be displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park. The birds
were singing and the kids were all back at school.</p>
```

If the code is to be found in the book's code archive, the name of the file will appear at the top of the program listing, like this:

example.css

```
.footer {
  background-color: #CCC;
  border-top: 1px solid #333;
}
```

If only part of the file is displayed, this is indicated by the word *excerpt*:

example.css *(excerpt)*

```
  border-top: 1px solid #333;
```

If additional code is to be inserted into an existing example, the new code will be displayed in bold:

```
function animate() {
  new_variable = "Hello";
}
```

Where existing code is required for context, rather than repeat all the code, a vertical ellipsis will be displayed:

```
function animate() {
  ⋮
  return new_variable;
}
```

Some lines of code are intended to be entered on one line, but we've had to wrap them because of page constraints. A ➥ indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/blogs/2015/05/28/user-style-she
➥ets-come-of-age/");
```

Tips, Notes, and Warnings

Hey, You!

Tips will give you helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings will highlight any gotchas that are likely to trip you up along the way.

Supplementary Materials

http://www.learnable.com/books/go1/

The book's website, which contains links, updates, resources, and more.

https://github.com/spbooks/go1/

The downloadable code archive for this book.

http://community.sitepoint.com/
: SitePoint's forums, for help on any tricky web problems.

books@sitepoint.com
: Our email address, should you need to contact us for support, to report a problem, or for any other reason.

Some Notes

HTTP Requests

In a lot of the examples in this book, the intersting part of what's happening is less about what a browser is displaying and more about the way the browser interacts with a web server. We're interested in seeing what data was sent in the request, such as the path we requested, and what headers we sent. In the response, we're interested in what data comes back, not just in the response body, but also the headers that come back.

In most of these cases, instead of showing a screenshot of a browser, I'll simply show the raw HTTP request and response. It's definitely not as pretty, but it's a concise and clear way to get the data that I'm discussing onto the page. Don't worry, there will still be plenty of screenshots.

As an example, here's what a request and response to the SitePoint website looks like:

```
GET / HTTP/1.1

HTTP/1.0 200 OK
Date: Mon, 01 Jun 2015 09:37:11 GMT
Server: Apache/2.2.22 (Debian)
Last-Modified: Mon, 01 Jun 2015 09:36:07 GMT
Expires: Mon, 01 Jun 2015 10:36:07 GMT
Content-Type: text/html; charset=UTF-8
Connection: close

<!-- Page Content Omitted -->
```

Generally I'll avoid leaving any headers in the response that have no significance to what we're talking about.

You can generate your own requests from the command line by using the `curl` tool (you can download curl[1] if you don't already have it). For example, requesting SitePoint with curl would look like this: `curl -i www.sitepoint.com`. This is a great way to inspect what's really going on under the hood.

Go Get

The **go get** tool is a very convenient package manager, but it's also limited in functionality compared to a lot of package managers. When it fetches a package, it has no concept of version, in other words, it will get the latest version of the code you're asking for. While this isn't much of an issue when you're working on your local machine. It means there's a chance that when you go to run your code on another machine, such as during deployment, or if you're part of a team, the new machine might download a different version of the package. Because of this, code that builds on your machine may not build on another machine because you're running different versions of the packages.

There are a few ways to get around this, and we'll cover the options in a later chapter, but for the moment, be aware that if you are getting build errors that appear to originate from another package, it might be because the version you've downloaded is slightly different to the version I used when writing the code for this book. Whenever I introduce a new third party package, I will also note what version of that code I'm using, so if you're comfortable with Git you can navigate to the library and check out the exact same version.

Formatting

There is a standard for how you should format your Go code, and although the compiler won't throw an error if you format your code differently, it's nice to follow the standards. You can completely automate this process by using the tool `gofmt` that comes with the Go installation. This tool will automatically reformat your Go code for you. All popular text editors will have a plugin that automatically runs `gofmt` for you when you save your files. I highly recommend taking the time to install one for your editor — trust me, it'll be one of the best things you ever do.

[1] http://curl.haxx.se/dlwiz/?type=bin

Imports

Most code examples will only show the relevant parts of the code, and aren't compilable on their own. Don't worry, the code download contains compilable versions of each code example. If you're writing the code as you read the book, the most important thing to note is that you'll need to be importing the packages that are used.

Imports can turn into a hassle, as it's not only a compile time error to use a library that isn't imported, but it's also an error to build code that imports a library that isn't used. You'll quickly find yourself getting annoyed at having to add and remove import lines from you files. Luckily the Go community knows this, and there is a tool called `goimports`[2] that will automatically add and remove import lines from your code as required. Most editor plugins that run `gofmt` will also handle `goimports`.

Guiding Gently, Not Explicitly

In some areas of the code, I will explain a few alternative methods for achieving a goal. Often there will be some trade-offs or advantages for one way or another, and I will attempt to explain these. I do this because frequently there's no right or wrong way to do something. A choice is often a matter of personal preference, or may depend on the specific project requirements. Because of this, I think it's best that I cover a few options, rather than make a choice for you.

Terminal

Throughout this book I'll use the term "terminal" to refer to your command line. Some users may have custom terminals installed, so feel free to use whatever you're comfortable with. If you don't have a preference, or aren't familiar with the concept, all systems will come with an application you can use. Mac users should use Terminal.app, Linux users will also have a terminal as part of their distribution, and Windows users can use cmd.exe.

[2] http://godoc.org/golang.org/x/tools/cmd/goimports

Want to take your learning further?

Thanks for choosing to buy a SitePoint book. Do you want to continue learning? You can now gain unlimited access to courses and ALL SitePoint books at Learnable for one low price. Enroll now and start learning today! Join Learnable and you'll stay ahead of the newest technology trends: http://www.learnable.com.

Chapter 1

Welcome New Gopher

This chapter is designed to give you a grounding in the fundamentals of the Go language. We'll start out by covering installation on various operating systems and write our first piece of Go code, before moving on to look at Go's tools, syntax, and types. Let's get started!

Installation

The first step in writing Go code is to get your hands on the Go tool set, available at https://golang.org/doc/install. The Go authors provide installation packages for Windows, Mac OS, and Linux. You can also install Go from the source code, and if you're on a system other than those with pre-built installers, you'll need to do it this way.

Once installed, you should be able to type `go version` in your command line and see your Go version printed. If there's an error, try reopening your command line software again; otherwise, refer to the Go website[1] for troubleshooting.

[1] http://golang.org/

Alias Golang

Go's main website is golang.org, and the language is quite often referred to as "Golang." If you're having trouble searching when using the term "Go," try using "Golang" instead, as it's a little less ambiguous. They're one and the same.

The Go Workspace

Throughout this book, you'll see examples of why Go is referred to as an "opinionated" programming language, and the way you should lay out your development workspace is one of them. Where other languages might allow you to set up your projects however you wish, Go likes you to do it Go's way, which in this instance is to keep all your code within a specific directory structure inside a directory known as your **GOPATH**. This enables Go to provide tools that let you easily download and manage third-party source code; without it, you'll be unable to use any libraries except for the standard ones that come with Go.

This may be at odds to your regular way of working, but give it a chance. I fought it at first, but have since learned to accept the "Go way," and now find it to be a common sense approach to code management.

Your **GOPATH** will contains several directories, the most important of which is the **src** directory. This is where you'll write your code, and where Go will install third-party source code. Create a directory on your computer now, and place an **src** directory inside it.When you want to write some Go code, it should be in a directory inside **src**. To tell Go what directory the `GOPATH` is, we'll set an environment variable of the same name. If you're unfamiliar with environment variables, they're a way of providing system configuration values, and are available on Windows, Linux, and Mac OS systems.

Mac OS and Linux

To add an environment variable in Mac OS or Linux, you'll need to add it to your Bash profile, which is loaded every time you run a terminal window:

```
echo "export GOPATH=~/Gocode" >> ~/.bash_profile
source ~/.bash_profile
```

This sets the directory **Gocode** in your home directory as your `GOPATH` and reloads the configuration.

Alternative Shells

If you don't use Bash, these instructions may fail to work. If you've installed an alternative shell, such as zsh, consult the instructions for that software or google it. Actually, if you've installed your own shell, chances are you already know how to do this.

Windows

The environment variables menu is available in Windows 8 by right-clicking the bottom-left of the screen from the desktop, selecting **System**, then **Advanced System Settings**, then the **Advanced** tab, then the **Environment Variables** button. You can see it in Figure 1.1.

In Windows 7 and Windows Vista, right-click the **Computer** icon on the desktop or the **Computer** option in the **Start** menu. Select **Properties** and then **Advanced System Settings**, the **Advanced** tab, then the **Environment Variables** button.

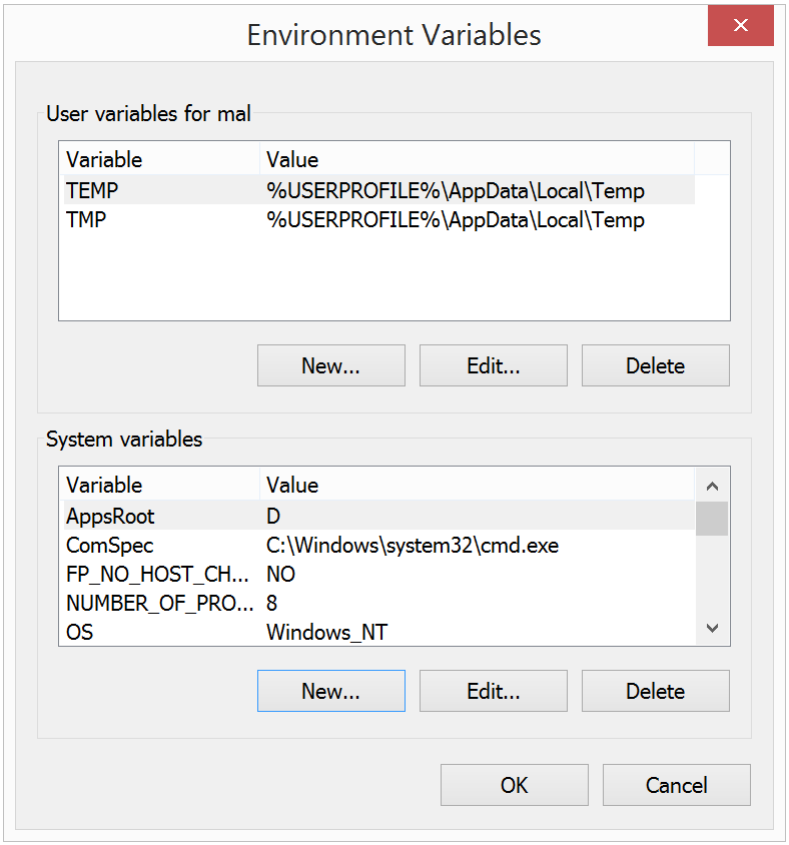

Figure 1.1. The environment variables menu in Windows

Once you're in the environment variables menu, select **New...** in the **System Variables** area and create a new variable called `GOPATH` with the location of your Go workspace as the value, as seen in Figure 1.2. In my examples, I've used `C:\Gocode`.

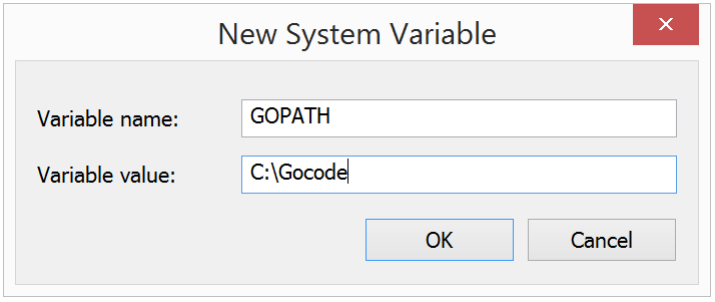

Figure 1.2. Creating a new system variable

Your First Go Code

This chapter will introduce the basics of the Go syntax as well as some of the standard tools we'll be using to write our code. Rather than a comprehensive guide to the Go language, it's an introduction to set you up with some foundation knowledge. As we progress through the book, we'll introduce more concepts when we come across them.

Any good programming book start with a basic "Hello World" example, so let's follow in the footsteps of giants, in order to get our feet wet with some Go. Create a new file, **helloworld.go**, and enter the following code:

helloworld.go

```
package main

import "fmt"

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

Now in your terminal from the same directory, you can tell Go to run your code:

```
go run helloworld.go
```

If your code runs successfully, you should see `Hello World` printed in your terminal. This is an elementary program but includes the basic structure of most Go applications. To start, we declare the package name to which this code belongs. A **package** is the way we group related code, and `main` is a special package that defines code to be executed directly. If you wanted to create a package to use in another Go program, you'd add a unique package name here. We'll cover writing your own packages in a later chapter.

Next up we import the packages that we'll use. Since this is a simple program, it just imports one package: `fmt`. The `fmt` package is part of the Go standard library that comes with the Go tools we've downloaded. It's capable of performing basic formatting and printing operations. Since `fmt` is part of the standard library, it's a very basic import statement. Later on, we'll see how Go handles importing third-

party packages from source control by including just a URL in the import statement; for example, `import "github.com/tools/godep"`.

Lastly we have a function called `main`, which calls the function `Println` in the `fmt` package. The `main` function is the entry point to your application, and will be run when your program starts. `Println` simply prints out a value, and you can pass as many variables as you'd like.

At this point you may have noticed a few characteristics about the Go syntax. Firstly, its parenthesis and curly braces look like many C-derivative languages. Secondly, there's a noticeable lack of semicolons; new lines are commonly used to determine the end of one statement and the beginning of the next. Finally, you might have noticed that it takes a relatively short time to compile or run your code. This is one of the decisions the Go team made early on—that Go would sacrifice a little on the performance side to ensure the code compiles quickly, and that goes a long way towards making the language an enjoyable one in which to develop.

Go Tools

We've seen the first example of the Go tools. We've used the `go` command line tool to run our code. `go run` compiles and runs the code. And if we wanted to create an executable program instead of merely running the program, we could run `go build`. The `build` command will create an executable binary file named after the current directory, and build all the files in the directory; there's no need to tell it which files to build. For example, I have my `helloworld.go` file in a directory `chapter1`. If I run `go build`, I will have an executable file `chapter1` created inside that directory. We can take this one step further with the `go install` tool, which will build the file and then place the executable into your `$GOPATH/bin` directory. If you add that directory to your system's `PATH` environment variable, you'll be able to install your Go program and run it from anywhere in your system.

PATH Environment Variable

Your `PATH` is a list of directories in which your system looks for executable programs. When you run any command on your system, it will search each of these directories (in order) for a command of that name, and run the first one it finds—or complain that you don't have it! On Windows, you can edit the `PATH` settings in the same area you set up your `GOPATH`. On Mac OS or Linux, you can use the same

method as earlier to add `export PATH=$PATH:$GOPATH/bin` to your Bash profile, which will append the directory to the `PATH`.

Basic Types

Go is a **strongly typed** language. What this means is that any variable can only hold a single type of value, so sometimes you'll need to convert between types. Once a variable has been assigned a type, it can only hold that particular type. Go provides a variety of primitive types, along with the ability for you to create your own types that extend these primitives.

Strings

One of the most common primitive types is the **string**. Let's look at how we'd create a new variable that holds a string:

```
myString := "I'm a string."
```

Notice how we assign the string to the variable `myString` by using a colon followed by an equals sign, `:=`. This is how we initialize a new variable. If we want to change the string stored in `myValue` after it's initialized, we can omit the colon. In fact, if we were to include it, the program wouldn't compile:

```
myString := "I'm a string."
myString = "I'm really a string, I promise."
```

If you tried to reinitialize the variable, you'd receive an error that says "no new variables on left side of :=" when attempting to compile your code.

To concatenate strings, you can use the `+` operator:

```
myString := "I'm a string."
myOtherString := myString + " I really am."
fmt.Println(myOtherString) // Outputs "I'm a string. I really am."
```

Commenting in Go

Both single-line and multiple-line comments are available in Go. Single-line comments start with a `//`, and multiple-line comments are surrounded by `/*` and `*/`.

Numbers

Go has a variety of categories for representing numbers. There are two categories of numbers that we'll be looking at: integers and floating-point numbers. An **integer** is a whole number without a decimal component, such as 1, -7, and 42. A **floating-point number**, or **float**, can contain a decimal component, such as 3.141 or -9.2.

Integers

Within each category of number, there are several types. Integers have the most types: `int`, `int8`, `int16`, `int32`, `int64`, `uint`, `uint8`, `uint16`, `uint32`, and `uint64`. Each type is capable of holding different sets of numbers based on how many bits of memory the variable is stored in (8, 16, 32, or 64). For example, an `int8` is capable of holding numbers from -128 to 127, while an `int64` is capable of holding from -9223372036854775808 to 9223372036854775807. Any type starting with `int` is a signed integer, which can represent both negative and positive numbers, and those starting with `uint` are unsigned, and can only hold positive numbers.

If you know the maximum value range you'll assign to a variable you can choose a more appropriate type, since it will reduce the memory assigned to store the variable; however, you'll find that in most instances you'll be using an `int64` or `int` type.

`int` Type Is Special

The `int` type is special, and its size will depend on the architecture of the computer you're using. In practice most systems will be running a 64-bit architecture, and `int` will be the same as an `int64`. This also applies to the `uint` type.

Floats

There are only two floating point types, `float32` and `float64`. Using a `float64` over a `float32` increases its precision, whether that's a larger number or more decimal places.

You cannot perform operations on variables that contain different types. If you attempt to, you'll incur an error along the lines of "invalid operation: a * b (mismatched types int and float64)." You can change between numeric types by performing a type conversion:

```
myInt := 2
myFloat := 1.5
myOtherFloat := float64(myInt) * myFloat
fmt.Println(myOtherFloat) // Outputs 3
```

Type Conversion

To convert a type, commonly called "casting" in other languages, you pass the variable into parenthesis after the target type. This will only work with types of the same category, as these numbers are; if you try to convert between incompatible types you'll get a compiler error. We'll see some more examples of type conversion later on.

Booleans

The **bool** type represents a value that is one of two predefined constants: `true` or `false`. There's little more to say about it than that—other than they're great for conditionals:

```
myBool := true
if myBool {
    fmt.Println("myBool is true")
} else {
    fmt.Println("myBool isn't true")
}
```

Arrays and Slices

Arrays are an interesting subject in Go. Most of the time you'll be dealing with **slices**, which refers to part or all of an underlying array, and is analogous to an array in

most programming languages. Because in almost all day-to-day programming you'll be dealing with slices, I'll skip covering more about arrays, but if you're interesting in learning about the underlying data type of slices, I recommend reading the article "Go Slices: usage and internals" by the Go team.[2]

Slices are defined in the form `[]T` where `T` is the type being stored in the slice. Only values of a single type can be stored in a slice. Let's look at how we can create a slice, and also make further slices from a slice (by slicing it!):

```
mySlice := []int{1, 2, 3, 4, 5}
mySlice2 := mySlice[0:3]
mySlice3 := mySlice[1:4]
fmt.Println(mySlice2, mySlice3, mySlice[2])
// Outputs: [1 2 3] [2 3 4] 3
```

First, we create a slice of type `int` that contains five values. The curly brackets are Go's way of instantiating a new instance of the type that precedes it. You'll be seeing it a lot.

After we've created our slice, we're able to create other slices from it. The colon notation inside the square brackets refers to a start and end index that should be used to create the slice. A single index number inside square brackets refers to a single element in the slice. If you attempt to use an index that's outside the bounds of a slice, you'll get an error along the lines of "panic: runtime error: index out of range" when your code runs. As per most programming languages, the first index in a slice is number `0` rather than `1`.

Runtime versus Compile Time Errors

The compiler understands a lot about our programs, and will often refuse to compile if we try to perform a task that we shouldn't. If you try to access an invalid index on a slice, however, this is a runtime error, since Go is unable to infer at compile time what the size of the slice will be (as they are dynamically sized, they can change size). As a result, you should always be careful about accessing a value in a slice by checking its length.

If we want to know the length of a slice, we can use the built-in `len` function:

[2] http://blog.golang.org/go-slices-usage-and-internals

```
mySlice := []int{1, 2, 3, 4, 5}
fmt.Println(len(mySlice)) // Outputs: 5

mySlice := []string{"Hi", "there"}
fmt.Println(len(mySlice)) // Outputs: 2
```

Looping

To iterate over a slice, we use a `for` statement with a `range` clause:

```
animals := []string{"Cat", "Dog", "Emu", "Warthog"}
for i, animal := range animals {
    fmt.Println(animal, "is at index", i)
}
```

This will output:

```
Cat is at index 0
Dog is at index 1
Emu is at index 2
Warthog is at index 3
```

If we didn't want to use the index value `i`, for example,we can't just assign it and ignore it. We'd receive the error "`i` declared and not used". If there are no plans to use a certain value, it must be assigned to the **blank identifier**, represented by an underscore (_). The blank identifier is how we discard a value that we have no use for:

```
animals := []string{"Cat", "Dog", "Emu", "Warthog"}
for _, animal := range animals {
    fmt.Println(animal)
}
```

Maps

Go provides a **map** type that lets us assign key/value pairs to it. This type is analogous to a hash table in other languages, or an object in JavaScript. To create a map, we assign a type for both the key and the value, such as `map[string]int`. For example, if we wanted to store the years that the Star Wars movies were released, we could create a map such as this:

starwars.go *(excerpt)*

```
starWarsYears := map[string]int{
    "A New Hope":              1977,
    "The Empire Strikes Back": 1980,
    "Return of the Jedi":      1983,
    "Attack of the Clones":    2002,
    "Revenge of the Sith":     2005,
}
```

If you wanted to add another value into the map, we can do so this way:

starwars.go *(excerpt)*

```
starWarsYears["The Force Awakens"] = 2015
```

Note that the `:=` operator is not used.

As with slices, you can ascertain the length of a map with the `len` function:

```
fmt.Println(len(starWarsMovies)) // Correctly outputs: 6
```

Looping over Maps

Looping over maps is very similar to interating over slices. We use a `for` loop with a `range` clause, and this time instead of the index integer as the first assigned value, we get the key:

starwars.go *(excerpt)*

```
for title, year := range starWarsYears {
        fmt.Println(title, "was released in", year)
}
```

If you run this code, you'll see an output along the lines of this:

```
Revenge of the Sith was released in 2005
The Force Awakens was released in 2015
A New Hope was released in 1977
```

```
The Empire Strikes Back was released in 1980
Return of the Jedi was released in 1983
Attack of the Clones was released in 2002
```

If you run it again, the values will output in a different order. It's important to understand that unlike arrays and slices, maps are not output in any particular order. In fact, to ensure that developers are aware of this, maps are ordered randomly when ranged over.

Dealing with Map Values

We can get a value associated with a key by passing the key inside square brackets:

```
colours := map[string]string{
    "red":    "#ff0000",
    "green":  "#00ff00",
    "blue":   "#0000ff",
    "fuchsia": "#ff00ff",
}
redHexCode := colours["red"]
```

It's worth noting that the variable `redHexCode` would be assigned the "empty value" of the map's type if the key didn't exist—in this case, that's an empty string.

If you want to delete a value from a map, you can use the built-in `delete` function:

```
delete(colours, "fuchsia")
```

The `delete` function won't return anything, and also does nothing if the key you've specified fails to exist. If you want to know if a key exists you can use the special two-value assignment, where the second value is a Boolean that represents whether the key exists or not:

```
code, exists := colours["burgundy"]
if exists {
    fmt.Println("Burgundy's hex code is", code)
```

```
} else {
    fmt.Println("I don't know burgundy")
}
```

Functions

We've already seen an example of a function in the Hello World excercise at the start of the chapter. Let's have a look at functions in a bit more depth.

A **function** is defined with the keyword `func`, and can take multiple parameters and return multiple values. Each parameter and return value has to have a type specified; if sequential parameters are of the same type, you can just add the type definition to the last parameter of that type. Here are a few examples:

```
func noParamsNoReturn() {
    fmt.Println("I'm not really doing much!")
}

func twoParamsOneReturn(myInt int, myString string) string {
    return fmt.Sprintf("myInt: %d, myString: %s", myInt, myString)
}

func oneParamTwoReturns(myInt int) (string, int) {
    return fmt.Sprintf("Int: %d", myInt), myInt + 1
}

func twoSameTypedParams(myString1, myString2 string) {
    fmt.Println("String 1", myString1)
    fmt.Println("String 2", myString2)
}
```

In the first example, we see a function with no parameters or return values; this is the simplest function you can create. `twoParamsOneReturn` shows a function that takes two parameters of different types and returns a single string value. `oneParamTwoReturns` takes a single integer parameter, and returns both a string and an integer. Lastly, `twoSameTypedParams` shows how we can take two parameters of the same type and only need to add the type information to the last parameter of that type.

Handling functions that return multiple values is as simple as providing comma-separated variables in which to place the values. We could use the `oneParamTwoReturns` function this way:

```
func main(){
    a, b := oneParamTwoReturns(3)
    fmt.Println(a) // Outputs "Int: 3"
    fmt.Println(b) // Outputs "4"
}
```

A lot of functions will return two values: one for the actual response from the function and a second to indicate whether or not an error has occurred. This is a very common practice in Go and forms the basis for handling errors, which we'll cover later. While you can return more than two values, it's not common to see this.

Pointers

When you pass a variable to a function, it receives a copy of the variable, rather than the variable itself. If you alter a variable inside the function, the code that called the function will fail to see those changes.

Pointers allow you to pass a reference to an object, rather than the object itself. This means that you can alter the underlying object, and other areas of your code will see the updated value rather than a function altering a copy. If you're coming from a language that supports pointers, sometimes called "passing by reference", you'll be familiar with this concept.

Pointers are best explained by this common everyday scenario: You have a banana, a pear, a fruit storage locker, and a cloning machine. You create a variable `myFruit` and place a banana in it, then pass that variable into the cloning machine. The variable that comes out of the cloning machine, `myClonedFruit` then has the pear put into it. Now `myFruit` has a banana and `myClonedFruit` has a pear. Changing `myClonedFruit` has no effect on `myFruit`.

As you might have guessed, this is what happens when we pass a variable directly into a function. It creates a new variable for that function, which is a copy, and altering it has no effect on the original variable.

Now let's look at what happens when we make use of our fruit storage locker. We'll start from scratch, so forget about the previous variables. We create a variable `myFruit`

and place the banana in it. We then place the variable in the fruit locker, send the location of the fruit locker to the cloning machine, and receive a new variable `myClonedFruit`. This time we haven't copied the fruit, we've copied the location of the fruit—the fruit locker. We then put the pear in the fruit locker. We now have two variables, `myFruit` and `myClonedFruit`, which both have a pear in them, because they're both in the same fruit locker.

In this second example, instead of passing a fruit variable into the function, we've passed a pointer to a fruit variable. This pointer points to the address in memory where the fruit is stored, so when we change the value at that address, both the function and the calling code are referencing the same value.

So what does this actually look like in Go? Let's check out an example:

```
func giveMePear(fruit string) {
    fruit = "pear"
}

func main() {
    fruit := "banana"
    giveMePear(fruit)
    fmt.Println(fruit) // Outputs: banana
}
```

This example passes the `fruit` variable into the `giveMePear` function, which receives a copy of the fruit and then assigns it:

```
func giveMePear(fruit *string) {
    *fruit = "pear"
}

func main() {
    fruit := "banana"
    giveMePear(&fruit)
    fmt.Println(fruit) // Outputs: pear
}
```

In the second example, instead of taking a string as a parameter, the `giveMePear` function accepts a pointer to a string—denoted with a `*string`. Inside the function we assign a new string into the value that the pointer references, again using the `*` symbol. When we call the function, instead of just passing in the variable `fruit`,

we take the address of the variable. We do this by prepending the variable with an ampersand, `&`. This tells Go that we have no interest in the variable itself, just a pointer to the variable, which is where that variable is in memory.

At this point, don't worry if you're struggling to fully understand the concept of pointers. We'll see them in action soon, and it is hoped you'll build up an understanding as we continue on through the book.

`nil` Pointers

One element to note when using pointers is that there's no guarantee you'll have a value passed into the function. You could also pass in `nil`, which represents no value (note, this is very different to `false`, which only applies to Boolean values). While it's obvious in our example that we're not passing nil into the function, more often than not you're using variables that have been passed through from other areas of the code, so you must check whether a value is actually passed. If you try to access or alter a nil pointer, you'll receive an error along the lines of "panic: runtime error: invalid memory address or nil pointer dereference". You can avoid this error by checking if the variable is equal to `nil` in situations where a nil value may be passed.

Structs

Go allows you to define your own types, and most custom types you write will be structs. A **struct** is a type that contains named fields. If we continue with the movie example from earlier, we might create a `Movie` type that has some data associated with it:

```
type Movie struct {
    Actors      []string
    Rating      float32
    ReleaseYear int
    Title       string
}
```

As you can see we've created a variety of fields, each with their own associated type. You'll see that the field names start with an uppercase letter. This is very important, and means that those values can be accessed by code outside the package in which the type was defined. These fields are referred to as being exported, and

we'll cover the practical realities of exported and unexported code in a later chapter. While not exactly analogous, you can think of an exported field or method as being "public" rather than "private."

To create an instance of the `Movie` type, we use the curly brackets like when we created slices and maps:

```
episodeIV := Movie{
    Title:       "Star Wars: A New Hope",
    Rating:      5.0,
    ReleaseYear: 1977,
}
```

We now have an instance of the `Movie` type stored in `episodeIV`. We only supplied three of the four fields, so the fourth field `Actors` will be instantiated with the empty value for its type—in this case, `nil`.

We're able to access the fields through dot notation, and can read and assign the fields just like with already instantiated variables:

```
episodeIV.Actors = []string{
    "Mark Hamill",
    "Harrison Ford",
    "Carrie Fisher",
}
fmt.Println(episodeIV.Title, "has a rating of", episodeIV.Rating)
```

Type Methods

You can add your own methods to any types you define. This is as easy as defining a function that has what's called a receiver type. A **receiver type** is similar to a single parameter definition in a function, but is declared just after the `func` keyword but before the function name. If we wanted to define a method that formatted a movie title for display, along with the movie's release year, we could create a `DisplayTitle` method like so:

```
func (movie Movie) DisplayTitle() string {
    return fmt.Sprintf("%s (%d)", movie.Title, movie.ReleaseYear)
}
```

```
func main() {
    episodeV := Movie{
        Title:       "Star Wars: The Empire Strikes Back",
        ReleaseYear: 1980,
    }
    fmt.Println(episodeV.DisplayTitle())
    // Outputs: “Star Wars: The Empire Strikes Back (1980)”
}
```

You’ll see that before the function name, `DisplayTitle`, we declare a receiver type of `(movie Movie)`. This gives us a variable `movie` of type `Movie` that we can access just like any other function parameter. This might take some getting your head around, as it means there’s no concept of “this” or “self” as you see in other strongly object-oriented languages.

Object-oriented Programming in Go

Receiver functions are the basis for object-oriented programming in Go, in that a type with various methods is analogous to an object in most object-oriented languages. I must stress that Go doesn’t provide a lot of the traditional object-oriented programming constructs, such as inheritance, but this is by design. Go provides various methods for creating complex data types, such as embedded types and interfaces, which we’ll cover soon.

Receiver types can also refer to a pointer of a type. Without a pointer, any changes that you make to the instance will fail to make it any further than the end of the method. A straightforward example of this might be a counter type with a method `Increment` that adds one to a count field:

```
type Counter struct {
    Count int
}

func (c Counter) Increment() {
    c.Count++
}

func (c *Counter) IncrementWithPointer() {
    c.Count++
}
```

```
func main() {
    counter := &Counter{}
    fmt.Println(counter.Count) // Outputs: 0

    counter.Increment()
    fmt.Println(counter.Count) // Outputs: 0

    counter.IncrementWithPointer()
    fmt.Println(counter.Count) // Outputs: 1
}
```

As you can see when we call the `Increment` method, which only receives a copy of the instance, the field `Count` is not incremented outside the scope of the `Increment` method. The method that receives the pointer to the object, however, alters the same `Counter` instance, and we see the change outside the `IncrementWithPointer` method. It's interesting to note that we immediately took the address of the `Counter` instance with the `&` symbol, and that we were able to call both the method with and without the pointer receiver type. This will not work in reverse: if we failed to take the address and just said `counter := Counter{}`, we'd be unable to call the method `IncrementWithPointer`.

Exported and Unexported Code

When writing your own packages and structs, you need to decide what methods, functions, constants, and fields will be available to code outside the package. An **exported** value is available to code outside the package, **unexported** values are not. If you're unsure whether or not some code should be exported, it's a good rule of thumb to make it unexported, then change the name if you realize it's required outside the package:

```
// myUnexportedFunc is not available to code outside this package
func myUnexportedFunc() {
}

// MyExportedFunc is available outside this package, though
func MyExportedFunc() {
}

type MyExportedType struct{
```

```
    ExportedField   string
    unexportedField string
}
```

Summary

That covers our introduction to the basics of the Go syntax and types. In the next chapter, we'll look at creating our own types, how to create interfaces that serve as a contract for other types to implement, and how Go handles third-party libraries.