Course Overview
Course Overview

Hi, I'm Nigel. I'm a longtime technology geek, and I've written best-selling books on things like containers in Kubernetes, and I've got a ton of courses here at Pluralsight. But let me tell you, I am excited to bring you this course, Go Fundamentals. Because you know what? Coding is a massively useful skill, but as well, Go is easily one of the most useful languages you could ever learn like it is powering just about everything these days. Anyway, as you progress through the course, you'll do everything from the, well, mandatory Hello World, but all the way through decision making with conditionals, the basics of functions, working with data using variables, and things like slices and maps. And you know what? You'll even learn how to create your own custom data types with structs. Plus, and this is a big one, you'll get your head around concurrent programming with goroutines. So, yeah, a really exciting agenda. And you know what, whether you've coded in other languages before or you're an utter nube, this course has got so much for you. And then, when you're done, I promise you'll have everything you need to press on at your own pace. And you know what, maybe even start using Go in some of your own or your work projects. So, join me on this journey as we add programming in Go to your expanding list of skills.

Course Introduction

Course Introduction

Right then, I'm Nigel, and welcome to this course, Go Fundamentals. And, I am stoked that you've chosen it, because honestly, Go is an epic language to learn. It's modern, it's relatively simple, and it is playing a huge role in shaping the future. Now, on that, just a few examples so you know I'm not just making stuff up, but, the who's who of major cloud infrastructure projects are all written in Go. So, Kubernetes, heck yeah, that's written in Go, Docker, Terraform, CoreDNS, Hugo, CockroachDB, Dropbox, SoundCloud; honestly, Go is the engine behind all of them. But you know what, outside of big projects and open source stuff like that, real companies hiring real people are using Go. So, we're not messing about, Go is a proper language, and it is one of the best tech skills to have in today's world. Magic? Well, let's go and see what we've got on the agenda.

Course Agenda

So, first and foremost, this is an entry-level course or entry-ish. And that's probably as clear as mud, so let me explain. I'm obviously not going to be telling you what a keyboard or a mouse is, and I'm not going to be showing you how to save files. But, what I will be doing is explaining Go stuff as clearly as is humanly possible, or at least for me. So, I'm assuming you know your way around a computer, but I'm also assuming that you've got zero clue when it comes to programming in Go. So, yeah, go on, a quick example. Let's say I'm explaining arrays and slices. Okay, well, I'll explain the lists of items that are all the same type, so maybe all integers or maybe all strings or something, and I'll compare it to, I don't know, a class register with student names or something like that. But I'll also be saying, hey, if you're familiar with other languages, it's pretty much the same as X, Y, Zed or X, Y, Z. Anyway, I'm

going to be as clear and concise as possible. So, if you're new to it all, you get the fundamentals. But if you've got some experience, I'm not going to be boring you with the uber basics. Well, you know what, for everything we learn, we'll cover the theory and we'll get our hands on. But look, I'm waffling. This is what we'll cover. We'll learn the history of Go and things like why it was created and how to install it. Then, we'll train in the footsteps of all the programming greats and we'll write our very own Hello World program. After that, we'll start getting into the weeds. We'll see how to work with variables and constants. We'll get our heads around functions. We'll do a bit of decision making with conditionals. We'll see how to loop. We'll look at lists with arrays and slices. I guess more lists with maps. And then, we'll round things out by looking at creating our own custom variable types with structs, and then the icing on the cake, with an intro into concurrent programming in Go. And honestly, when we're done, well, on the downside, you will be sick of my voice, but on the plus side, you'll have a proper idea of how Go works and you'll be ready to power on with further learning. Okay, well a quick mention of who I am and a bit about the courses repo on GitHub.

About Me and About the GitHub Repo

All right then, two quick things, who am I, and where can you get the code for the course? So, I'm Nigel, I'm a proper techaholic, and I'm from the UK, though really, I live on the internet, only, not in a creepy way. Anyway, this is me just about everywhere on the net, and I am more than happy to connect. And, really quick, I've worked in tech since my late teens, and I feel like I've done everything pretty much from writing code to Windows and Linux admin, networking, storage, and these days I mostly work with containers in Kubernetes, but that's enough about me. Throughout the course, we'll be using sample code files, and they're all on GitHub in this repo. Now, honestly, the stuff we'll be doing is so easy, you can type everything in by hand if that's your thing, but, if you want to, you can download it all with the following commands. Now then, you will need to install Git, but it's dead easy. Just Google, how do I install Git? And then, once you've got it, just clone the repo like this, and then switch into the directory, and from there, you've got all the code in the folders, just like on GitHub. Great, but seriously, don't stress if you don't know Git or GitHub, you don't need to, I'll be keeping things so simple to follow along with. But yeah, I reckon that's it. Come on, let's crack on with the course.

Introducing Go

Module Intro

Right, then. The plan for this module, I think, is to talk a bit about Go as a language, like what do we mean when we say things like it's a compiled, statically-typed systems language that does its own garbage collection, it infers, it supports concurrency, all of that jargon, what does it mean? Now, look, if you already know all of that, magic. This course will be a walk in the park. But if that actually sounded like I was speaking Klingon or something, that's alright as well. We'll explain everything as we go. And you'll be putting it all into action, so you'll pick it up. And you know what? By the end of the course, you'll be like, wow, I actually know what all

that means. Anyway, after that, we'll see how to install Go, which is super easy. Alright then, let's try and explain Go.

Go Overview

Okay, so a quick 5 or so minutes getting our head around what Go is and maybe its major features. Well, I reckon first up we need to address the gopher in the room. This is the Go gopher, and it's the official mascot of Go. And no, it doesn't have a name. It is simply the Go gopher, but it is a big part of the community, and you're going to see it everywhere. Now interestingly, it was designed and drawn by Renee French. She's a well-known writer and illustrator, and she's of interest to us because she's married to Rob Pike, one of the three original creators of Go, seriously. So look, Go was created as a language at Google by Robert Griesemer, Ken Thompson, and Rob Pike. So, Rob Pike, one of the Go founders, his wife designed the gopher. Anyway, work on the language started in something like 2007. And as the legend goes, while the three founders were waiting an eternity for the C++ program to compile, they decided to sketch out what C or C++ would look like if they designed it, or do you know what? Maybe what it would look like if it was designed then in 2007, rather than like 30 or 40 years earlier. Well, fast forward a couple of years to 2009 when that C++ program had almost finished compiling, and they released the first publicly available version of Go, and they made it open source under a BSD style license. So, at the time that I'm producing this course, Go is well and truly established. I mean, okay, it's properly young compared to the likes of C and C++, both of which are 70s and 80s babies. But you know what? It's even young compared to the likes of Python and Java. Now you can look at it being young in two ways. Yes, It's obviously not as mature as some of those other languages, but hey, they were all written way back in the last century. So no, Go is not as mature as some of those, but at least it was written in this century. And when you consider how much the world has changed with the advent of cloud and the likes, then Go as a pretty modern language, it's well suited to this century's demands. In fact, you know what? The who's who of major cloud infrastructure projects are all written in Go. So Docker and Kubernetes, that's right, both written in Go. Etcd and Terraform, you've got it, written in Go. So yeah, it's young, but it is widely used and very much production-ready. Now, I think it's fair to say that Go was designed as a systems language. I'm using air quotes there. So, it was designed for things like building operating systems and infrastructure projects like Docker and Kubernetes, yeah? But do you know what? It's evolved over the years to be a solid choice for things like web services and other high-level apps. So today, Go is a solid all-rounder, and it is a cracking language for you to learn. Now then, as a language, it is a bit like C and C++, only fortunately, it is way simpler. Like as a language, it's cleaner and more concise than C, plus, it does its own garbage collection, which C doesn't. Now, I'm aware that I'm throwing around the odd buzzword. And if you're not sure about what some of those mean, don't sweat. We'll be explaining them all later on, and you'll even see most of them in action. But for now, garbage collection is just the language itself taking care of memory management. You know, like deallocating memory and the likes automatically so that you as a developer, well, I guess you can just think about other stuff, yeah? Anyway, Go only has 25 keywords, and that really isn't many. It's definitely less than C and Python, and it is way less than Java, so technically easier to learn. Obviously as well, it's cross-platform, so you're good to go on Linux, Windows, macOS, BSD, and even good old Solaris. It is a compiled language. That means the nice, easy-to-read code that we're going to write, that gets compiled down to machine code

that runs without an interpreter. Buzzwords again. Okay, tell you what. My first programming language, FoxPro, way back in the day, that was an interpreted language. Jargon that meant the programs I wrote in FoxPro needed a separate piece of software called an interpreter to run them. So I'd write programs for customers, and then for the customers to run them, they'd basically need to install the FoxPro interpreter to run my programs, and I was never a fan. Well, it's not like that for Go programs. They compile down to native machine code that run on their own without the need for an interpreter. And not only is that a whole lot simpler in my view, it also makes them a whole lot faster. Go is a strongly-typed language. This means it can be picky when it comes to variables and the likes. So, maybe, I don't know, if you try and add an integer with a floating point number, that's not going to work. The compiler is going to throw a type mismatch and your code just won't run. And I will say, if you're not used to this kind of behavior, I'm not going to lie, it can take a bit of getting used to. In fact, you know what? Strong typing is a bit like someone that's really picky with grammar, you know, the self-appointed grammar police. I'm sure most of us know at least one person like that, and, of course, we love them. Well, the strongly-typed nature of Go means it's really strict about assigning the right types to variables. And yeah, this means we're probably more likely to get errors at compile time, but it does make us write good, clean, predictable code. Oh yeah, sticking with types, Go supports type inference. So, we can write something like this on the screen, and Go's going to look at it and say, okay, 5.5, that looks like a floating point number to me, so it will assign the variable type as a float. But then maybe if we assign 10 to a, well, this time it's going to assign it as an integer. Now, if you compare this type inference to something like C where we have to explicitly state the type, you might be able to see how Go can be simpler and more concise. Well, do you know what? I reckon that'll do for now. Go is a modern, clean, simple, and concise language. It's great for systems programming, but it's also a really good all-rounder. We said it is strongly typed, and it compiles down to machine code, magic. Now let's go and see how to install it. Sorry, I had to do that at least once. Come on, let's go!

Installing Go

Alright, installing Go. And, I'm just going to level with you, for the most part, this is so easy, it's probably not even worth me recording the lesson, but I am, so I'll make it quick. We already said that Go is cross-platform, and I actually meant a couple of things when I said that. I meant it compiles programs that run natively on different operating systems and architectures, but, I also meant you can install Go itself, so, the language, the compiler, and the other tools, also on different OS's and architectures. Now, installing on Linux, Windows, and Mac are easy. It's a clickety-click wizard for Mac and Windows and a walk in the park command line for Linux. For anything more exotic, sorry, you're probably installing from source, which is basically jargon for harder, you know, no nice wizard or whatever. Anyway, look, I'm on a Mac right now, and I literally typed install Go into my favorite search engine, and I ended up here at the official Download and install page. So, I'm just going to click this big Download Go button, wait for that to download, and then I'll fire it up. A few clicks and the likes, and check that out, all done. So I think a quick check in the terminal, and I am ready to rock and roll, told you it was easy. So you've now got the Go compiler and all the other tools that you need to write Go programs and build them into executable apps. Now you've also got the standard library packages that you'll get familiar with later in the course. Now, I know that was just a Mac install, but it's practically identical on Windows, and like I said, it is a simple

command line operation on Linux. And, if you are following along, bring it on, because you are ready to rock and roll with Go. Speaking of which, next up, we're writing code.

Hello World

Module Intro

All right, this is where the good stuff starts. We'll start the ball rolling with an overview of Go modules, packages, and source code files. Then we'll roll our sleeves up and get our hands on. We'll start out by creating a workspace and initializing a Go module, and then we'll start writing our first program. And it's okay if some of this is jargon. We're about to demystify it. Anyway, as we're writing that code, we'll dip into the notion of functions; we'll circle back to our code and import a package that lets us print text to the screen; we'll see how to run, build, and install the fully compiled app; and then we'll wrap the module with configuring VS Code for Go. Now, it might look a bit complex, but believe me, it is not. There's basically three things, writing the code, building it as an app and running it, and then configuring VS Code. And you know what? The VS Code bit's totally optional. You don't have to, and you can totally follow the course without it. But, I recommend you do, because if you're getting serious about Go, it's probably the most popular tool available, and it's free. But you know what, talk is cheap. Let's crack on.

Modules, Packages, and Source Code

Okay, so a bit of housekeeping before we crack on. First up, a version check. Everything we're going to do is going to work fine in Go version 1.14 and newer. You know what? It should even work as far back as 1.11. You just need the $Go11MODULE variable set on for anything before 1.14. And the other thing you need to know is the structure of Go packages and modules. So a Go application consists of one or more source code files. All files relating to a particular program get grouped into a package, and packages get grouped into modules, which I know is a ton of jargon, so let's look at it from the inside out and then from the outside in. From the inside out, you write your app as a bunch of source code files that live in the same folder on your computer. Well, in Go we call that folder and source code a package, and that means when you compile the app, you're actually compiling the entire package, so all the source code files in that directory. Now, we'll see it in just a second, but because all these different files are part of the same application package, then variables and functions and stuff that are in them are accessible to all the other files in the package. Anyway, a module is then a set of related packages that you release together. Magic. But in the hands on, we'll be doing it the other way, from the outside in, so we'll create the module first, then we'll create the packages, and finally the source code files. So in a second, you'll use the go mod init command to initialize a new Go module in an empty directory. Then you'll create a subfolder for your first package, and finally, you'll start writing code. Now, I totally get that it can look a bit complex and maybe even a bit daunting if you're new to this, but I promise you, you'll set this stuff at once, and for the rest of the course, you'll just be coding and barely even know that the package and module stuff's even there. But you know what, I literally said talk is cheap, and I'm still talking, so let's go and get our hands on.

Initializing a Go Module

Alright, then. We're about to create and initialize a Go module and then write some source code for our first package. So the first thing you need is an empty folder somewhere on your computer. I don't know, it's probably easiest to create a folder called go in your home directory, but you don't have to. I mean, I'm down in the depths of some arcane folder structure that I've been using for years for my Pluralsight courses. So feel free to choose anywhere. It just needs to be an empty folder on your machine that you'll use to store all the programs we'll write. Well, once you've got that, you need to initialize it as a Go module. So making sure you're in that module folder you just created, run the go mod init command like this. Now then, this bit here can pretty much be whatever you want, but if you plan on ever publishing the code in the module, then it's got to be a path to the repository where you'll eventually host the code. So for me, the code for this course is hosted on GitHub in this repo. And look, if you're unsure about this or you don't think there is a cat in hell's chance of you ever publishing this code, which there probably isn't, I mean it is just learning code from the course, then maybe just use your name or something like this. It really doesn't matter. Okay, so that's created this go,mod file here in the root of your module folder. And if we look at it, yeah, we can see it lists the name or the path to the module, and then this is the version of Go that created it. Now, the whole idea of modules is that how GO tracks dependencies and versions. So, as you add dependencies to your apps, they'll appear in here with the required versions and the likes, but right now, that's your Go module initialized, and we're ready to write some code.

Writing Your First Program

Remember we said modules contain packages, and packages contain source code files. Well, we've got our module, so let's create a new folder inside of it for the hello-world package. And see how the name of the folder is the name of the package. Well, any code you write in this folder is going to be part of that hello-world package. Well for now, I'm going to write the code in Vi. If you're on Windows, I recommend you probably go with Notepad, and for sure we're going to get more professional in a second. I just want the first program we write to be the simplest thing in the world with no fancy tools distracting us from what's actually going on. Anyway, I'm calling the file hello.go. Right, the first thing every Go program needs is a package declaration. Now calling a package main makes it special. This tells Go to compile this code that we're going to write as an application and not a shared library. So, actually, an application's something that maybe you would double-click to run and it does something, whereas a shared library generally only runs when another application calls it. Anyway, look, next up, we want a function, and we're calling this one func main. Be careful how you pronounce that. But func main's special, just like package main, because at runtime, so when we eventually run this as an application, the first thing that'll run will be func main inside of package main. And sometimes we call this combo the application's entry point. Basically, without these two, the application won't run. Now, you know what, functions are pretty much at the center of everything you're going to do with Go. So I want to give you a quick heads up on function basics before we go any further.

Functions Primer

Right, then. Functions are at the heart of modularizing your code, and pretty much all the code you'll ever write in Go is going to be inside a function. Like just about every Go program has loads of functions. In fact, it's pretty much one function per, well, function or maybe feature. Like if you program takes some input from the user, maybe it performs a conversion on that input and then returns the converted output to the user, you're probably looking at three functions, so one for each feature or function that the program executes. Now, we'll see it as we go, but functions and packages are the two major ways that we modularize and reuse code in Go. Now, at a high level, every function starts with the func keyword, followed by a name. And for the most part, we can name them anything we want that is except for name, which we just said is special. But, I highly recommend you give your functions meaningful names. Well, we can pass and return values to a function in parentheses, and then the actual code of it goes in between curly braces. And again, I'm really sorry about this, but if it's jargon right now, honestly it's fine. We'll be doing it really soon. But that's the basics of functions. But you know what? For our simple app, we're only interested in this main function here. So we already said that func main is special. It's the first block of code that'll execute in any Go application, so we call it the entry point. And in fact, actually, if you're writing a shared library and not an executable, then it doesn't get a main function. But we are writing an executable, so func main is our entry point, and it's special again because it doesn't take any arguments, and it doesn't return any values. Well actually, you know what? When func main exits, the entire program exits, and it returns an exit code, which, of course, follows the standard practice of 0 indicating success of the application and a non-0 code indicating some kind of error. But for us right now, that's the very basics of Go functions and func main. But you know what? Right now our function's empty, so let's go and add some code.

Importing Packages

Okay. The stuff we've got so far is pretty much the structure of a Go program. All that's missing is the actual program itself. Well, as this is our Hello World package, we're going to be printing some text to the screen. And we do this in Go with the Println function from the fmt or the FMT package. So, up at the top here outside of our function, we need to import the fmt package. Now, well actually two things, feel free to pronounce this however you want, fmt, FMT, even format, everyone knows what you mean. But the main thing to learn is we're importing a package of functions to be able to use them in our app. So to be clear, fmt here is a package in the standard Go library that gets installed in your machine when you install Go. And like any package, it is a collection of functions. In fact, give me a second here. Okay, yeah, this is the official fmt package hosted on GitHub. And then we can see if I highlight here, the Println function. So, by importing this into our package, we'll be able to call on this Println function here. So I'll tell you what, let's jump back to our function, and we'll add this line here. And all it's saying is from that fmt package that we just imported, use this Println function, which we just saw, to print this text to the screen. And that's our first program right there. Make sure you save it, and we're ready to see it run.

Running and Compiling Go Apps

Okay. So there is a few ways that we can run our app. We can just run it, we can build it, or we can install it, and they're all dead simple so we'll do them all. Now, probably the one you'll do most is just run it. So in a terminal making sure we're in a directory where the app is and as long as you've got Go installed, just go run and then the name of the file. And there we go, Hello Pluralsight. And if that's your first Go program, seriously, congratulations. Now what happened there was that Go compiled the app into a temp directory somewhere. It ran it and then it cleaned up. And honestly, you'll do this a lot, especially while you're developing and testing your apps, but when your programs are ready to ship, you'll either build them or install them. So go build here takes the same file name, or if you're in your packages directory, you can just go with go build on its own or you can add a period and it'll compile the app as a binary executable file for your operating system that you can just run. So you know what, this time I'll run it without the name of the file because I'm in its directory. Now, if we list the directory, see how we've got a new file called hello-world. Well, that's the compiled application ready to run. Oh, and it's called hello-world because we're in the package folder called hello-world. So maybe if we had multiple source code files here making up the app and we ran the build command, they'd all get compiled into an executable binary with the name of the package. Well, I'll tell you what, if we run that executable, there we go again, Hello Pluralsight. In fact, you know what? You can even run it from Finder or Windows Explorer like a normal app because it is. And like I think I said, this is useful for building apps that you're ready to ship to your customers. Well, the final thing we mentioned was go install and this does the same as go build, only it places the compiled binary in a folder called go/bin in your home directory. So I should have a file here, which I do, and like we said, it's called hello-world in the go/bin folder of my profile. Now you can change the location where it drops the file by setting the GOBIN and the GOPATH variables. I'm showing the commands here, but actually I'm cool with the default location. And seriously, that's our first program working and compiled as a self-executing application. Now, yeah, go on then. I want to just clarify something in case you're new to coding. When I say compile, I mean Go taking the programs that we write so our source code files and building them into a machine code app that'll run like we just showed. Okay. Well everything we've done so far is well and good, but I'd say probably all of you are going to write and run your code in an integrated development environment like Visual Studio Code. Yeah? Other IDs do exist and you can totally follow the course with other environments, but I'll be rocking for the rest of the course with VS Code, and unless you've got a reason not to, do you know what, I recommend you probably do as well.

Visual Studio Code

Right, then. Let's see how to get VS Code and how to configure it for Go. Now, a couple of things right off the bat. VS Code's just a tool to help us with Go, so I'm not going to dwell on it. We'll get it configured, and we'll crack on with the course. The other thing, what I'm about to show you might change slightly in the future, and there's not a lot I can do about that, but for the most part, any changes will be minor, and what you see here is still going to apply and help you configure VS Code. Alright then, VS Code is a free IDE, that's integrated development environment. And do you know what? The download and install is so simple, I'm not even going to show you. Just download the version for your OS and crack on with

the installer. When you're done with that, fire it up. Well, it looks pretty professional, yeah? But all we've got right now is a framework. First order of business is to install the Go extension. So I'll hit the extension's icon here and type in go at the top. This is the one we want, so we'll install it. And if it recommends any updates or tools, install them. And then when that's done, I think the last thing to do is add a workspace over here, and that's just the root folder of the module you created before. Oh, yeah, you might be asked to trust the workspace and maybe install additional tools like gopls or whatever. I'm going to say yes. This might take a second as well. But when it's done, this is a module with the hello-world package and then your source code file. In fact, see how the code's colored. That's because the Go extension recognized this as a Go program. Magic. Well, that's VS Code ready to go. Time, I think, for a quick recap before we crack on digging deeper.

Recap

Magic. You've just written a Go program. So you initialized a brand new Go module, and we've not really got into modules yet, but I'd say for now, that Go's built-in dependency tracking system. But once they're installed, they'll pretty much just sit in the background, for most of this stuff we'll do. Anyway, within that module, we created a package called hello-world, and we wrote a program. So, as we can see, source code belongs to packages, and packages belonged to modules. And it'll sink in later if it hasn't already. But looking at our code, we started by saying every Go program that we want to compile as an executable application needs a package main and a func main. These are what we call the applications entry point, and therefore, the first bits of code to run when the program starts. This line here imports the func package and makes all of its functions available to our code so that down here in the main function, we can call on the Println function from the fmt package and use it to display text to the screen. With the code all written, we then use the go run command to run the app. And we said that's probably how you're going to test your code while you're developing it. And I'm sure we'll see it later, but if we make typos or whatever, they'll get flagged and we'll have an opportunity to fix them. Anyway, when you're happy with your program and you think it's ready to ship, go build is the command that you'll need. This compiles your code into a binary executable that runs on its own on other computers without the need for the Go compiler. And I think that was pretty much it. So with these basics ticked off, let's start digging deeper with variables and constants.

Working with Variables and Constants

Module Intro

Okay, check that out for a module title. Total opposites, yeah? So variables are designed to, well, vary, yeah? Whereas constants, not so much. Anyway, here's the plan. We'll look at the basics of variables and how we create them. We'll look at some of the different types of variables Go supports, and then we'll do a bit of a sidestep, showing the common short assignment method. Now that's buzzwords at the moment, but you'll get it when we get there. After that, though, we'll look at pointer variables, and we'll see how they let us pass variables by either value or reference. And of course, again, that's more buzzwords, but we'll

demystify everything as we go. Anyway, look, we'll round things out with a look at constants, then how we can access environment variables on your system, and, of course, we'll do a quick recap. Now, look, to help with the demos, we'll use a simple program that shows info on a Pluralsight subscriber, so maybe the name, which course they're watching, and whereabouts in that course they're up to, I don't know, like, maybe module 4, clip 2 or something. But the thing is right. It's just a daft, oversimplified program whose only purpose in life is to introduce you to variables, o it is not supposed to be production-grade code. Anyway, look, we know by now that talk is cheap, so let's get coding.

Declaring Variables

Okay. We already said that Go is a statically-typed language and we said that's jargon for, well you know what, it means a few things actually, but for now, it means when we declare a variable, we've got to tell Go what type it is like is it a text string or maybe a floating point number or something. Anyway, look, we've got the framework of a program here, it's actually in the variables folder of the course's GitHub repo and I think it's called var-framework.go. Maybe if you want to clone it locally and work from that, and if you do, the commands are here and you'll need git installed to be able to do that. But you know what, you don't have to do that so maybe feel free to just type the code into your editor, or you know what, even copy and paste it from the file on GitHub. It's all good as long as you've got that code in your editor. Well, if we declare variables up here at the package level, so outside of any function, we've got to do it with the var keyword. So no, Go doesn't let you use non-declaration statements at the package level. Okay well, for now we go var and then I like to list them out on their own lines like this. So each line here is starting with the name of the variable and then the type. Now, on the topic of names actually, I'm not sure if I said this already, maybe I did, maybe I didn't, but variable names have got to start with a letter and they can't be any of the 25 Go keywords that were shown here on the screen. So I mean you can't call a variable var or func, or to be honest, any of these 25 words on the screen. As well as that though, they can't contain spaces and no special characters like math operators and the likes. Oh, do you know what, and of course look, do yourself and the universe a favor and use common sense like for the love of all things good, give variables meaningful names and preferably short names. And as well, you know what, if the variable name is long and it contains more than one word, then capitalize the first letter of each word just to make things more readable, camelCase here. Do you know what? And on the topic of this, for now right, always make the first letter of a variable lowercase because and I don't really want to get here now, but we'll find out later in the course that making the first letter of a variable a capital letter, that makes that variable visible outside of the package it's defined in, but you know what, that's a bit advanced for now, we'll get to it later. Well look, it is possible to list variables of the same type on the same line like this. So name and course are strings and then module and clip are integers. And then actually declaring variables like this where we're not assigning them any values, they'll be initialized with 0 values. So the integers will get actual zeros here and then the strings will be assigned an empty string. In fact, do you know what? Let's add some code to our main function here to see if that's actually true. Now, before we run this right, these are just a couple of Println functions like we've already seen, only this time within the parentheses, but outside of the double quotes, we're referencing the variables we created. Well, if you're in VS Code, you can hit Run Without Debugging up here and it'll show you the results at the bottom of your screen, and you know what, feel free to do

this with all of the examples if you're using VS Code. But if you're using something different, just open a terminal in the directory where you create it and save the file and you will need to save the file, but then just do a go run like this. I've called mine vars.go, yours might be different. But look, that's our program and actually, yeah, it's not so easy to tell with the strings because they're just empty spaces here, but the ends are clearly initialized as zeros. So 4 variables, each assigned a name, and of course, a location in memory, but importantly, initialized with 0 values, oh, and each one has got a type. And actually speaking of types, that's where we're going next.

Variable Types and Type Conversion

So we can check the types of variables and stuff in Go through something called runtime reflection using the reflect package. So, let's go and import that up here, and then let's add these two statements down here. Now, again, that just prints ln functions, only this time, we're calling the type of function from the reflect package as well. So this one here is going to print the type of the name variable and then this one the type of module. Well, tell you what, save that, and we'll give it a run. Okay, name is of type string, and module is of type int. But yeah, I mean we knew that, right? We called it out when we declared them. Well, Go actually supports something called type inference, and it's pretty cool because it lets us write shorter code. So, if we change the variable declarations up here to be this, see how we're no longer declaring the types, but we are assigning values. Well, Go was clever enough to figure out that Nigel Poulton and Getting Started with Kubernetes are both strings and then that 4 and 2 are integers. Although, you know what? I'm saying it's clever enough. Let's save that and run it to see if it actually is. Magic. Take a second to look at that. Okay, now look, I'm not a fan of initializing multiple variables on the same line. I just think it's easier to read if we list them out on their own lines like this. Now then, a couple of things. These double slashes here are how we add line comments in Go. Basically, anything we put after them are comments and get ignored by the compiler. But the thing is, they're really important for good code, especially if you work on a team where, of course, it's important for your teammates to understand the code that you've written. Well, as well though, Go is really picky about types. Like you can't add or subtract a string from an integer, and that's a bit of a daft example. I mean, how would you do that, right? It makes sense that you couldn't. But, you can't even mix and match integers and floating point variables when you're doing math operations. Well, look. To show this, I'm going to change the variable declaration here so that module is actually a string. Now, I know it looks like an integer, but because I'm putting it inside of double quotes, I'm telling Go, no, store it as a string, and you'll see why in a second. Well, this code here will try and sum that module variable with the clip variable. Let's give it a try. Okay, no joy. And that's expected, right? Even though module looks like an integer, it's a string, which is where conversion comes to the rescue. So the strconv package has a bunch of functions that do type conversion, and we know the crack by now, right? We import the package up here. We'll add this code here, which, yeah, okay. It might look a bit complex if you're new to this. But believe me, it's not. And you know what? It's a good way to get you thinking about some of the stuff that we're going to cover in more detail later. Well, the main part of the code is here. No, yeah, okay. We've imported the strconv package, and we're using this Atoi function that converts ASCII strings to integers. So, we're feeding it our module string, and it will return the equivalent integer. Only, it gives a return code as well, and we have to do something with that return code. So, we're saying this new iModule variable here, assign that the integer that

the Atoi function is going to return. And then this err variable, assign that the return code that the Atoi function will return. Well, then we're doing some basic error checking and saying, so long as the return code's a 0, which indicates a success here, well, as long as we get a success, do this calc here, and print the answer. And yeah, if it seems a lot, don't stress. The point is, we're using a function from an external package to perform a type conversion, but as well, we're just handling it with some good old-fashioned error checking. And we'll see a bunch more of this as we crack on, so it will all fall into place as we go through the course. Well, I guess we should save that and check if it works, which it does. So the Atoi function from the strconv package converted the ASCII string 4 to the integer 4. Then after that, the calculation worked because, obviously, we can sum two integers. Marvelous! Well, I've dropped this little gem in here, so I should probably explain it.

Short Declaration

Okay, quickly, back to variable declaration. So far, we've only talked about variable declaration outside the functions at the package level. And you know what, declaring them here makes them available to all functions in the entire package, so global in scope. But if we declare them within a function, which we'll do a lot, we can use a shorthand, more concise method of declaration, like we did here. Now, declaring variables within a function makes them local, so only available inside the function they're declared in. But that's okay for us because right now, we've only got one function. Anyway, if we declare them here, like this, then we can get rid of these up here. Now, this might seem a small thing declaring them like this, but it's the most common way, and you'll see it everywhere. In fact, if we jump over to the kubernetes project on GitHub here, Kubernetes is written in Go, yeah, well, in this package, if we search for the var keyword, so the long way of declaring, nothing, not a single occurrence. But if we look for the short assignment statement, there's loads, so this is the really common method. But back here, like we just said, it only works inside of functions. Oh, and obviously, if you're not in a position to be assigning values to your variables yet, you'll have to use the long form of the var keyword. Well, I'll tell you what, let's throw in another variable to indicate whether the viewer has completed the course. So, this one is a true or false Boolean, and we're assigning it false for now. Well, let's save that and try to run it, the emphasis being on try. Okay, check that out, no joy; Go is not liking the fact that we've created the courseComplete variable, but then not used it. I guess, it doesn't like waste. Only there's a but, or a caveat, you can totally declare a variable at the package level up here and then never use it. But if you're declaring it inside of a function, sorry, guys, you've got to use it. Well, I'll tell you what, for now, we'll comment that out and try it again. Okay, it works this time. So we've switched our variable declarations to inside the function, and we've adopted the short assignment method. Next up, we'll learn about pointer variables.

Values and Pointers

Right, come on, let's start with some jargon. Go passes arguments by value, not by reference. And what, pray tell, does that mean, Nigel? Well, on this line here, we're passing the module variable to the Atoi function from the strconv package, only we're not. What actually happens behind the scenes is that Go makes a copy of Module and passes the copy

to the function. So real quick, when we create a variable, Go allocates it in area and memory, and let's say the address of that memory is 0xaa, and it's the course variable with the value Getting Started with Kubernetes. Well, when passing this as an argument to a function, Go makes a copy of it and passes the copy to the function. See how it's got a different address but the same value? Well, this means we've got two copies, and I suppose we could say it is a form of immutability, like any changes made by the function only affect the copy at 0xbb. But this is only the default behavior, and we can absolutely change it. Well, no, actually, I should say, we can get around it, and we do that with pointers. So, as an example, if we add this line here to our code, by putting the ampersand here in front of the course variable, when we run the code, we'll see the memory address of the variable instead of its value. So we'll give that a save, and let's try it. All right, there we go, hex whatever, yeah? That's the memory address of the variable. Anyway, point is, as the name suggests, are special variables that point to other variables. You know what, it's probably easiest if I just show you. So, we've got an existing variable called course, and we know that it is storing the value Getting Started with Kubernetes. It's a great course, by the way, and you should most definitely take it. Anyway, this line here creates a new variable called ptr, and then the asterisk before the type makes it a pointer variable. Then the &course bit on the end says point it to the memory address of the course variable. So, if we add this line here, obviously, it's printing text and variables to the screen, but you know what, let's step through it. We are printing this text here that says "Pointing to course variable at address," and then, of course, we want it to print the memory address. So we do that by printing the contents of the ptr variable, which, remember, because it's a pointer variable, is the hex address of the course variable. Stick with me. We're also then saying, which holds this value, and then we're wanting to print the value of the variable, so we do that by referencing the ptr variable again, only this time we actually dereference it with the asterisk. So this time it'll print the value stored in &course, which should be Getting Started with Kubernetes. Oh, tell you what, let's give it a try. Oh, and remember to save your changes first. All right, magic, that looks good. However, I totally get it that it can be confusing, especially if it's new. So, for the sake of those who are confused, and it's totally all right to be confused, most people are when they're first learning pointers, but winding it back, variables are addresses in memory that store data. Usually, we store a useful value, but pointer variables store the memory address of other variables, so they're useful for referencing other variables. Well, as well as that, in code, the asterisk performs two functions. First up, if we declare a variable with an asterisk like we did here, then that asterisk makes it a pointer variable. But when referencing pointers in code, slapping an asterisk in front of it dereferences it, which is jargon talk for, instead of telling us the memory address of the variable you're pointing to, actually tell us the value inside the variable you're pointing to. Woo, so this gives this, and this gives this. Well, finally, putting an ampersand in front of any variable will return that variable's memory address. And you know what, we'll see it really soon, but pointers are really important when we start using functions, and actually, you know what, let's quickly set the scene for that.

Passing by Value

Right then, I think this code's getting a bit cluttered, so just a quick bit of tidying up. We're done with the reflect and strconv packages, so we'll lose them. We are going to use the name and the course variables, but you know what? We'll lose the rest because remember, Go doesn't like it when we create a variable inside of a function at least and we don't then use it.

And you know what, we'll just lose all of this here. So, basically, we're at the bare bones of an app. So, what we're going to do here is write a simple function to change the value of the course variable. And we'll get properly into functions in the next module, so don't overthink things right now, we're just looking at how variables get passed around. Anyway, in our main function here, let's print out the current value of course. And look, I'm just adding a line feed character on the end here to make the output on the screen easier to read. Well, next, let's call to a new function called updateCourse and we'll pass it the course variable. Now then, we've just done this, Go doesn't actually pass the course variable, yeah, it makes a copy and it passes that. And this is crucial. So keep that front and center of your thinking. Well, now to write the function. So in the function signature here, and that's just this top line that we're writing, we give it an input, that's the course variable or copy, yeah. And we tell it the type, which is a string. And then, every function returns an output, which for us, that will be an unnamed string. Basically, we're putting a string in and we're getting a string out.
But remember, don't overthink this yet, we'll come properly to functions next. Let's add some statements to do some work. So this line here is assigning a new value to course. This line's printing that to the screen, then we're telling it to return the update to value, and that's our function. Take some input, change it and print it to the screen, and return a value. So I'll tell you what, back up here we'll add another Println to check whether or not the change we made in the function actually stuck. Now a couple of things. First up, program flow is like this, we start at package main, import a bunch of other packages, and jump into func main. Remember from right at the beginning, package main and func main are the programs entry point, so they're always the first things to run. Next, we initialize a couple of variables and print them to the screen. And it's all pretty orderly so far, basically starting at the top and stepping through line after line; however, this call here to the function switches flow to the function down here. We then step through that one line at a time, and when it's done, flow returns back to here, and we run this line before the program exits. Cool. Well, the other thing I wanted to mention is this line here. Notice how we're just using the equals operator and not the full short assignment. That's because we're not initializing a new variable, we're assigning a new value to an existing one. Well, let's see if it runs. Okay. So, current course is Getting Started with Kubernetes, we're changing it to Getting Started with Docker, but then it's back to getting started with Kubernetes. But that's expected, right? The updateCourse function made changes to a copy of the course variable, not the original. In fact, the original was unchanged, and we can see that here. Okay, fair enough. But what if we actually want to change the original of the actual variable? I'll tell you what, no sweat. We're going to do that next.

Passing by Reference

Okay, to get this updateCourse function to change the actual value of the course variable, so instead of a copy, we need a pointer, which, if you remember from before, is going to be a combo of ampersands and asterisks. So, we're initializing course here, and we're assigning it the value, Getting started with Kubernetes. Then, we're printing it to standard out. But then, as things stand, this call to updateCourse is creating a copy and working off of that. Well, if we lash an ampersand on the front here, instead of sticking a copy of the value, Getting started with Kubernetes onto the stack, it'll stick a pointer or a reference to its location in memory. Well then, down here in the function, we stick an asterisks here before the string. Remember, doing that makes the variable a pointer. Then we need to stick an asterisks here

when we're changing it to Getting Started with Docker. So what this is doing is telling Go to assign Getting Started with Docker, which is another epic course, by the way, but assign this new value to the location in memory that the course point of variable is referencing. Oh, yeah, another one here, if we don't, we'll get the hex memory address again, and another one down here on the return, and then I reckon, yeah I reckon that should work. So this time around, we should actually change the original value. Well, moment of truth I guess. Like we ever doubted it. So, current course is Getting started with Kubernetes. We're updating it to Getting Started with Docker, and the changes stuck. You know what? That folks, is pointers and passing by reference, which can be very useful. Well, next on the cards, constants.

Constants

So this will be nice and short because constants are pretty much the same as variables, except, I guess, for the obvious. Constants are immutable, so they never change, whereas variables, well, these most definitely can and do change. Anyway, I'm cleaning this up so that we've got pretty much a new program, and I'm going to say that as consts.go. Now then, if your package imports and things go away like mine just did there, it's probably because you're using VS Code with the Go plugins and you've imported a package that you're not actually using in your code yet. Well, we actually are going to use func, so I'll put it back. Okay, well, constants are declared in Go with the const keyword, and it's pretty much the same as the long form variable definitions that we saw earlier. Though actually, this is the only way to declare a constant, so there's no shorthand way like we have inside of functions for variables. Well anyway, look, this one is defining probably the world's most famous constant, the speed of light, which if you didn't know, travels at pretty much 186,000 miles per second, or I guess 300,000 kilometers a second, if you prefer kilometers. Kilometers, kilometers, I don't know. Well look, this will print it to the screen, and I think we'll give that a save and run. And we're good. Simple stuff, yeah? Now look, really quick, just to prove that constants cannot be mutated, this line is trying to change it. So speed it up a tad, yeah, actually to Warp 9 in miles per hour. And I know I'm mixing miles per hour and miles per second, but you know what, who cares? But I'll tell you what, let's try it. Now, no joy. As we said before, constants can't be changed. Okay, well, you know what? Time for a quick look at how we interact with environment variables on your system before we wrap the module with a recap.

Accessing Environment Variables

Right. A super quick look at environment variables with Go. Now first up, we're in a new program file here and well, environment variables are well, they're variables here or they're settings on your host system. So for me right now, that's my office Mac because that's where I'm running the demos from. For you, it'll be whatever machine you're following along on and it's pretty much the same whether it's Linux, Windows, or Mac. Now to get access to environment variables, we're going to leverage functions from the OS package. Actually, it gives access to a ton of OS stuff, so not just environment variables, but we know the deal by now. It gets imported up here. Anyway, the environment function here lists all our environment variables on our system, so let's save that and see. Now this time the file is called env.go. Okay, well here, it's a proper mess, but it is a list of all of the environment

variables on your system as key value pairs, but you know what, I'll tell you what a simple loop here like this that'll iterate through them and then put each one on its own line and look, we'll get into loops later so don't worry about it. It's just going to go through that list of environment variables with each one on its own line. So we'll give that a save and go again. Oh, miles better. But I'll tell you what, let's jump back to the variables program that we've been working on. Mine was saved as vars.go, but there is a fresh copy in the variables folder on the GitHub repo called envvars.go and you can work from that if you didn't save it from earlier. Anyway, look, first things first, we've got to import the OS package. Then down here, let's change the name variable so that, instead of assigning it this string literal here where it's my name with a capital letter at the beginning of each of my I guess first name and last name, instead of that, let's assign it the value of the currently logged on user. Well, to do that, it'll be os.Getenv. Oh, actually before we do that, see all of the options that we've got here, so env, pagesize, pid, a bunch of host system stuff, yeah. Well we want Getenv, and then because I'm on a Mac, I want USER. On Windows, it's USERNAME. But do you know what? It's all simple stuff, it's just an environment variable on your system holding the name of the currently logged on user. Oh, and of course we're assigning the output to the name variable here. Well, I'll tell you what, give that a save, and then in the terminal here, well, you know what, first up, I'm going to show you the value of the USER environment variable. Okay, nigelpoulton with no capitals or spaces. Well, let's see if the program pulls that and it did, nigelpoulton like that instead of with capitals and a space because it grabbed the value from the environment. Spot on. Well, of course, we're only just getting our feet wet, totally feel free to have yourself a play around with some of the other stuff available through the OS package. But do you know what, as far as this set of lessons on variables goes, we're done and it's time for a quick recap.

Recap

Okay, I feel like we've covered an absolute ton, so I'll keep this as sharp as I can. We learned that we can declare variables with the var keyword, and we can even do that without initializing them with a value. But, if we do that, Go automatically assigns them a default value. We saw that strings defaulted to an empty string, integers to a 0, and I think we saw that Booleans get a false. Well, we said that variables declared at the package level are global in scope, so, available to all functions. And also though, when declaring at the package level, so outside of any function, the only way to do it is with the var keyword. However, if we're inside of a function, then the more idiomatic way is with the short form operator, and to be fair, that's the way we'll do it most of the time. We also saw that Go supports type inference. So, if we do initialize a variable with a value, well, Go's clever enough to figure out what type it should be. We also said that Go passes variables to functions by value and not by reference. So, instead of passing the actual variable itself, Go creates a copy with the exact same value as the original, and it passes the copy to the function. But do you know what? That's not always what we want, so we looked at pointer variables, and, these are how we can force a function to work on the original variable instead of a copy, and we call that passing by reference this time, rather than by value. Then we switch to constants, which, honestly, are pretty much the immutable sibling of variables. They look, smell, and feel a lot like variables, only once you create a constant, you can't change it. Well, then we wrapped the module looking at how Go can access environment variables from a host system and use those values in programs. And that was about it, though, of course, we've only covered the

basics, and you should feel free to take some time to play around, make a bunch of mistakes yeah, and generally try things out. Well, next on the card, and this is a good one, functions.

Working with Functions

Module Intro

Right then, functions. And if you've been following along, I've kind of bigged them up already, and rightly so, because they are a big deal. As well though, if you're new to coding and maybe it's your first encounter with functions, I recommend you buckle up, because this is gonna be, yeah, one heck of a ride. So we'll start the ball rolling with why we have functions, kind of set the scene. Then we'll get into a bit of detail. We'll start out at looking at how we define functions in Go, so, like, what does the syntax look like? Then we'll get our hands on, and we'll probably start out with func main that we've said every program has, but we'll also create our own. And then we'll round things out with some slightly more advanced stuff. Now, I don't mean anything hard here. We're a fundamentals course, yeah, so we're not going to be rocking it with things like callback functions and closures. But we might give them a mention later in the course. The thing is, right, the aim of this module is to get you comfortable with the basics of functions and able to start writing your own. And then this will obviously have you set up for the more advanced stuff later. Now, one last thing, functions. They are a massive topic, and there's no way we can do them proper justice in a course like this. But don't worry, we will be giving you more than enough to get started. So, come on, let's do this.

Why Functions

Okay. So why do we have functions? Basically, they let us write clean, tidy, reusable code. So let's say that this is 150 lines of code here. I mean obviously, I know it's not, but you get where I'm going. Well anyway, let's say that at three separate points were doing, I don't know, let's say some text conversion, maybe converting to title case. And we'll assume that the code to do that is 10 lines long. Again, I know it's not showing us 10 lines on the screen, but you get my point, we're repeating the same code over and over again in different places. Now, not only is that wasteful, it's also prone to error. A much better option is to just write it out once and then every time we need it we make a call to it or a reference. Well, that's the essence of functions. Take a block of code that does a job, in our case, convert some text into title case, write it out once as a function, and then every time we need it, we call out to it. And you know what, as simple as that, we go from writing out 10 lines of code 3 times, to writing it and testing it only once. And look, again, it's a totally naughty example, but it does demo the point. Functions exist to let us reduce, not only the number of lines of code we write but also the amount of testing we do and the amount of mistakes we'd potentially make. And you know what, it is slap bang right at the heart of coding in Go. And you'll see this all the time as you progress, but Go programs, they're basically a collection of smaller modular functions. Okay, well you know what, that's the mega high level, let's add a little bit of detail. Architecturally speaking, we'd give a function an input, the function does something with that, and it returns an output. So for our title case example, we'd have a

function, and maybe we'd call it title case, but you know what, you can call it pretty much anything you want, but we pass it a text string, the function performs its magic, and it returns the same text as an output. Only this time, formatted as title case. And actually, yes, that is proper title case propositions like with don't actually get converted. But you get the point right? It takes an input, performs a task, returns an output. And you know what, that kind of job, so changing text into title case, it's exactly the kind of thing that functions are designed for. So performing a discreet operation that can be reused throughout the program. So instead of writing out the same block of code, time and time again with the risk of typos and all the other mistakes we can make, just write it out once as a function, and any time you need it, make a call to it. Sweet. Well, coming up next, we'll have a look at how to write functions.

Function Syntax

Right, we start defining functions with the func keyword, followed by whatever we want to call it and a set of parens. And if it looks familiar, it's because we've already seen it with the main function. However though, any code we want the function to execute, which in the examples we've been citing is converting text to titlecase, well, that goes inside a pair of curly braces. Okay, well that's kind of the basic structure, but we're not done yet. In the little picture we just saw, we gave the function an input, and it turned out an output. So, the parameters we passed to a function go here, and that's what we actually call them, parameters. So, I guess that means somewhere in our code we have got a variable holding the text we're going to convert. And then, when we make the call to the function, we pass that variable as an argument. Here, we're calling our function as part of a Println statement, and then the argument that we're passing gets represented here in the function signature, which actually is what we call this line here as well, the function signature. Anyway, look, starting at the top, we've got a string variable holding the value containers on aws wavelength. Then, we call to the Println function from the fmt package, which in turn makes a call to our function called titlecase. And as part of that, it passes it the text variable, or actually, it passes a copier. Remember, Go passes arguments to functions by value rather than by reference. Okay, well next up is our actual titlecase function. So we've declared it with the func keyword, and we've given it a meaningful name. Then, inside the parens, we'll list any parameters passed to it. So, we passed it the text variable, which is a string. Now, it's possible to pass multiple parameters to functions, and we'll see all of that later as we crack on with the course. But, however many we pass, we have to tell the function what type they are, so for us, a string. Well, then we tell it the type of the value we'll return. That will be a string as well. Remember, we're feeding in a string, we're going to convert it to titlecase, and spit out a string. Well, I guess that takes us to the body of the function, which contains the code to convert the contents of the text variable into titlecase. And then last, but not least, we use the return keyword to end the function and return the converted value back to the caller, so back to Println. I'll tell you what, well, was there anything else? Oh yeah, do you know what? Be careful where you put these curly braces. So they've got to go exactly where I'm showing here. Like, I don't know if you put the opening curly on its own line. Believe me, the compiler is going to introduce you to the pain function. Never mind. Oh, do you know what? Actually, we can also return multiple values to the caller, but to do that, we need to enclose them in a set of parens as well. Oh gosh, you know what? We can name returns and do loads more. We are literally only scratching the surface. But do you know what? I do think for us right now, that is enough on the theory. Let's go and get our hands on.

Writing Your Own Functions

So, let's ease ourselves into this. So we've got a shiny new program and we're importing the fmt and strings packages. And you know what, as we're learning functions, let's write out our own main. So it's the func keyword, obviously, it's called main, and it gets a set of parens. Now the main function's a bit special, yeah. I mean we already know it's the program's entry point, so it gets called automagically. But for us right now, it doesn't even accept any inputs, and it doesn't return anything either, so its signature's super simple. Obviously, though, we do give it some curly braces, and then the code we write goes in here. Well, for this demo, and look, it is just a demo, remember, but let's implement a quick mod of the title case example we just hashed out on the slides. So we'll declare and initialize a couple of variables here. Feel free to use your own, of course, but let's print them to the screen. Only, let's do that after we've modified them through a function. So, we're calling the name of the function converter, and we're passing it the two variables. And, yeah, we're calling it from within Println. Magic. Well, if we move down here, and notice, we're outside the main function. Well, this is where we'll define ours. Okay, so we've got to call it converter because that's what we called it up there. It's receiving two strings, and it is returning two. Now, you might find that it's a good idea to write your return line now so you don't forget it. Okay, well, to actually do the converting, I'm going to use a couple of functions from the strings package. So, we're converting the author to uppercase and the course to title case. And you know what, let's go from the top. We're declaring and initializing two string variables, but importantly for us, both of them are holding lowercase values. Well, we're passing them to our custom function called converter, and then we've rigged things up in the function signature to receive them, and we're also going to return two strings, so input and output. Then, of course, within the function, we actually do the conversions. Well, let's see if it works. Magic, uppercase author and title case course. Now, a couple of quick things. When calling the function, back here, we're passing it the author and course variables. But, down here in the signature, we can totally assign them different names if we want. So, I don't know, let's say, well, s1 and s2. Now, look, because of the order we pass them in, author will be assigned to s1 and course to s2. And remember, we pass by value rather than by reference. So we're not passing the variables themselves; we're passing the values in them, and then, of course, we're assigning them to two new variables with these names. So you know what, even if we had kept the same names, they wouldn't be the same variables; they're copies. Anyway, if we do this, we need to change these lines here and the return. And, okay, go on, you know what, seeing as how I'm feeling adventurous, let's give the returns different names as well. Now, look, honestly, in no way am I saying you should do it like this. I mean, to me, it is messy and a recipe for confusion. And as well, your teammates are probably going to hate you for this. I mean, look, we're passing them in this author and course, referencing them inside the function as s1 and s2 and returning them as str1 in str2. I mean, it's horrible and it's unreadable, but it can be done, or, you know what, at least I'm saying it can. Let's give it another try. Yeah, okay, it does work, but like I say, I'm not saying it's recommended. Well look, up next, we're going to say how to write functions when we have no idea how many arguments are getting passed.

Variadic Functions

Quick question. What do you do when you don't know the exact number of arguments that will be passed to a function? Answer, create a variadic function. Sorry, what's that, Nigel? A vari-what-ic function? Yeah, I know, a proper buzzword, but the clue is in the vari bit of the name. Variadic functions can be called with a varying number of trailing arguments, and you know what, if that still sounds complicated, it really isn't. So, look, we've got some code here. It's actually in the functions folder of the course's GitHub repo called vary.go. Anyway, we've got a package declaration, an import statement, and a main function. Now what we're going to do is create our own function to calculate Max Verstappen's highest Formula One championship finish since 2015. So, to do that, we'll create a new variable here, and assign it the return value of a function that we're about to write. And, look, if you think about it, that's pretty cool, yeah? Using a function to declare a variable. Anyway, we'll pass that Max's six previous championship results, and we're doing it as six ints, and that's so we can show how functions can deal with an unknown number of input parameters, because, I guess next year it'll be seven, yeah? Well, then a call to Println to print his best finish. Now, for a custom function. So, it's func and then the name, and then for us the crucial part when telling it what to expect as input parameters, we put ellipses before the type, and it's these ellipses that tell the function to expect any number of ints. Okay, well, we'll also return an int that's going to be his highest finish, but you know, I'm getting ahead of myself. For now, the way this works under the hood is that the input parameters get stored as a slice of ints. Now, I do know we haven't covered slices yet, so, just think of them as a list of ints, so, for us, a list of the six championship finishes that we're passing in, and then, because it's a list, we'll use this loop to range over it and determine the best finish. And again, I know, we haven't covered loops yet either, but honestly, the detail doesn't matter, it's just some code that reads the values in the list or the slice here, and it figures out which value is the lowest. So, just for clarity, we create the best variable here, best, it's called best, yeah, I'm not saying it is the best. Anyway, we create that, and we give it the value of the first item in the list. Remember, computers start counting at 0. Then, we pull in every other value one at a time, and if it's lower than the value currently in best, we update it. And, you know what? Honestly, if loops and the likes are a bit new to you, no sweat, we'll cover all of this in more detail later. Anyway, after we're done ranging over the slice, we return the value in best to the caller that assigns it to our best finish variable back here in main, and we print it to the screen. Shall we try it? Oh, look at that. Best championship finish at the time of recording is third place. Though honestly, right now he's looking like this season he'll either win it or come second, but the thing is, for us in this course, I reckon that's enough on functions. Though, honestly, like I've been saying, we are only tickling the surface, there's so much more. But for us, as a fundamentals course, that is more than enough to keep us going. Let's do a quick recap.

Recap

All right, that was a quick tour of functions in Go, the emphasis being on quick. But, I need to stress a couple of important things. First up, you really do need to understand the basics of functions because they are absolutely at the center of everything you're going to do in Go. So, if you've completed the module and you're still a bit unsure, I reckon you need to have a play around, maybe even watch the module again because it really is important you're comfortable with them. As well though, we've honestly only scraped the surface, there's

like higher-order functions, closures, user-defined functions, loads more. But it's all good, they all build on the fundamentals you've learned here. Anyway, functions are brilliant for writing modular reusable code. And we said that all Go programs are lots of small functions, each one doing a specific job but working together to create a useful program. We also looked at the syntax of writing functions. They start with the func keyword, they need a name, and we list the input parameters and outputs inside a pair of parentheses, including types. Then, the actual function code itself goes inside a set of curly braces. We also looked at variadic functions that use ellipses to handle situations where we don't know exactly how many input parameters are going to be passed. Okay, sweet. Well. if you think you understand most of that, and I hope you followed along with the examples, then up next, we're getting into conditionals, so ifs, whats, and buts, kind of.

Working with Conditionals

Module Intro

Right, then. Working with Conditionals. And I get it if for some of you, that's jargon. However, if you've programmed in other languages, I know you'll know what we're talking about. It's like if/else, else/if, all of that stuff. However, if you're brand spanking new to programming, it's about making decisions in code. So, for example, maybe your program asked the user for input. Maybe, I don't know, what their age is. And then your program does one thing if the user is under 18 years old, but it does another if they're 18 or over. Well, conditionals, like we'll see, let your programs do exactly that. If the user's over 18, branch out onto this set of code, if they're younger, branch out somewhere else. Well, for us, we're going to be learning if and switch in Go. We'll start out by looking at the basic syntax of if statements, then we'll get our hands on with some basics, and we'll dig a bit deeper. Once we're good with if, we'll look at switch case. And again, we'll cover the syntax, get our hands on, and dig a bit deeper. Finally, we'll round out the module with how to use if to handle errors, and then we'll do a recap. Well, let's get rocking and rolling!

"If" Syntax

The humble, but immensely powerful if. And let's just bust some jargon here. If statements let those evaluate conditions, and based on whether or not that condition is true or false, branch out and execute specific code. So, yeah, like we just said a second ago actually, if a person is 18 or over, execute, whatever, code A, yeah, whereas if they're younger, execute code B. Now then, on the jargon front, we are evaluating conditions, and that's where the term conditionals comes from. As well though, you might hear the term branching. Well, like I just said, if a user is 18 or over, branch out onto the orange code here on the slide. If they're younger, we branch to the blue. The if statement evaluates true or false conditions. And if you like your jargon, I'm talking Boolean true or false, so that means using Boolean operators. So we're not talking just a string here with the words true or false in it or, I don't know, even an integer of 0 or 1 representing true or false. No, remember, Go is really strict when it comes to types, so we'll be dealing with a Boolean true and Boolean false, and, well, we'll see it in a second, so don't stress. Well, for the syntax, first up, we've got the if keyword. And yes, it is case

sensitive, so always lowercase. Well, then after that comes the expression to evaluate, and like we said, this can be anything that evaluates to either true or false. So like we said, again, is a user 18 years or older? If they are, do this; if they're not, do that. Other examples. Oh, is free disk space more than 10 GB? If it is, carry on; if it's not, throw an error. Hopefully you get the picture. Then we need a set of curlies, and like I just said with functions, they've got to go here; do not be going and putting this first one on its own line. If you do, you'll just make the compiler grumpy. And actually, on that note, now, this doesn't really matter, so don't let it mess with your mind or anything, but the reason behind all the strictness with the placement of curlies, well, you know how in some of the languages, they make you terminate lines with semicolons? Well, guess what? Go actually does as well, only Go is really nice and the compiler does the semicolons for us. So, if you throw your opening curly on its own line by accident, the compiler is going to throw a semicolon up here and it's going to break the program. But who cares, right? I don't know why I even said that. Anyway, look, we put the code that we want to execute inside the curlies, and that's a basic if, that block of code will only execute if the expression here evaluates to true. But you can do else if and else branches, and we're about to see it all in action. But that is the basics of if syntax in Go. Whew! Let's go see it in practice.

"If" in Practice

Alright, we've got this framework of a program with a couple of variables declared and initialized. Actually, the code I'm using is in the conditionals folder of the course's GitHub repo, and it's called if.go. I know, really creative. Anyway, this first variable here is showing the length of my Docker Deep Dive course in minutes, and then this one is showing the same for my Containers on AWS Wavelength course. So, Docker Deep Dive's, like, I don't know, what is that, about 4 and a bit hours long, but Containers on AWS Wavelength, only half an hour, so entirely different beasts. Okay, oh actually, remember, Go can infer types, so it knows both of these are integers. Anyway, let's throw in a simple if block. Alright, magic. Like we just saw, we start with the if keyword, and we're evaluating whether Docker Deep Dive length in minutes is greater than Containers on AWS Wavelength length in minutes. Wavelength, length. Anyhow, like I said before, we use Boolean operators like these I'm showing on the screen to do the actual evaluations. Well obviously, this time we're asking is the length of Docker Deep Dive greater than the length of Containers on AWS Wavelength? If it is, execute this code here. Okay, well, magic. But what if it's not, or actually, what if they're the same? Well, this else if here uses the equals operator in case they are the same, and we use double equals here so that Go doesn't think we're assigning a variable a value. Anyway, it executes this line here, and then we've gotten else on its own that basically says if none of the above are true, run this code here. Now, it's a no brainer. I get it. Docker Deep Dive's obviously longer. But now I'm seeing it in action, and remember, save any changes you've made. Okay, mine's called if.go, and there we go. Docker Deep Dive is longer than Containers on AWS Wavelength. Okay, I'll tell you what. Let's jump back here, and we'll have a step through. We've got the usual blah, blah, blah at the top that I'm hoping we know by now. We're declaring a couple of integers as variables here, one's way bigger than the other, and then we're running this line that says, hey, if Docker Deep Dive's longer than Containers on AWS Wavelength, which it is, run this code here. And because that expression was true and its code executed, program flow then skips right to the end here, and it doesn't even bother checking any of the rest. And then because we've got no more code in

the program, the program exits with a 0 return code, indicating a success. Fabuloso! But, what if Docker Deep Dive wasn't longer, like what if the variables were like this? Well, looking at the code, it's pretty simple. We come into the if block, is Docker Deep Dive length in minutes more than Containers on AWS Wavelength? That will be a negative this time, Houston. So, we skip here. Are they the same? No, that's also a negative. So, we skip to this catch-all else statement, and we execute this block of code. Well, I'm saying that's what will happen. Let's give it a save and a try. Containers on AWS Wavelength must be longer than Docker Deep Dive. That's the fundamentals of if in Go, but there's more. So, up next, we're going to dig a tiny bit deeper with simple initialization statements.

Simple Initialization

Okay, sticking with the same code, if blocks also allow simple initialization statements that are executed right before the expression gets evaluated. So as an example, instead of declaring and initializing our variables up here, we can actually do it right here in the if block. And a reason we might do it this way is that any variables we declare here are scoped within the boundaries of the if block. So, they only exist while we're executing this small block of code, then they get garbage collected when program flow moves on. Okay, so let's get rid of these here, and we'll declare them here instead. Anyway, look, when execution reaches here, we read it left to right, top to bottom. So, we're declaring and initializing dddLengtMins as 275, and cawLengthMins is just 30. Then, we're evaluating the expression is Docker Deep Dive longer than Containers on Wavelength? And then, everything's as it was before. So, yes, it is longer. And we execute this code here and jump immediately to the closing curly. And I think, I don't know actually, well, I think I hinted at it earlier, but once any expression evaluates to true, and remember, we go from top to bottom, but any time one of them evaluates to true, we execute its code, but then we jump straight to the end and execute any code right below here. So it's not like we evaluate this one. Oh yeah, it's true, we execute its code, then we evaluate the next one, and the next one. No. As soon as any of them is true, execute its code, and jump to the end. Also though, we can have as many else ifs as we want, but only one final catch-all else. And then, I think last, but not least, we can nest ifs within ifs, within ifs, within ifs. So just as a quick example, and this code's in the ifnest.go file in the conditionals folder of the courses GitHub repo, but we're evaluating this here. It's true, so we execute its code. But part of that code is another if saying is the course over 4 hours long? Again, that's true, so execute this code here. And go on, just so you know I'm not lying, save any changes, and let's give it another run. And there we go. Yes, Docker Deep Dive is longer, but oh my goodness, it is borderline put you to sleep territory. But that's if in Go, dead easy once you have a play around, but super powerful. Well, next up, switch and case.

Switch and Case Syntax

Right then, switch case in Go is a bit like if, so this should be nice and familiar. However, there are some important differences, and we'll see them as we go, but first up, we'll nail the syntax. We start out with the switch keyword, and then just like with if, you can have a simple statement before the expression. And also, like we saw with if, any variables we declare here in the simple statement are scoped only within the switch block. So, they get garbage

collected once execution leaves the block, but as well, and I reckon you're getting used to this by now, but the placement of the curlies is once again and always vital. So the first one abso-freaking-lutely has to go in the same line as the switch keyword. Whoo! Anyway, between the curly braces, we stick one or more case statements like this, and an optional default block, and you probably guessed it, the default block is the catch-all that executes if none of the case expressions are true. Okay, well, to keep it simple for now, we won't bother with a simple statement, and we'll go straight with a simple, simple, simple expression, Kubernetes Deep Dive. Right, well, go put that value in memory and then for each case statement, it looks for a match. Well, in this example, we match on the first case, and we execute this block of code here. And then, importantly, and this differs from some of the languages, but once the first case block executes, flow jumps immediately to the closing curly, and it picks up on the following line. Now, like I say, some other languages have an implicit fall-through behavior, which is where once any case matches, all of the ones below it will execute as well. But that's not the case with Go, but I'm getting ahead of myself. Let's go see it in practice.

Switch in Practice

Here we are then in a new code file called switch.go, and if you're following along, it's in the conditionals folder of the course's repo, and look at it, it's got a simple framework for a switch case. So, in the expression here, we'll start out epically simple with just a string literal, Kubernetes. Now, obviously, this can be a variable or a call to a function, or whatever, but for now, we're keeping it really simple so the focus is on what we're doing with switch and case. Anyway, Go's going to put that in memory. Then, it's going to step through each of the cases looking for a match, so, let's add these. Now, I'm being daft here, I know, but I just want to be clear, when it's looking for matches, it's got to be exact. So, even though some of these might look similar, they're not, only this one here is going to match. But as well, and this should be obvious as well, the types have to match. So we're comparing against a string, that means an integer or a float, or a Boolean, or anything else is never ever going to match. Well, tell you what, let's add some trusty Println s here to tell us which case matched. And by the way, it's totally okay to put the code blocks on new lines, in fact, I recommend it. Nice readable code should be a top, top priority in all of our lives. Well, program execution is going to come in here and it's going to store Kubernetes (with a capital K) in memory, then, it's going to step through each of the case statements looking for a match. For me, this one's not going to match, so, we should not see this text. But this one here, that is a match, so when we run it in a second, we should see this text, Case 2. Kubernetes, with a capital kicking K. But, then Go has no implicit fall through behavior, so, when it's done with that, it's not even going to bother looking at these below it, it'll jump straight down to here and the program will exit. Well, let's take it for a run. Oh, and don't forget to save any changes, and there we go, Case 2. Kubernetes with a capital K, and look, it's just a super simple example. You should definitely have a play around and try stuff out. Once you've done that though and you feel like you know what you're doing, join me in the next lesson, where we'll take a closer look at breaking and fall through.

Breaking and Fallthrough

Okay, so we've said there's no implicit fallthrough in Go's implementation of switch case. And the reason for that is that each case statement comes with an implicit break. So in some languages, if you don't want fallthrough, you have to manually insert an explicit break. Well, with Go, it's the opposite, breaks are in by default. So, if you want fallthrough, you've got to manually insert one as the last line before the next case block. Probably best if we just see it. So we're obviously the same code here, matching on this one here, the print Case 2, Kubernetes with a capital \"K\"." concur. But, if we put a fallthrough here, well actually, what do you think will happen? Do you reckon it'll run just the next one or do you think all of them will run? Oh, and what about the default block? Well, I'll tell you what. Let's take a look. Okay, so obviously, it ran Case 2, but it only ran Case 3, the one immediately following it. And if we look back here, yeah, look, we've actually got four cases plus the default block. So, these didn't run. And you know what? If you think about it, that's right. Remember, we said every case has an implicit break. Well, we overrode the one in this block with the fallthrough, so execution moved down to here. But when it finished, it hit the implicit break for this block, and it jumped straight down here, and the program exited. So, fallthrough only applies to the case statement it's part of, meaning if we want a fallthrough to the next level, and the next, and the next, we need a fallthrough each time. So, right now, this should run cases 2, 3, and 4, but hopefully not the default block. Let's have a check. Okay, yeah. But, will it fall into the default block if we put a fallthrough down here? I guess only one way to find out. Yeah, there we go. So putting a fallthrough in the case above the default block lets execution drop into the default block. Okay, magic. But one last thing. To test the default block on its own, we'll put whatever up here. So we're no longer going to match on any of these cases. Well, save that, and there we go. We only got the default block. Now then, and please don't hate me for this, but it's more idiomatic in Go, and that just means more common, by the way, or more of a good practice, but it is more idiomatic not to use fallthrough and instead just match multiple values in the same case statement. So to demo this actually, I'll dial it back to this. The actual code's in a file called idioswitch for idiomatic switch. And, of course, it's in the conditionals folder of the GitHub repo, if you're following along. Anyway, I'm importing a couple of packages here to help with some random number generation. But also, we're getting a tad more realistic here by switching on a value returned by a function rather than just some daft string literal. Well, we're declaring a new variable called tmpNum, and we're initializing it with the value returned by a function called random. So, down here, let's drop this code here in for the random function. And if you've been following along, you'll know it's taken no argument, but it is returning a single integer. Here, we're setting it based on the current time to help with the number generation, and then based off of that seed, we're returning like a randomish number between 0 and 9. So, back up here, tmpNum gets assigned that random number that's going to be between 0 and 9. So, in our case statements, let's do this. Now, each line is a comma-separated list that importantly matches multiple values per case statement instead of having loads of cases and loads of fallthroughs. Basically, we're branching on whether the number is odd or even. Well, as always, let's see if it works. Okay. Yeah look, sometimes generating an odd number, sometimes an even. Alright, time for one last look back at if, but in the context of error checking.

"If" and Error Handling

Right. Time for a quick look at how the if statement is commonly used for error handling. Now, there are other ways to handle errors, but what we're about to look at has been pretty much the way for the last, I don't know, forever. Basically, you'll see this in just about all code out there. And you know what? Error handling might possibly be the single most common usage of if in Go. Anyway look, and I'm not sure if we mentioned this yet, but it's idiomatic in Go for functions to return an error value as its last return, like this function here that's testing connectivity to a remote host. So we've got the function name and its input parameter and type, but then two return values, and of interest towards us is the last one, and yes, it is of type error. So yes, error is actually a defined type in Go, just like we've seen with strings and integers and Booleans and the likes. As well though, it's normal practice in computer science in general for a nil or 0 error code to indicate success, and then anything other than nil or 0 indicates an error. So if the function executes as expected, like it gets the input values and everything it needs, but more importantly, the function code executes as expected, then it returns nil value and all is good in the world, but if something goes wrong, then it returns a non-0 code, like a 1, or a 2, or a 3, or whatever, right? And of course, it's normal to define different error codes based on the type of the error, and look, we'll see it in a second. Now then, of course, it's all well and good to write good functions that return error codes, but the onus is on us as developers to check for them and do something with them, and we'll see this in a second, right? But honestly, all good Go code consistently, like all the all the time, is checking for errors and acting accordingly. Anyway, what we've got here is a call to the open function in the OS package. Oh, and we're in the iferror.go program in the Conditionals folder of the Github repo, yes? Anyway, we're trying to open a file called test1.txt in my current directory. Now really quick, there are more idiomatic ways to open a file in the current working directory, such as the getwd or the executable functions, both from the OS package. But you know what? That'll just clutter this code and distract from what I'm showing. So I'm doing it like this. Anyway, the os.Open function returns two values. We're not bothered about the first one, so we'll just ignore it with the underscore, but the last value it returns is an error code, so we'll capture that there with the err variable. And honestly, oh, you are going to see this here everywhere, but, and this is the important bit, we check that error code stored in the err variable, and if it's not nil, so if there is an error, we're going to print it to the screen. Obviously, in the real world, we'd have some code hid to catch it and handle it, but because I want you to see what's actually in the error code, we'll print it to screen. Well, we'll save that and give it a whirl. I said this one was called iferror.go, and remember, you have to have saved any changes, but also, if you cloned the repo locally, you will need to run it from within the Conditionals folder. But there we go, and funnily enough, it threw an error. Who would have thought? Well, this is the actual text returned by the Open function in the OS package, and it looks like the file doesn't actually exist. Let's have a look. Ah! Right! We're looking for test1, but it's actually just test. So if we jump back here to our code and make this just test, and as it's obviously going to work this time, let's put a copy of this println down here, so we see what the return code looks like when it actually does work. Give that a Save and Run. Okay, magic! An actual nil value, just we said. So we've got this here when it errored out, and then a nil value when it worked. Well, I have been saying you'll see this kind of error checking all over the place, so, oops, give me a second, actually. Right. This here is the deployment_controller package from the Kubernetes project, meaning Kubernetes is obviously written in Go, and actually, you know what? Most of the major cloud infrastructure projects are. So Docker, Istio, Linkerd, obviously Kubernetes, and way more, all of them, written in

Go. Anyway, the reason we're looking at the Kubernetes code is that it is proper weapons-grade production code. And if we search up here for if error, well, look at that; 20-odd references just in this package. And look, the second one down is pretty much exactly what we've done. If err is not nil, okay, and this is saying or, if the length of the value in the deployments variable is 0, do this here, basically exit the function and return a nil value. So I guess yes, a tiny bit more complex than what we've just done, but the point is, the code is literally littered with this type of error checking, so like we said, very idiomatic. And you know what? That's been a lot. So time for a quick session recap.

Recap

Okay, I'm going to make this short and sweet. We started out looking at if statements and how they use Boolean logic to evaluate if things are true or false. The structure is basically if something is true, execute the block below it, and, of course, we can have as many else ifs as we want, and then, obviously, if any of those are true, execute their code, and we round things out with a catchall else that'll run if none of the stuff above it is true. Well we also said that as soon as any of the statements evaluates to true, and we evaluate them from top to bottom, but, as soon as one fires as true, we run its code, but then program execution jumps straight to the end, so we don't bother checking the rest. Well, then we looked at how if is commonly used to check for errors. So, it is good practice for Go functions to return error codes as their last return value, and then, we normally capture that in a variable called err, and then use it to say if err isn't nil, so it's not nil, indicating an error, run this code and handle it. Well, we also looked at switch case in Go, and the logic is similar to if, but, it tends to be a shorter way of writing an if block without having to have tons of else ifs. Anyway, we evaluate an expression. The first case to match runs its code, and then it breaks to the end here. But if we want to run more of the case statements, then we can drop in a fallthrough, and the next one will run, but only the next one. So, if we want them all to run, they all need a fallthrough. And I reckon that about wraps it. Again, practicing is vital, so, fire up your editor and play around, and see if you can break it and figure out what works and what doesn't, and when you think you're good, join me in the next module, because we're looking at loops.

Working with Loops

Module Intro

Right, if you're following along, we are properly building some Go skills. Well, in this module, we're looking at loops. And in Go, there's only one loop, the for loop. But to be honest, that's a bit misleading because it is a really flexible loop, in fact, I should probably have said there's only one loop keyword in Go, the for keyword. But it can do infinite loops, while loops, and it can even range over lists. Anyway, we'll hack it like this, we'll look at the basic syntax first, then we'll look at infinite and while loops, then ranging, and a bit about breaking and continuing, and we'll finish with a recap. Well, let's do this.

Syntax

So the basic syntax is dead simple, it's the for keyword and an expression. Though actually, the way to create an infinite loop is to just leave out the expression. That way, Go assumes it's Boolean true and it will loop while true. If it's not blank though, then it can be either a Boolean expression or a range expression. And we're about to see them both, so don't worry if that's a bit confusing right now. Anyway, as always, placement of the curly braces is vital. Then between them, we put the code to execute. So, like we said, an infinite loop looks like this, just the for keyword followed by the opening curly. And, also like we just said, it's basically for true, or while true as you get in some other languages. The thing is, it'll loop forever unless we throw a break in the code somewhere here. Well, to loop on a Boolean expression looks something like this, this one's saying for i is less than 10, but you can use any valid Boolean expression, and it'll basically loop while i is less than 10. Oh, and it's idiomatic across just about all programming languages to use a lowercase i as your incremental or your index value. We'll see it soon. Anyway, finally, there's for range, and this is the bizzo, right? It takes a list of some sort, maybe a slice or a map, and we'll cover these next I think, but it takes the list and it illiterates or it ranges over it from the top to the bottom looping through the whole list. And then, when it reaches the end, the loop exits and code picks up after the closing curly. So, in the example we're looking at, the list is called courseList and for every iteration of the loop, it works its way down the list and assigns the current value to the i variable. Okay, one last thing about syntax before we see it in action. Just like with if and switch in the previous module, we can give simple pre and post statements on the opening line. So, if we want the example back to the Boolean expression, basically a while loop in a lot of languages, well, we can do this. So, it's three statements divided by semicolons. This is a simple pre statement, initializing i as zero. Next, is the Boolean expression saying loop while i is less than 10. And then, this final bit is the post statement, and it's applying the increment operator, basically, incrementing the value of i by 1 each time the loop completes. Now, it's important that this is a post statement, yeah. So it runs after each iteration of the loop. So we initialize i with 0 in the pre statement, the expression evaluates to true because, well, we've just set it to 0. The loop then runs and then the post statement increments it from 0 to 1, and we go again, rinse and repeat 10 times until i actually is 10. Then the loop exits, i gets garbage collected, and execution picks up after the final curly. Well, we know that talk is extremely cheap, so let's see this stuff in action.

Infinite and While Loops

Right then, we're in a new code file here called self-destruct.go, and you'll see why in a second. But, it's in the loops folder of the GitHub repo, if you're following along. Now then, we're going to write a super quick countdown timer, counting from 10 down to 0. And, because I'm, I don't know, a little bit of a sci-fi geek, I'm imagining we're on a spacetime ship that is under attack from a hostile alien force intent on stealing some important technology, of course. Well, we're also outgunned and we're about to lose the tech. So, as the captain of the ship, I've launched the rest of the crew in escape pods and heroically stayed behind to initiate the ship's self destruct on a 10 second timer. So, we're about to write that self-destruct timer code. Anybody still here, or have you all left? Okay, look, we're inside the main function and I guess we could initialize the time of variable right here, but we're not going to, it's more idiomatic to declare it as part of the loop. And like I said, we'll start at 10,

but, we want it to count down to 0 and then stop, so, each time the loop runs, we want to decrement the value by 1. Okay, well, like all good self-destruct timers, we need something telling us how long is left, and we're going to need something to insert approximately a 1-second delay each time we iterate. Okay, I reckon that might do us. Oh, I forgot to mention, obviously we're leveraging a couple of functions from the time package we're importing up here. Anyway, when the timer reaches 0, we want something a little bit spectacular, so let's whack in a quick if statement, so, if timer = 0, then run this, boom. Yeah, not quite so spectacular. Whatever though, let's give it a save and a run, literally, run, before the ship blows up. Oh dear. Okay, so, what shall I do in the last few seconds of my life? Oh, yeah, obviously, Sunderland are the greatest football team in the world, ciao, ciao. Okay, it worked, well, kind of, I've got this sloppy 0 hanging around. Now, to fix that, I guess we could loop while greater than or equal to one, but actually no, it would never trigger the if statement, would it? So, tell you what, we'll slap in a break. Now, we're going to look at break properly later. Anyhow, after we execute the boom, and I suppose it wouldn't make much of a sound in the near vacuum of space, but what the heck, then as soon as we hit the break, we exit the loop and we run any potential code immediately following the closing curly. Meaning, we shouldn't see the sloppy 0 anymore. Well, we'll check if that works. Let me speed it up this time, it's a bit like Groundhog Day or Boss Level, if you've seen that. All right, way better this time. But, really quick before we move on, just to be clear, this pre-statement here gets executed before the first evaluation of the expression. Then, the post statement, that gets executed post, or after each iteration of the loop. But importantly, that is before we reevaluate the loop condition. Okay, well, next up, we'll look at for range loops.

Range Loops

Okay, for-range loops. And to demo this, we're back from outer space. I guess I went back in time or something so that we're before the self-destruct. Either way, we're back to our good old Pluralsight courses. Now then, this here is a list of courses in progress. And to make it easier to read, I've split it over multiple lines. Now technically, it's a slice, basically an unordered list of numbered items. Well actually, under the hood, it's a reference to an array and the likes, but we'll cover all of that in the next module. For now, all we need to know is it's a list. So, we've got a slice of four strings, four courses, here. And to ease you in gently, let's just jump straight and see for-range in action. So, the range command is going to take the list, range over it, so step through it, one value at a time until it reaches the end, but it steps through one element of the list per iteration of the loop. So, the first time the loop runs, it'll get one course. The next time, it'll get another, and so on. Now then, with slices, for-range returns two values each time it loops, the index value and the data value. We've got four items in our list, so that will be indexes 0 through 3. The first time through the loop, it's probably going to get 0 and Docker & Kubernetes: The Big Picture. The second time, 1 and Docker Networking, Then 2 and Getting Started with Kubernetes, and finally, 3 and Kubernetes Deep Dive. Now, for our for loop, we're not bothered about the index value, so we'll ignore that by passing it to the underscore, the blank identifier. But each data value, yeah, we'll hang on to that with the i variable. Then obviously, we're printing i to the terminal each time we loop. So, we should end up printing the names of all four courses. Okay, let's save that and give it a go. Alright, just as expected. Now then, we can also nest loops, so loops within loops. So to do that, we'll have another slice of strings. This time, courses marked as completed. So, just to be clear, we've got two lists now, one listing courses in progress, the

other listing courses completed. Well, let's add another for-range here, importantly inside the existing loop. Well, this one's going to loop through the completed courses list, and see how it's using j to hold the values from it. So, the outer loop is holding values in i, the second inner one in j. Well, let's add an if here, comparing i with j and then printing this if they match. So, looping over courses in progress and also courses completed with an if block to spot if a course appears in both the in progress list and the completed list. And if you're new to this and it's feeling a bit much, let's just step through the flow nice and slowly. We have got the outer loop stepping through the list of courses in progress, but inside of that, we've got a second loop stepping through the list of courses completed. Now then, on the first iteration of the outer loop, it's going to stick Docker & Kubernetes: The Big Picture into the i variable. In fact, let's put that up here. Okay, anyway. Next up, we hit this line here, so we start iterating through the inner loop. And we stick the first value from the completed list into the j variable. And look, it's going to be Docker & Kubernetes: The Big Picture as well. That means when we do this if comparison here, we'll get a match, and we'll print this line. Okay, well, we'll hit the closing curly of the if statement here, and we'll go back to the top of this loop again, the inner one, and we'll run through that again. This time, we'll put this in j. Does i match j this time? Negativo. And we hit the end of the completed list. So, we leave the inner loop. We hit the closing curly of the outer loop, but we've not reached the end of the list for the outer loop, so we go again. Well, this time we'll put the Docker Networking course into i and go through it all again. So back to our inner loop, a couple of iterations, one for each of the two values here. Each time again, checking for a match. This time, we won't get any matches, so we'll come back up here, rinse and repeat. Shall we see if it works? Okay, well, remember to save your changes. Oh, and we don't need this Println here anymore. Let's give it a go, though. Oh, and as if by magic, it works. Now, I know it was a bit of a convoluted example, but we've seen how to nest loops within loops, as well as throwing in some if blocks. Well, time for a quick look at break and continue before we recap everything we've learned.

Break and Continue

Time for a quick look into how break and continue work. In fact, we saw break earlier when the self-destruct timer was flashing a 0 that we didn't want. Remember, we put a break in here inside the if block, which in turn is inside the loop. But, as soon as program flow hits the break, it literally breaks out of the loop and execution picks up after its closing curly. And you know what, yeah, that is break in its most basic form. It breaks out of the current loop. But what about nested loops like we've got here? And apologies for the colors. They're just there trying to help highlight the different loops. But we've got three nested loops, and apologies if you're color blind. I hope they are visible and make sense. Well, if we break here in the innermost loop, we'll drop out of this loop here, the blue one. Then we'll hit its terminating curly, go back to the top for further evaluations, and potentially more iterations. Magic. But what if we want to break out, I don't know, two levels, maybe to the outermost orange loop? No sweat. That's where labels come into play. So, we can make a label up here. We'll call this one breakPoint. But, honestly, you can call it pretty much whatever you want. Just, obviously, don't call it one of Go's whatever it is, 25 keywords. Anyway, when we make the call to break here, if we give it the label, instead of just breaking out of the current loop, we'll break to here, thanks. And that really is the crux of break. In its most default form, it breaks us out of the current loop, but if we use labels we can break to pretty much anywhere we want. Though, a bit of a gotcha, Go being Go, yeah, it is not a fan of you defining labels and

they're not using them. So, if you define one, you've got to use it. Well, switching gears slightly to continue. The idea with continue is that whenever Go encounters one in a loop, it drops whatever it was doing and it jumps straight back to the top for a new evaluation and potentially more iterations through the loop. Oh, and actually, post statements, they are actually executed as part of the continue. So basically, stop what you're doing, run any post statements, and then reevaluate the expression, and maybe go again. Well, we've got a bit of our old self-destruct timer code here, so defining a timer as 10 and then looping while it is >= 0 and decrementing by 1 for each iteration of the loop. Well, if we change this if statement to this, okay, we're using some simple math to determine whether or not the value in timer is an even number. If it is, we'll run the continue here, which, we just said, interrupts normal flow, so it skips the Println here and obviously the injection of time as well, but it jumps us straight back to the top where we run the post statement here to decrement the timer by 1, and then we reevaluate the expression. Look, basically, we are skipping the Println every time it's an even number, meaning when we run it, we should only see odd values. Well, let's go and see if that actually happens. Okay, magic, only odd numbers. And that, folks, is break and continue. Whoo, let's wrap the module with a quick recap.

Recap

Okay, let's make this quick. Go only has one key word for loops, for, but it is pretty flexible. It can do infinite loops, traditional while loops, where it loops while a Boolean expression is true, and it also does range loops, where it steps through a list. So, the for key word on its own is the equivalent of while true, so an infinite loop. If we use it with an expression like this, it loops while the expression evaluates to true. Like I said, a lot like a while loop in some other languages. But, then range loops, which look like this, range or iterate over a list, and the loop exits when the end of the list is reached. Well, as well as that, we saw they can have simple initialization statements that run before the loop starts, as well as post statements that run after each loop. Okay, we also saw we can nest loops within loops within loops within loops. And, of course, other code, like if statements and the likes, can all go inside of loops. But, then last but not least, we saw how we can use break on its own to break out of the current loop, but if we couple it with a label, we can target pretty much anywhere for break to land. And then, the continue key word lets us drop the current flow of a loop, return to the top where we run any post statements, and maybe run through the loop again. Okay, well, look, a really quick peek into the Kubernetes code base again, and we'll just go for the same deployment controller code that we went with last time. Only this time we'll look for for i. Yeah, anyway, look, it's just easier than looking for for on its own. Anyway, this one's a classic example. So we can see it's got a pre-statement initializing i, told you it was idiomatic, then it's evaluating if i is less than the value stored in workers, and then, the post statement here increments it for each run of the loop. Then, of course, there's code here inside the loop to run, which this time is a call to a function, but it's using the Go key word to start it as a Go routine, and I know that might be a bit complicated, but if you've been following along, you'll know that the wait package here must be being imported somewhere at the top, and that this function here is part of that weight package and it's exposed because it starts with a capital letter. Actually, in fact, if we click this, yeah, look, this is the package it's declared in and we can even click this, and there it is. But you know what? That's not the point? The point is, you will see for loops everywhere in Go, massively useful. Okay, well we're rattling through stuff. Next on the card, arrays and slices.

Arrays and Slices

Module Intro

Okay, arrays and slices. And as usual, the idea is that when we're done, you'll have a decent grasp of what they both are, as well as obviously, how to work with them in Go. Well, we'll set about things like this. First up, we'll nail any theory we're going to need. You know, basic stuff like what are arrays and slices, and how are they different? But also, pretty much as we're about to see, how slices are better and more powerful than arrays. Then, we'll look at the syntax of how to declare them, and we'll see how to work with them. Along the way, we'll look at some examples, so see how to actually implement them in code. And I reckon also, we'll look at how we can manipulate them, like grow and shrink them, and change their values and stuff. And yeah, that's the plan. So first up then, let's go and see if we can explain the theory and the fundamentals.

Theory

Now then, right at the top here, I want to throw out this warning. If you've worked with other languages and you've got experience with arrays and slices, well, the way Go handles them might be different. So, don't just assume, oh, yeah, I've done this kind of stuff before. Let's just crack on. Don't. Just give me a few minutes of your time to make sure we're all on the same page. Anyway, jargon aside, we're basically talking about numbered lists,
where everything in the list has to be of the same type, so I guess all strings or all integers or floats or whatever. Well, let's break that down. Like we said, we're talking about lists. So a list, of course, is here. And then each item in the list is numbered, or, as we tend to say, indexed. As well, though, every item's of the same type. Here, we're obviously strings, but it could be floats or ints or something else. The point is, within the same list, they've all got to be the same type. And I guess backing that, and I know I've said it a ton of times, Go really is picky about types. Anyway, same again for a list of ints, all the same type, oh, and always zero indexed. Oh, and then this one here, that's illegal. It's trying to mix types, and the compiler won't have it. And you know what, I reckon that'll do for the uber-basics. Let's switch tack a bit, and we'll see how arrays and slices, while being similar, are actually different, but they work together. Talk about confusing. It's not actually, though. Join me in the next clip, and we'll see how easy it actually is.

Arrays vs. Slices

Right, then. Two quick things before we dive in. Slices are absolutely where the action is, like you will rarely see arrays used directly in Go. As well though, we're going to go a little heavier than normal on the theory here because I think it's really important to understand how slices are put together behind the scenes, certainly if you plan on being serious with Go. Well, arrays are like the lists we just saw. But importantly, arrays have a fixed length, meaning if we create one with six elements, then we're stuck with six, like there's no adding a seven or an eight. I mean, technically you can do it, but it is a proper bunch of work. Like, you'll have to create a new array, copy the contents of the old one over, do a bunch of renaming,
and probably a bunch of other stuff I can't think of right now. So, if only there were a better

way. Captain obvious, say hello to slices. So at the very highest level, slices look and feel like arrays, but they're resizable, meaning need to add some elements? No sweat. Append a value to the end. Need to make the slice shorter? Ha, walk in the park. Just create a slice of it, so a slice of a slice. Now then, the way it all happens is basically abstraction. You see, slices are actually built on top of arrays. And I'm going to back right up and repeat that again because it is dead important to know, so here it is, slices are built on top of arrays. In fact, it's probably where the name comes from because slices, well yeah, slices of an array. And maybe actually a picture helps. So this is an array with 10 elements, then on top of it we create a slice of just 3 of them. And it doesn't have to be the first three. It could actually be any, as long as they are contiguous. In fact, drawing it like this might be a bit better, a bit more obvious to see why we call it a slice, yeah. Back here though, no actual data gets stored in the slice construct. It's basically pointers pointing to the actual data in the array below, meaning the slice itself is basically a name, I guess a type, an offset in the underlying array, and a length, no actual data. And just to blow your mind, we can have multiple slices of underlying arrays. And if we do that, as they're basically pointers, they are really cheap. However, in that model, obviously changing the value of any element in one of the slices changes it in the array below, meaning every other slice that references it sees the change. And look, I know it's not rocket science, but it is super important to know, especially when passing slices as arguments to functions. So, because slices are references by nature referencing the array, well, when we pass them to functions, they get passed by reference, not value. Well, actually, the slice header, which is basically jargon for the location of where the data actually is in the array, that does get passed as a value, so copied to the function. But as it is actually a reference to the real data in the array, the overall effect is that of passing by reference. Oh, now, I know that might sound confusing, especially if you're new to all of this, but it should be pretty simple if you watch the functions module where we covered passing by reference and passing by value. Basically look, the slice itself gets passed to the function by value, so a copy of the slice is passed to the function. But seeing as how the slice is actually just pointing us to the array, the actual array values are not passed to the function, just the slice, and the slice is basically a reference. I'm starting to confuse myself. No, just kidding. But actually, I reckon that is enough on the theory. Let's go and get our hands on.

Working with Slices

All alright, to create a slice, and we'll be working exclusively with slices by the way. Remember, slices are where the action is. But, to create one, we can use the built-in make function. Now, this accepts three values, type, length, and capacity. The type, well I guess that's pretty obvious. Are we making a slice of strings or integers or whatever? I reckon the length should be pretty obvious as well. How many elements do we want in the slice? Like how many entries. But capacity. What's that about? And how is it different to length? Well, capacity specifies the maximum size of the slice, or maybe actually, the expected size. Basically, under the hood, this is going to be how big to size the array that's backing it. Which is why we did what we did, explaining how slices are backed by arrays. It's kind of fundamental to know. Well, anytime we create a slice, Go creates an array to hold the actual data, remember? And I know, I am beating this point to death, but a lot of us do learn by repetition, so slices don't actually store data, it's always stored in the array. Anyway, look, if we wanted to define a slice of strings with an initial length of 5 and a capacity of 10, we'd do it like this. Okay, let's add this here. It's basically a Printf from the fmt package that

displays the length and the capacity of the slice. This here just says, insert the output of len(courses) here, and the output of cap(courses) here, but a standard base 10 integers. And then, okay, len and cap are built-in functions that return the length and capacity of slices. Anyway, look, we'll get length of slices 5, and capacity is 10. Well, let's give it a save and a try. Marvelous. Now then, all we've actually done so far is declare the slice, we've not initialized it with any values, so it's basically an empty slice. Well, to actually put some data values into it, we can do this. And look, I could put this anywhere, but I'm putting it up here below the definition of the slice, basically, because I think it looks tidier. But, what we're saying is stick with this in here as the first entry in the slice, remember? Slice is a 0 index, so the first element is always item 0. But then, stick this in at element 1, and then this at element 2. Well, let's add this to print the contents of the slice to the screen, and it's going to be ugly, but it'll do the job, for now, we'll throw a loop in later to tidy it up. I really want to focus at the moment to be on slices and not anything else. Well, let's give it a whirl. All right, so the same length and capacity, but now it's showing the entries. But the thing is, we can make it look a lot tidier. So instead of declaring the slice here and then initializing it later, seeing as how we know the values now, let's just do it all in one go like this. Now interesting. What do you suppose the length and the capacity functions here are going to return now? I mean, we've just declared and initialized it with 3 values, but we've not specified a length and a capacity. Well, my guess is we'll get 3s. But before we check, this was a bit ugly, wasn't it? I'd rather see each element on its own line, so we'll do this. Recognize that? You should, it's our old friend for range. And we're basically saying, range over our courses slice, and for each iteration of the loop, assign the data value to i, and print i. And then because Println automatically adds a new line each time it runs, we'll get three entries printed over three lines, it'll look a lot tidier, trust me. Well, let's see. Epic. Oh and look, length is 3 and so is capacity. Brilliant. Well, let's go and dig a little bit deeper.

Getting Under the Hood

We've already seen a bit of this in the last clip, but we can manipulate slice elements by whacking the element's index value inside of square brackets. In fact, remember this. So, these are the element's index values, and we populated them with these data values. Okay, well, I'm switching to the elements.go file in the slices folder of the GitHub repo. Alright, we've got a slice of 10 integers and a Println to put them on the screen. I reckon this will look tidy enough without putting a loop in. However, if we put an element number in here like this, it will only print the value at index position 4, so let's test it. Okay, 5, but we wanted element 4. Now, I'm only messing with you really, and I'm sure some of you know what's going on. But just for clarity, like all good computer systems, Go starts counting at 0, not 1, meaning our index positions are like this. So index 4 is data value 5. And again, I know it's not rocket science, but it is easy to forget. And when you do, and most of us do from time to time, but when we do, it can be properly frustrating. But, just to keep you on your toes, and this makes perfect sense actually, but functions like len and cap, they obviously start counting at 1. I mean, we can't have a slice with one value, but then have len return its length as 0. I mean, that would be proper madness. Anyway, as well as referencing elements for printing, we can obviously manipulate them as well. So, this here is going to change the value of element 1 to be a 0. Now remember, element 1 is actually the second element, so it's currently holding a 2. Well, we'll stick a Println here to verify the change, and let's take that for a spin. Okay, element 1, now holding a 0. Well, we can also slice a slice. In fact, that's how we make a slice

smaller. So to do that, we create a new slice, and we tell it to point to a subsection of the existing slice. Here, we've got a new slice called sliceOfSlice, and it's basing itself off of values 2 through 5 from mySlice. Though, and I'm starting to think slices have quite a few quirks, but the way Go works is it will use values 2 through 4, so not 2 through 5. I know. Basically, the first number that we specify is inclusive, so we'll be including element 2, but the last number isn't included, so it'll stop at element 4. And by my reckoning, we'll get 3, 4, and 5, which will be elements 2, 3, and 4. Well, let's whack in this Println, and we'll see if we're right. Bingo! Anyway, a couple of points on the syntax of slicing. When creating slices or even actually referencing elements, if we don't put anything before the colon operator here, index 0 gets implied at the start of the slice. So this here will be 0 to 5, and it's the same for emitting the value after the colon. So in this instance, the end of the slice is implied. Also, just one last quick reminder that passing slices to functions, although it actually passes by value, because the slice itself is a reference to an array, it's effectively passing by reference. And I reckon that'll do. Next up, we'll see how to expand a slice.

Appending to Slices

I think I mentioned it, probably at the top of the module, that a major benefit that slices have over arrays is they can grow and shrink. Well, to grow an array, we can use the built-in append() function, and it works like this. Go take the current capacity of the underlying array, And for each append(), it adds data to the next slot in the array. So if we've got a slice here with the length of 1, but a capacity of 4 and then we add a value, it'll go to the second element of the array below, if we add a third, it'll go here, and a fourth here. But as soon as we add a fifth, Go doubles the size of the array below, so we'll go from an array with four slots to one with eight. Only, I don't know, Nigel, I thought you said arrays can't be expanded? True. So Go is actually doing a bit of magic in the background, creating a new backing array of double the size, copying over any values, and doing anything else that needs to make the whole process seamless, and we don't even notice. And that's the crack, fill up the current backing array until it's full. Once we outgrow it, Go performs a little bit of background magic and doubles its size. Well, I think it's time to put that to the test with a bit of code. All right, let's step through that. Obviously, we're making a slice here with the length of 1 and a capacity of 4, and then this line here is printing the info to the terminal, basically just proving our starting point. In fact, we'll put up here what we're expecting to see. Anyway, we create a loop here that'll iterate 17 times. And as we can see, for each iteration, we're appending a new element to the end of the array, but we're also printing this line here saying the current length is X and the capacity is Y. So, for the first run through, we'll see array length is 2, but capacity is 4. Next time, it'll be 3 and 4 and then 4 and 4. But as soon as we append the fifth value goes, Go is going to work its magic to double the underlying array size, which is basically the capacity of the slice. So, on the screen, we'll get length is 5, but capacity is 8. Well, we'll rinse and repeat until we get to 9, at which point, Go should up the array to 16, more rinsing and repeating until we reached 17, and it should be doubled to 32. Well, everything looks good on the slides. Let's see if it actually works like that. (Laughing) As if there was ever any doubt. Well, you know what, take a second to look at that. But it's essentially what we sketched out on the slides, appending to the existing array until we reach its capacity, then doubling its size, and rinsing and repeating. Magic! Well, there's a couple of other things I want to point out before we do a recap.

Miscellaneous

All right, let's tidy up a few loose ends. Simply referencing an array or slice like we do with a variable, so here the slice is called mySlice, well, that'll reference the entire thing, like every entry, which isn't necessarily the same as other languages. I know, like, for example, doing this in C only returns the first element. But here in Go, it returns the entire thing. Also, and I know I showed this earlier, but for-range loops are pretty much perfect for working with slices, and you'll see them together, like, all the time. But recapping, for-range returns two values for every iteration of the loop, so it returns the index value and then the data itself. Now, actually, this might not be the best example here, because it is a slice of integers and just might make it look confusing. But the first run through is going to return index position 0 and data value 1, second run, index 1 and data value 2, eventually finishing with index position 4 and data value 5. However, if you're only interested in the data value, which is pretty common, then you can disregard the index value by passing it to the underscore like we're doing here. Well, next up, seeing as we just looked at append, it is totally possible to append slices to slices. Well, it kind of is. You see, the process doesn't actually append one slice to the other; it appends the values from one slice to the end of the existing one. I mean, look, the result's the same, but how it happens under the hood maybe isn't what the operation looks like. But look, honestly, the result is the same. Now, small print does apply, but it's all pretty obvious, like you can't append slices of different types. So, if you've got a slice of strings, you can only append other strings to it. Anyway, we've got a slice called mySlice with a bunch of ints, so let's create a new one, and I know I've gone a bit OTT with the crazy names here. But the point is it's all integers, and then we append to it like this. And I get it; i totally looks like---well, actually, first, the formula for the syntax is the slice to append to, followed by the slice to append from, with ellipses bolted to the end. So, this one will append newSlice to the end of mySlice. Only, like I said, what it's really doing is it is appending the data values from newSlice to the end of mySlice. But again, like I said, the mechanics don't massively matter. We'll end up with a slice of, what is that, eight elements. Well, let's throw another Println here to make sure. But actually, before trying it out, let's quickly step through. We are declaring and initializing mySlice here as a slice of 5 integers with a length and a capacity of 5. And then we're proving that with this line here. Then we're showcasing a for-range loop that'll put the data value from the slice in the i variable and print it to screen. So, we should get lines showing 1 through 5, and we'll get a new line each time because Println prints a new line each time it runs. Anyway, then we're creating a new slice called newSlice with these three values and appending it to the end of mySlice and printing this. So, we should get mySlice NOW contains [1 2 3 4 5 10 20 30] and has a length of whatever and a capacity of whatever. And I reckon the length will be 8 because of the 8 values, but the capacity will be 10, because when we went over the initial capacity of 5, Go will double it to 10. Well, shall we see? Epic. I reckon, yeah, I reckon that's exactly what we expected. Now, maybe you might want to pause the video here and just double-check that, especially if you're following along in your own code. But yeah, that looks right to me. Anyway, look, next up, quick recap.

Recap

All right. We started out by saying that when it comes to arrays and slices all the action, and I do mean, all the action is with slices. They are both lists of a single data type, but arrays have

got a fixed capacity, whereas slices don't. The long and short, we can append two slices to make them bigger, and we can slice slices to make them smaller. The moral of the story, 99% of the time, use slices, not arrays. That being said, we did dig into the theory a bit and slices are just a bunch of pointers that point to the real data stored in arrays. But of course, it's all done cleverly, so we basically never need to know. We just act on a slice like it's a flexible array. Though, I do think that knowing the inner workings helps you understand things like why if multiple slices point to the same array elements, changing the value in one slice impacts all the others pointing to the same elements. We also said that the for range loop is the perfect companion for slices. Like it iterates over a slice with the loop exiting once the end of the slice is reached. And honestly, you'll see the two of them used together all the time. But I reckon that's it, and I'm keen to get on to the next topic, maps.

Working with Maps

Module Intro

All right, maps in Go. And if you've been following along, they're going to look a bit familiar because they're quite a bit like arrays and slices, but there are a couple of differences. I think first and foremost, maps are, well, maps are key sorted lists, or at least, sometimes they are. Uh, what's that, Nigel? Sometimes? Oh dear, go on, do tell. And you're going to see this as we go, but, retrieving a map without a loop, Go's going to return it in key-sorted order. But when iterating it with a range loop, ha, you get them back in random order, or at least, there's no guarantee the order you'll get them in. And I get it right, if that sounds a bit confusing, I don't blame you, but it will be crystal clear by the end of the module. Anyway, another difference with slices is that maps are key-value pairs. And if you've programmed in other languages, you might think hello, you look like, I don't know, a dictionary or maybe an associative array or a hash table. In fact, yeah. Maps are basically Go's implementation of a hash table. So, as a quick example, and I know, I'm getting ahead of myself a bit, but I do like seeding ideas before diving in, but this is a list of some of the bigger football teams in England and how many times each one has won the top league. Now, it's not a full list and it's not in order or anything, it's really just a sampling. Well, the column on the left is the key, and the one on the right's the values. And then, we can dynamically update stuff like adding, and removing, and even updating existing entries. So, yeah go on, let's get rid of Newcastle here and we'll throw in Man City. And then, what about predicting the future? Okay, that's unlikely to happen if you know anything about English football, but as far as maps go, heck, yeah, that's all doable. We'll look back on track, here's the plan. Well look at the basic syntax and getting our hands on with some basics. We'll look at iterating over them with range loops, we'll see how to insert and delete elements, we'll fill in a few of the gaps, and finish on a recap. Well, let's crack on.

Getting Started with Maps

First up, let's see how to define maps, and like I've said already, they're a bit similar to slices. Well, then so are the ways that we declare and initialize them. Anyway, the basic syntax is map, and then because Go's strict about types, we'll declare the key type in square

brackets like this and then the value type right after it. Now, a couple of things. The key type has to be a comparable type. So, a type that can be compared with either the equals or not equals operators, basically, that's bools, strings, numeric data types, and pretty much any of the other types like arrays and structs, so long as they are composed of bools, strings, and ints; however, it does not include non-comparable types, so, not slices and not functions. As well though, and this one's pretty obvious, but key entries have to be unique. You cannot have two entries in the same map, with the same key. All right look, enough with the theory. I'm in the titles.go file in the maps folder of the courses GitHub repo, and we can use the make function like this. So, this is creating a map called leagueTitles with the key type as a string and the data type int. Now, it's only declaring the map, so it's not initializing any values. To do that, we can go with something like this. So we reference the map, populate the key, a string here, then the value is an int. So, Sunderland have won 6 English league titles. That's actually 6 in the all-time list by the way. Anyway, look, same again for Newcastle only, they've only got 4 titles, which again, if you know much about English football, that is to be expected. They're not as old as Sunderland, and they don't have as rich a history. Anyway, as with slices, we can declare and initialize them in a single go with the composite literal form. So, this time we're calling the map recentHead2HeadWins, any Newcastle fans are not going to like me. The map keyword again, we'll have strings and ints again, and then a couple of entries. Now this time, we're listing the number of wins that each team has from the last seven clashes. Obviously, there's only one winner there, Sunderland with 6, Newcastle, oh dear, none. The other one was a draw. The thing is though, look, we'll put a Printf here to display then, and let's give that a save and a try. All right, the formatting is a bit nasty, well, so are the stats, if you're a Newcastle fan. For now though, we only really care that we have defined and populated two maps and we can see them. Oh, but look, see how it's returned Newcastle first both times, even though I think, yeah, we've actually added Newcastle second both times. So that is Go returning them in key order and for Newcastle coming before S for Sunderland in the alphabet. But that's the basics. Up next, we'll see how to use a range loop to iterate over them.

Iterating Maps

So here we go. We just saw that Go returns maps in a key-ordered fashion when retrieving them outside of a loop. Fair play. But instead, it gets all random if we use a range loop. Well, to show that, I am in the maploop.go file from the maps folder in the GitHub repo, and we're looking at a simple strings to int, so mapping capital letters to their position in the alphabet. Well, let's throw in a quick range loop, and like with slices, the loop executes for each key in the map. And then for each iteration, it returns two values, the Key and the Value. So here, we're assigning the key to a variable called mapKey and the value to one called mapVal, and then foreach loop will print the values to the screen. Now, this %v format verb here is saying, print these two variables, but instead of printing them as base-10 integers, like we've seen with %d already, well, this time, just go with whatever the default type is. Anyway, we should get nine lines saying Key is whatever and then Value is whatever, and as well, if we run it a few times, we should see different ordering. Remember, Go doesn't guarantee iteration order with range loops. Fair enough. Let's go and see if it's actually true. Yeah, starting at a different offset pretty much every time. Anyway, up next, we'll have a crack at inserting, updating, and deleting entries from the map.

Updating Maps

Okay. And again, this is the same as with slices, but we reference individual map entries with square brackets. So sticking with the same code file as before, if we lose the loop here, and then put this Println in. Well, like this, it will print the entire map, but if we do this, it'll just print C. Well, actually, it'll print the value associated with C, so 3. We're basically saying, print the value associated with the C key. Let's give it a try. Okay, magic. Well, let's say we want to update a value, maybe change A to 100. Well, here we can just say testMap "A" equals 100. And don't forget the quotes, yeah? The compiler needs to be sure of the type, but that will change A to 100. Well, I'll tell you what, let's change this back here to return the entire map, and let's go again. Magic. A: 100. And go on, what the heck. Yeah, running it a few times, see how it's always ordered by key. Anyway, if we want to add a new item, it's the same syntax. Just call the name of the map, put the key in square brackets, and assign it a value. Now, if the key already exists, like it did actually when we just updated A, it'll get this here as its new value. But, if it doesn't exist, the whole thing gets created. Key and value. Well, let's take that for a spin. All right, J equals 1973. And again, Go maintaining its ordering by key. Well, to delete an element, we use the built-in delete function, and inside the operands, we just name the map and tell it which entry to lose. Okay, so we're updating A, adding J, and then printing this to the screen. Then we're deleting J, so let's add another Println here to prove it's gone. So we'll get one line with a J and then one line without. Let's have a look. All right, nice one. There it is in the first line, but it's gone in the second. Right, I've got a few more things I want to show you before we do a recap.

Miscellaneous

Okay, just a couple of quick things before a recap. Like we just saw with slices, maps are reference types. This means when we pass them to functions, they get passed by reference. Which in turn means, any changes a function makes to the map, well, they're obviously visible to the caller, but also any other functions. In layperson terms, if a function modifies a map, it really does modify it, like it's not acting on a copy or anything. No, functions modify actual map data. Well, a knock-on effect of this behavior is that maps are cheap like you can have a huge old map, but passing it around, cheap as chips. Because remember, you're not really passing the map, are you? You're only passing pointers. And on the topic of performance actually, it is often considered a good practice to specify the size of a map, especially if it's a big one. I mean, you know have to, and Go can resize them on the go, but there is a cost to re-sizing large maps. As well right, and it's probably out of scope for this course, but maps are not thread safe, so they are not safe for concurrency. Basically, it's not defined what happens to them if written to simultaneously. But you know what, that is enough of that, time for a quick recap.

Recap

Then, what did we learn? Well, first up, we learned how to declare and initialize maps, so we can use the make keyword pretty much the same as we do with slices. And as Go is strongly typed, we've got to tell it the type of the key and the type of the value. Anyway, then we confused things over ordering. So, as of Go 1.12, which is kind of a while ago, but maps are basically key ordered unless we're retrieving them through a range loop. We also learned how

we can insert, update, and even retrieve individual elements using the name of the map followed by the elements key in a set of square brackets, and we can use the built-in delete function to delete elements. Well, I reckon that's probably it. Oh, no, actually, look, let's not forget we just learned that maps are reference types. So when they get passed functions, they get passed by reference and not value. And now, we definitely are finished. Though, next up, structs, proper cool stuff.

Working with Structs

Module Intro

Okay, we've seen a few of the different types of Go supports. Obviously, we've worked a bit with strings and ints. But arrays and slices, and we've just done maps, these are all types as well. So, strings and ints are like the basic types, then pointers, slices, maps, even functions, actually, are reference types. And I know, right, we've mentioned that as we've been going. But I don't actually think we have for functions. I mean, I know we've done functions, but we've really only scratched the surface. Anyway, despite all of these types, there's still going to be times when you need something a bit more specialized, and this is where structs come into play. Anyway, we'll go about the module like this. We'll wrap our heads around the high-level stuff. Actually, then we'll give a mention to object-oriented programming from a Go perspective. Then we'll get our hands on defining structs and seeing them in action, and then we'll finish with a recap. Come on.

What Is a Struct?

Right then, at a high level, structs are how we define custom data types, and like I said, they are the bizzo when the standard data types don't quite do what you need. Anyway, the classic example for needing a struct is a geometric shape, like, I don't know, a circle. Well, clearly, none of the standard types are fit for a circle. Like, to accurately describe one, we'd probably want properties like radius, diameter, and circumference. Well, halle-freakin-lujah, structs let us define a custom type with all of those fields, and we can even call it a circle if we want. Then, when that's created, we can reference it or create variables like we do any other type. Now, yeah, I know we don't need all of those values for a circle. We can infer some of them. But, you get my point. What we're looking at right now is a custom type, perfect for defining and working with circles. Now then, just for clarity, although they actually are in this example, the fields making up a struct do not need to be all of the same type. Okay, anyway, the point is, well, two actually. Firstly, this is only defining the type. We're not instantiating any variables here. So, we're basically saying, hey, Go, define me a new type called circle with all of these attributes, but don't actually create me one yet. Well, then, the fields that comprise it, yeah, they're named and they have set types, but we can pretty much have as many of them as we need. So, like, I don't know, if we created a struct to define a new type called person, we could have, well, honestly, loads of fields, but things like name and age, preferred pronouns, eye color, hair color, whatever. We've pretty much got free reign. And I reckon that's the basics. Now, we are going to get hands on with them in a second, but I do just want to make a quick side step to mention object-oriented programming.

Object-oriented Programming and Go

Okay, so a very quick sidestep here, and, look, you know what, if you're not already into object-oriented programming, honestly, just hit that forward button down there, save yourself a couple of minutes. Still here? Okay, well, look, just in case you're not an object-oriented programmer and you ignored my advice and didn't hit the forward button, well, object-oriented programming is pretty much an application of the old UNIX philosophy of creating smaller, modular, reusable components, objects, yeah, that specialize in doing one thing, but then bringing lots of those specialist objects together to form a useful application. Anyway, if you've got experience with OO, then I'm sorry to say, Go does not have an object type, and, actually, it doesn't have a class keyword either. So, while structs might look like we're heading towards OO with Go, I'm really sorry, we're not. Because they're just not objects in the full sense. I mean, yeah, they're a bunch of named fields and you can associate methods with them, but they don't support inheritance. So, no true objects, no classes, and no inheritance. Basically, no OO with Go, and that's by design. Now, yeah, like all things, you can be clever and make Go do things it was never designed to do, but I wouldn't recommend it. If you're going to be using Go, honestly, leave whatever OO you can behind you, and work with Go the way it was intended. But look, I'm waffling. Structs are not OO in Go, but they are quality, so let's get our hands on with them.

Defining Structs

All right, no more slides. And actually, I am in the struct.go file in the structs folder of the GitHub repo. Now to define a struct in go, we can use the type keyword. Then, we give it a name. Now look, geometric shapes have been done to death with structs, so I'm going with Pluralsight courses. Though to be fair, I've probably overdone them a bit in this course. Anyway, look, we're going to call our new type courseMeta. Then we go struct because it's going to be a struct. Now, actually, we can define new types as simple ints and strings and the likes if we want to. So yeah, even though there's already an int type, we can create a new int type that's a simple int as well. And the reason we do this sometimes is we can assign methods to custom types, basically attaching a function to the type. Anyhow, look, we slapped the fields inside a set of curlys, and of course, we make sure they have unique names. Well, as we're going with metadata for Pluralsight courses, we're having author as a string, level as a string, and then rating as a float64. Now we could have more, but that'll do for now. And now then, I know I'm over-stressing this, but in case it's new to you, all this is doing is defining a new type. So it's called courseMeta, and it's got these properties, we're not actually creating a variable of this type yet. Well, to actually do that, we can go either this or this. Now, both of these declare a new variable called gettingStartedWithK8s, but importantly, instead of its type being an int or something, it's of our new courseMeta type. But again, these only declare the variable, so they initialize all the fields with default 0 values. But there is a difference between the two, using the new keyword here yields a pointer. Now then, we're not going to use either of those, but I am going to leave them on there just for reference. Okay, the composite literal form looks like this. So we're declaring a new variable again called gettingStartedWithK8s, but this time we're initializing it as well. So why not, go on. From the top real quick. We're defining a new type here called courseMeta with three named fields. These are a couple of ways to create the variable of this new type, but they only declare it and they don't initialize it. Well,

down here we are declaring and initializing in one shot. Well, I'll tell you what, let's have a Println here to show its values and we'll give this a try. Okay, not bad. Well, next up, we'll see how to work a little bit more intricately with structs.

Working with Structs

Right then. We referenced individual fields in a struct with the period operator. So, I'll tell you what, let's change this Println here to be this. So, we've got some texts saying Author of Getting Started with Kubernetes is and then we're referencing the author field from the new variable. So, it is variable name, period, and then the field we're referencing. And remember, this is the gettingStartedWithK8s variable of type courseMeta, which is a custom type created as a struct. Boom. Well, let's see if it works. Ah, magic. Obviously, we can use the period to more than just retrieve fields, we can change them as well. So, probably not a good idea, but let's say the impossible happened and everybody fell out of love with the course. I don't know, maybe the rating plummets from 5 stars to 1.2, or something horrific like that. Well, it's just like working with the other types we've been doing all course. So if you've already declared an instance, you can use the equals operator to assign new values. We'll put this Println here to show the new rating, and let's give it a run. Okay, all good, or not so much actually, if you are me, the author of that course. But look, that's it. That is the basics of struct. I mean, obviously, there is a ton more, we could spend all day and fry our brains on the stuff, but for a beginner-level course, that is done. Time for a quick recap before moving on.

Recap

Then we've seen a bunch of Go's built-in types, ints, floats, strings, even arrays, slices, and maps, and that's only a few of them. B ut there's still loads of times when we need something a bit more specialized, and that's where structs to come into play. They are literally Go's way of defining custom types that we can use just as if they're regular variables. So, under the hood, a struct is a new type with an arbitrary number of fields, each field is named and has its own specific type, and each field can be of a different type; they don't all have to be the same. I think in our example, we use the type command to create a new type called courseMeta of type struct, and we used it to store details of a top quality Pluralsight course called Getting Started with Kubernetes. Though, actually, just to be clear, the type was called courseMeta, it had three fields for storing the author of the course, its intended audience level, and a customer rating. Then we created a variable called gettingStartedWithK8s, and the type of that variable, instead of it being a slice, or an integer, or whatever, it was a courseMeta type. Whew! Marvelous! Well, we also saw how to manipulate individual fields with the period operator, and I reckon that was about it, and honestly, that is the basics of structs. Next up, and oh my goodness, our last module, but it is a proper good one, concurrency. See you there.

Concurrency in Go

Module Intro

Right then, our final, final topic of the course, and it's a really good one and a real puller actually. Like, it's one of the major features that brings both developers and projects to Go. But, you know what, it's also a topic that's easily misunderstood. Anyway, as the title suggests, we're exploring concurrency in Go, and we'll go about things like this. First off, we'll figure out what concurrency is, we'll explain how Go implements it, and at some point we're going to compare and contrast it with parallelism, because, yeah, for sure, they are related topics, but they're absolutely not the same thing. Anyway, we'll also get our hands on coding some simple examples, and we'll cover off channels as we go. Now then, looking at that, there's probably going to be a bunch of theory, especially up top. And I just ask that you stick with me on this, because it really is important, but you can also see we will be getting our hands dirty. Okay, well, one last thing before we get going. The aim of the game is to give you a decent picture of what concurrency is and how to start implementing it in Go. But remember, we're only a beginner course, so you're not going to walk away ready to implement concurrency in a major project at work, but, you definitely will be bootstrapped and ready to take your next steps. So, look, as per this plan, let's crack on with the theory.

Explaining Concurrency

Okay. So what even is concurrency? Well, at the highest level it's about creating multiple processes, and I use the term processes lightly. But, it is about creating multiple processes that execute independently. Now, while I do use the term process lightly, I am very serious about the term independently. Like, I do not mean simultaneously and I don't mean in parallel. Yeah, go on, maybe an analogy will help. Though, it is just an analogy, so don't take it too far. Anyway look, most mornings when I start work, I've got a bunch of things I need to do that day. Like obviously, I've got project work to do, like editing videos and planning and testing labs and maybe finishing book chapters and the likes. But, I've also got a bunch of admin stuff as well, like emails and messages that demand answering. So let's say I've got a couple of project-related emails to start the day with. Well, instead of firing off the first one and then putting my feet up and sitting around waiting for a response. You know what? I'll just crack straight on with the next one and then when that's sent and I'm waiting for responses to them. I don't know, yeah, go on. Let's say I finished recording and editing a new module for a video course, so I set it off converting to an MP4 and uploading to Dropbox or whatever. Well, while all that's happening, I'll go and check through a chapter of one of my books that's been translated, you know, making sure all the images are still in the right places and the fonts all line up and all that kind of jazz. Well, okay, I've got three things going here. I'm waiting on responses to a couple of emails, I'm waiting on a video production and upload, and I'm reviewing a translated book chapter. Well, here's the difference between concurrency and parallelism. At no point, am I working on any of those three tasks at the same time. I'm not, like, I wrote one email at a time, and then only after I'd sent them, did I switch to the video editing and uploading. And again, only when I kicked off the upload did I switch to the book translation. And that people, is concurrency. So instead of firing off the emails and sitting around waiting for responses, which might not actually come until later in the week, by the

way, I cracked on with something else, the video work. And then, when that started producing and uploading, instead of hanging around again and waiting for that to happen, which could be several hours with large videos, I cracked on with reviewing the book translation. But remember, I'm just one person, so I only did one of those at any point in time. If let's say I want to work on the book translation at the same time as writing emails and doing the video stuff, well, either I need a major brain upgrade or I need to hire some help. Now one last thing actually, if I'm doing the video work or the book stuff and I get a notification that I've got an email response, then heck yeah, I can break from what I'm doing and I can check that email. But the point is, I do have to break from whatever it is I'm doing at that time. Like, I can't do both at once. So, concurrency is dealing with lots of things at once, whereas parallelism is doing lots of things at once. Well, let's have a quick look at how we sometimes achieve this in computer programs.

Concurrency in Computer Programs

In order to understand Go's implementation of concurrency, I think we need to be clear on a very quick bit of computer science. So, computers have processors, and for this simple example, actually, let's assume it's got a single core. Old school, I know, but stick with me. Well, then we've got an application, and when we run it, the OS creates a process, which is basically the runtime instance of the program. So it's the process that tracks memory, file handles, I/O, all of that jazz. And also, the process starts out with a single thread, threads being what actually run on processor cores and execute the program code. Well, the program starts with a single thread, though it can launch more, but it is the job of the OS to schedule threads on cores. Okay, magic, why are you telling us this, Nigel? Well, it is a common way of achieving concurrency, and you know what, even parallelism, if you've got multiple cores. So, for parallelism, all the OS does is it runs different threads in parallel on different cores. You know what, though, back to concurrency. The orange thread here could be the first email I sent in the morning, then this blue one could be the second, the green one, the video processing, and then whatever this is, plum? I don't know, but it could be the book translation. So, four threads or four things on the go, but only a single core in my head or on the processor, so basically only one of these actively working at any one point in time. And that's threads, and, yeah, they're pretty low level, they're kind of complicated, they're actually a bit resource hungry, and if you're not careful, they can drag down performance. Well, it's just as well that Go doesn't use threads for concurrency. Wait, what?

Go's Concurrency Model

Okay, hang on, Nigel. What is the point of talking about threads if Go doesn't even use them? Well, explaining concurrency the way we just did with threads, it's actually useful for understanding the concepts of concurrency. But, Go does actually use threads; it's just, it abstracts them through a lightweight construct layered on top. Well, this lightweight abstraction is called a goroutine, and if you know your computer science, they're a bit like green threads, or they are, at least from the fact that they're not scheduled by the OS; it's the Go runtime that schedules them. Fair enough. But why use this model instead of traditional OS threads? Well, a bunch of reasons actually. So, goroutines are way lighter than OS threads. I think the initial stack size, it's tiny, like 2K, so way smaller than around a MB for OS

threads. Plus, they can grow and shrink as needed. As well though, Go does all of the heavy lifting with goroutines, and that is a mahoosive bonus. Because it means you and I as developers, we can just crack on with our app code; we don't need to learn about OS-specific thread stuff. As well though, switches are expensive with OS threads. Well, they're expensive with goroutines as well; it's just they happen way less with goroutines. So Go layers goroutines on top of threads, yeah. And then when a goroutine blocks, like maybe it's waiting on network I/O or something, no sweat. Go just swaps it out for another goroutine, but running on the same thread. And, I mean, yeah, of course there's overhead with doing that, but it is massively less than scheduling a full new thread. Whew, now, a footnote here. Even with goroutines, there's going to be block conditions needing a brand-new thread. It's just that happens way less often. And as well, goroutines have faster startup times and, thanks to channels, multiple goroutines can easily and safely communicate and share data. Whew, look, the long and short, goroutines let us switch a lot of work on and off a single thread. And I think I've said a few times, they work threads to within an inch of their lives. And you know what, due to the cost of threads, if you can do something on a single thread, I'm telling you, you absolutely should. I mean, way too many cycles get lost into the black hole of queuing and swapping threads on and off processor cores. Look, when it comes to threads, less is definitely more. Now then, as we wrap the theory, just in case you're interested, the concurrency model used by Go is the actor model, or, if you like to sound technical and you want to impress people, it's a modern implementation of the communicating sequential processes model, CSP, if you like your acronyms. Anyway, though, in this model, actors safely pass messages between each other via channels. Well, in Go, actors are goroutines, and channels, well, they're called channels. So, I suppose we need to explain channels. I'll tell you what, trying to keep this brief is harder than I thought. But look, channels are just like pipes. One goroutine puts data onto the channel, and another goroutine grabs it off, simple. And you know what, for now, I'm just going to leave channels there. Let's go and see how some of this stuff actually works.

Writing a Concurrent Program

Right then. We've got a simple program here, package stuff, imports, a main function, and a couple of anonymous functions. So as things stand, execution comes in at main, then our first anonymous function gets called. Though actually, I don't think we've actually mentioned anonymous functions. Yeah, no, I don't think we have. Well look, Go let's you create functions without names, so nothing in the operands here. And then sticking an empty set of operands at the end makes it self executing. Anyway, program flow will come in, and we'll hit the sleep here. That will put the entire program to sleep for 5 seconds. Once they're up, we'll print Hello to the terminal and drop back into main. This next anonymous function will be called, that'll print Pluralsight to the screen, and then as there's nothing else left to execute, the program will exit. Net will get Hello Pluralsight. Now look, I know it's far from special, but it's program flow that we're interested in, so execution is top to bottom, one line at a time, one function at a time. So, when we hit the sleep here, the entire program blocks and we do nothing for 5 seconds, literally nothing. Well, when the 5 seconds are up, we're back in business stepping through one line at a time. Well, I'll tell you what, before we do anything else, we should probably see it. Okay, so this is waiting for the 5 seconds. All right. Okay. Tell you what, let's go and add some concurrency. And you know what? It could not be easier, just slap the go keyword here in front of each function, and it makes them into goroutines. And as we'll see, it

totally changes flow. So going from the top again, we step through a line at a time until we hit our first anonymous function. But, that's a goroutine now, so this time when we hit the sleep, only this goroutine blocks. So, flow picks up here with the next anonymous function, which is also a goroutine, and that gets switched onto the thread and starts executing. And of course, it's fast, yeah, like, it is not going to take 5 seconds, that's for sure. So, we'll get Pluralsight printed to the screen and that goroutine lets it. When the 5 seconds is up, this one will resume execution so get back onto the thread and it will print Hello. Net result will get Pluralsight Hello. Now there's one more thing we need to do before we can actually run this, func main here is also a goroutine, it's the main one, so as soon as it exits the whole program does, and for us, that means our main function is going to exit before any of this completes. Well, definitely this one here with the 5-second sleep. So, we need to use waitGrp from sync package so that our goroutines have a way to tell main when they're finished. So we'll import sync up here and then we'll add these lines here. Okay, then we'll add 2 to the waitGrp, so one for each goroutine. This way main's going to wait until both report back before it continues and exits. Anyway, look, that should make those good to go. So, remember to save your changes, and we'll give it a try. Now remember, we're expecting to see Pluralsight, then a bit of a pause, and then Hello. Okay, sleeping for a few seconds, and there we go. Quality. So our code executed top to bottom, one line at a time until our first goroutine fired. At that point, execution continued and a second goroutine was created for this function here, but it didn't get any time on the thread until the first one blocked on the 5-second sleep. Well, as soon as that happened, the sleeper got bumped off the thread, and the second one got its moment. It punched out Pluralsight and it reported back to waitGrp that it was done. After 5 seconds was up, the first one here came back to life, printed Hello to the terminal, and also reported back that it was done. Then at that point, main continues. Well, there's no more code to execute and the program exited. Brilliant. Though I know, not rocket science, but not bad at explaining the high-level concepts of goroutines. And it shows a potential pitfall, our text came out back to front because of the delay in scheduling. So, you need to keep that in mind when working with concurrency. Anyway, let's take a look at channels.

Channels

So we mentioned before that Goroutines use channels to safely share data. Well, there's two types of channel, buffered and unbuffered. And unbuffered channels can't hold data. So any Goroutine putting data onto one blocks until there's a receiver on the other end. I guess kind of forcing synchronous behavior. Well, on the other hand, buffered channels, they can hold data, so a Goroutine can drop data onto it and crack straight on with whatever it was doing without having to care if there's a receiver on the other end. Okay, marvelous. Well, to create an unbuffered channel, we can use make with the chan keyword. And then, obviously, Go wants to know the type of data the channel is going to hold. Remember, Go is obsessed with types. But, you know what, for an unbuffered channel, that's it. But to make it buffered, all we do is add buffers. So this one here will have 5, and because it's for int, it means it can hold up to 5 ints. Well, like we said, the effect of a buffered channel is that Goroutines using it don't need to block. In fact, actually, as long as the channel is not full, other Goroutines can come along after and can also put data on it. So proper asynchronous behavior. Now, obviously, if a buffered channel is full, then, for sure, any Go routines wanting to use it are going to block until it frees up. And then, likewise, if a receiver is trying to grab data off of a channel and the

data's not there yet, well that blocks until it is. And you know what, that's enough for a taster. Let's go and wrap the module.

Recap

Well, that's the module. In fact, if you've been following along, that's the course pretty much. Anyway, concurrency. We learned that it's about spinning lots of plates, so dealing with lots of things, but it's not about doing lots of things at the same time. That's parallelism. So I think I said I can fire off a bunch of emails, and while I wait on responses, I can crack on with other stuff, maybe edit a video. Then, if I reply to an email that comes in, of course, I can park the video stuff and I can reply to the email. Then when I'm done with that, I pick up the video work again. So yeah, like we said, we're only doing one thing at a time, but we're actually dealing with lots of things. Anyway, Go has native support for concurrency via goroutines, which are just regular old functions prefixed with the go keyword. Though, you've got to be careful of the scheduling and timing impact that can have. Remember, we saw Hello Pluralsight printed as Pluralsight Hello because of some blocking and scheduling. Well, under the hood, goroutines are built on threads, but they're more lightweight and they are way easier to code. They also use a construct called a channel to share data, and we just said that channels can be buffered or unbuffered, with the buffered kind allowing asynchronous behavior where senders can drop data onto the channel and crack straight on with whatever else they're doing. Well, that's us just about done. Join me for one last super quick module to discuss where to go and what to do next.

What Next?

What Next?

Right then, we're done. Amazing, yeah? And you know what? Massive congrats if you've made it all the way through. You should be rightly proud of yourself. And you know what else? If you've been with me from the start, honestly, thank you. I genuinely appreciate your time. But as well, looking at the stuff we've covered, you should have a decent grasp of Go and some of this stuff that it can do. But we've only scratched the surface, and I'm telling you, there isn't much that Go can't do. Now look, if you like what you've seen, then honestly, check out all the other stuff we have got here on Pluralsight. We've honestly got loads, and we're adding more all the time. And you know what? It's not just video courses, I always recommend you get your hands on as much as possible, and then any tests, and self-assessments, and things that we provide, seriously, you should do them and don't stress about them. Just enjoy them. They're always good fun. Anyway, away from Pluralsight, and I suppose aside from the obvious of get your hands on and hack a few of your own projects, but get involved with the community. So things like conferences, I love them, they're always a great way to learn new stuff and meet great people. So maybe search for gophercon and golang conferences, and you know, what even golang meetups. There's probably a meet up close to you. So yeah, check out our videos, and our hands-on, and our tests, and the likes, get yourself a project to take your skills to the next level, and get involved with the community. And I reckon that's it. So just a final thanks from me. Honestly, I've invested a ton

of effort into this course, so I hope you liked it. If you're into the infrastructure side of things, then I highly recommend my containers and Kubernetes courses. Just search my name on Pluralsight, I reckon you'll find me. But yeah, honestly, it has been an absolute pleasure spending time with you. I'm Nigel, and have fun coding.