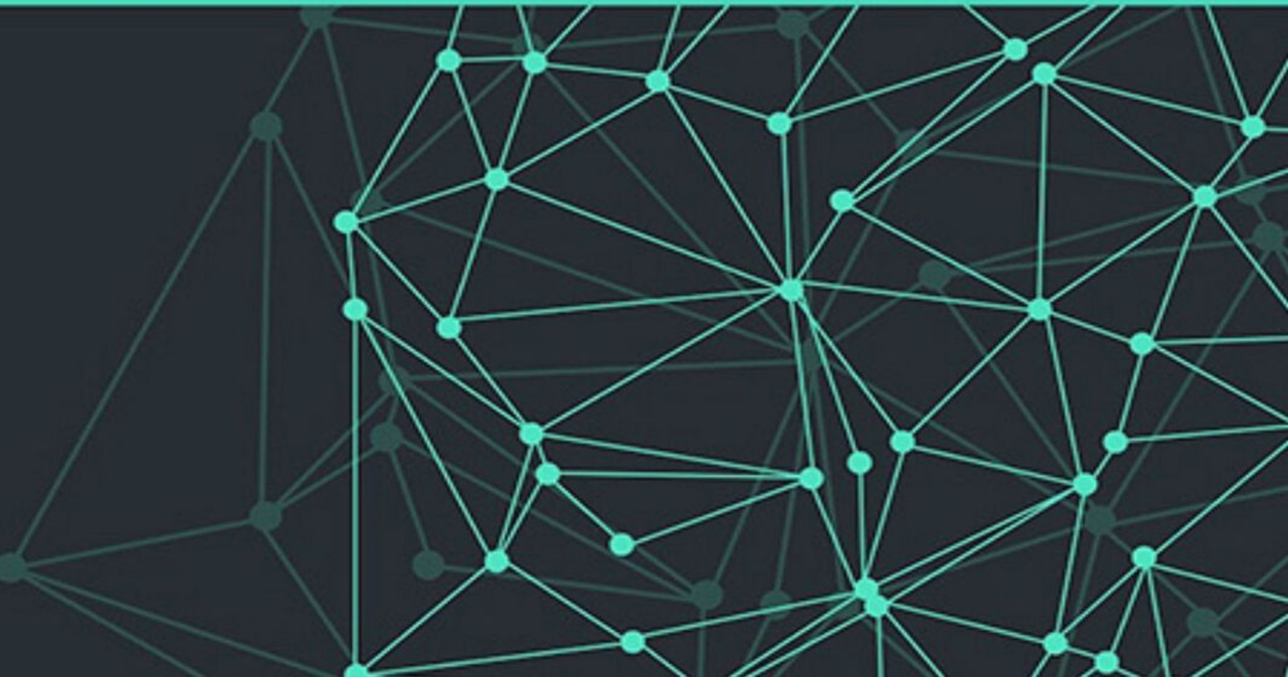
mEm Inc

Third Edition

Go programming language

The Ultimate Beginner's Guide to Learn Go Programming Step by Step

JOHN BACH & ALEXANDER ARONOWITZ



Go
programming language

The Ultimate Beginner's Guide to Learn Go Programming Step by Step

By John Bach

3 nd Edition

"Programming isn't about what you know; it's about what you can figure out . " - ***Chris Pine***

MemInc.com

Introduction ……………………..………………5

Chapter I

General concepts……………………………….6

Chapter II

Understanding go……………………….………19

Chapter III

Writing the first program……………….……….34

Chapter IV

the rules………………………………………..53

Chapter VI

Create Backend……………………..…………84

Introduction

You may have heard in the last few years about a new programming language that originated from within Google called Go (or Golang as a searchable term for search engines), through this book we will try to identify this language, its advantages, disadvantages and what makes it different from others. The first chapter of this book will be a verbal lesson only, focusing on the points of difference of language with the rest of the languages, and is directed to those with some programming background with the rest of the languages, but the rest of the lessons will be directed to beginners.

Chapter One

General concepts

As usual every decade, there are new tools and techniques that try to take advantage of the mistakes of their ancestors from tools and techniques. And "What is the need for a new programming language?" Or "Why don't they agree on a single programming language?", A programming language is just a tool for your tasks, each of which is more appropriate for a task. Choose for each task the most appropriate tool, and there is no need to sanctify the tool and the establishment of war wars - non-technical - around them or create an imaginary obsession prevents learning.

A team of programmers from within Google felt it was time to improve their workflow in C and ++ C, and they needed a new tool that eliminated the flaws of those two, further improved their productivity, and was relevant to the quality of Google's computational needs. They founded the Go programming language within Google in 2007. The long established team of the language includes Unix's founder, Ken Thompson (a computer science, Ken) who worked for Bell Labs, and Robert Pike, who was also part of Bell's Unix team. Labs, one of the founders of the UTF-8 codec, and finally Robert Griesemer is a V8 JavaScript engine worker.

Knowing this simple historical overview of Go, I like to kill you all overly enthusiastic, I don't like you to love programming language because of its founders, or it came out of a company you love, so I will start to list things you may not like about the programming language Go, even if you did not like it , You might want to stop reading. I would be personally happy to see very few readers of this series really knowing what they are doing, rather than a momentum enthusiastic about everything new without awareness and knowledge of things.

Things you may not like about the Go programming language

1. Object-oriented is not

If you think it is, there is no concept of Class or Objects and therefore there is no inheritance. However, Go carries some of the advantages of object-oriented programming such as the provision of Interface, Structural Functions, and Struct composition.

Why? The founders of language believe that object-oriented languages carry many flaws and complexities that can be eliminated by abandoning some of these concepts altogether. Even Java programmers themselves recommend Composition, for example, more often than inheritance, language founders who believe that object-oriented programming is often a bad idea, technically.

2. No Exception Handling handled

This may be a corollary due to the absence of Go's concept of object programming. Go errors are handled in a fairly traditional way, since errors are returned as normal error values. Where error is a primitive type in itself as any other native type (int, string .. etc). However, Go allows you to throw an error for exceptional cases via the keyword panic (similar to raise or throw in other languages) as well as recover from these errors by recover.

3. No default or optional arguments can be passed to functions (default / optional arguments)

In other programming languages, you might be used to doing something like:

```
function listFolders (path, subfolders = false, recursive = false) {...}
```

In Go, however, you cannot pass subfolder = false nor recursive = false as a sign to the listFolders function because it will not accept such default / optional parameters, resulting in a compile error.

Why? Language founders believe that these behaviors contribute to building unstable APIs or make their behavior unpredictable. In our previous example, for example, they prefer to write the function without default operators, ie:

```
func listFolders (path string, subfolders bool, recursive bool) {...}
```

This forces you to write the behavior you want from the function explicitly instead of letting the code interface dictate you to act as the default, this to reduce human errors. This may also prompt you to write three functions, each with its own distinct behavior, for example: listFolders, listFoldersRecursivly and listFoldersWithFirstLevelSubFolders.

4. No Overloading feature

For the same reasons, there is no Method overloading, ie you cannot redefine a function with the same name but with a different signature. For example, if there is a function named:

```
func listFolders (path string) {...}
```

You cannot create another function with the same name but with a different signature such as:

```
func listFolder (path string, level int) {...}
```

You should change its name to:

dunc listFolderToLevel (path string, level int) {...}

However, there is an indirect way to make a function accept arbitrary values by making the {} interface type signature we will look at in the next lessons.

5. No generics

Other languages enable you to write generic functions or classes, where you do not declare the type of parameters you accept but leave them to know the type of parameters later when you call them. A type to be determined, so <List> String or <List> Integer can be easily created while maintaining the same functions and operations that can be performed on the List in general.

There is no Go like this, and instead there is a {} interface as a universal type that satisfies all types, but it is not a perfect alternative to Generics.

Why? The reason for the lack of Generics in Go is that its founders have not yet figured out the most appropriate way for them to add this feature to the language without increasing load during run-time.

6. Go is a boring language and is not the best programming language!

Because of the simplicity of language and nothing new, many consider it a boring language. The number of keywords and their original types is small compared to other languages, and they greatly reduce the existence of more than one way of doing a task. It doesn't even have a while loop and is limited to a for loop. Many consider this a feature of language, but I mentioned it to you so you don't expect anything new to show off.

Also, the language will not allow you to leave a variable without using or importing an unused import / variable and the compiler will never accept it.

7. Stubborn language

The founders of language are firm in their opinion and decisions in the design of the language, do not expect significant changes that may occur in the short or medium term in the language or changes in the way things do and the behavior of the compiler. There is no need to open blank discussions about language design and its flaws unless you are at the same level of experience and wisdom.

They themselves declare this, and remember that there are other options and programming languages if you don't like Go.

8. There is no consensus on a single package manager

Python has pip, javascript has npm, and other languages have a popular or agreed package manager, Go is not without a package manager, it has a lot of it, but it has not yet agreed on a single package manager or how to get and list the dependencies In a standard way, but recently it is done through the concept of Vendoring.

These eight things, for the male and not limited to the harshest criticism of Go as a programming language, if you agree it may not suit you language, and if you feel that behind it wisdom - like me - continue reading the chapter on things you may like about Go.

Things You May Like About the programming language Go

1. Light parallel threads (goroutines)

Go allows you to run parallel processing and simply take full advantage of your CPU, as it has the so-called goroutine, a parallel thread that is lighter than thread in other languages. This allows you to release tens of thousands of parallel threads (instead of just tens or hundreds of threads) without having to create them and make them communicate or complement each other. It is enough, for example, to create a function that works in a thread alone by going with the keyword go, for example:

```
go func () {// A nasty function that does things in a thread alone
  ....
}
```

This allows writing high-performance applications much easier.

2. Fast compiled language

Unlike Ruby, Python, Java and PHP, Go is an assembly language. The final output of your program will be a stand-alone executable containing all the dependencies needed to run it without the need for external dependencies. This is very useful in web development, for example, to eliminate or reduce dependency problems.).

This feature also facilitates the deployment of your application (deployment) where you will have to just transfer the executable to the server for example, then run it and enough. No need to install anything else to run your software, no need to even install Go on the server itself.

It may come to mind that compiling can take time, especially if the size of the program is large, but the fact that Go is very fast and often you won't even notice it.

3. Productive language

Because it's a boring language, it narrows you to argue with your co-worker or someone working with you on the project or looking for the best-amazing way to do something, everything is simple and clear.

4. Multiple platforms par excellence

The same software can be reassembled to run on Windows, Linux, Mac OSx or even smartphones and ARM architecture. Even after version 1.5 of Go you can create an executable file for all platforms through the platform you are in, for example through the Linux system can create an executable file for Windows and Mac OS without the need to be on that system.

5. Rich Standard Library

Despite the novelty of language, its standard library has made progress in a standard circumstance. It is often advised to rely on the standard library as much as possible, as it contains many things needed by most types of applications, and even has an HTML template engine, tools for dealing with json encoding, encryption of passwords and other usable things in web applications, for example.

6. Unicode everything by default

The subject of encoding strings and how to handle the character digitally is great, but Go has made everything unicode by default unless otherwise specified explicitly. Go has a rune type instead of a char type in the rest of the languages, which is simply an abbreviation (alias) for the int32 type that only points to the Unicode encoding point for that character. So the string in Go is a rune string, not a char string that refers to byte as in C, for example.

7. Good built-in tools

The documentation of your software can be generated by simply typing a godoc command, improving the formatting by typing gofmt, or starting code testing procedures (for example, unit testing) by typing go test, all tools that come with the Go installation.

8. Easy language in terms of writing and formatting (Syntax)

The format of the language is somewhat similar to Python, where there is no semicolon after the end of each instruction (;), and a new variable can be made without specifying its type. : = 35 The int type will be assigned to the age variable. Also, functions can return more than one value, such as in Python.

Perhaps the most different thing to other languages is the method of writing functions, since the value returned by the function is written to the right of the function (as in Pascal) and not to the left, for example:

```
func HelloHsoub () string {
   return "Hello Hsoub Academy!"
}
```

Note that string is written to the right of the function, not to the left of it, that is, it returns a value of type string. Functions can also return other functions as a value. Variables can also refer to functions (such as in JavaScript, Python, etc.).

```
sayHi: = HelloHsoub
```

So as not to take sides, these are 8 things you might like.

Chapter II

Understanding go

In the previous chapter, we learned about Go and its differences with other programming languages. In this chapter, we will try to explain how to install it, configure and understand its workspace. It is very important to understand the folder partitioning that the Go environment dictates to you.

Go has two types of compilers. As a quick reminder, the compiler's job is to take your code in its purely textual form, and turn it into an executable (or library) program. One of the Go compilers is the standard compiler named gc for Go Compiler, and the other is a compiler that relies on the gcc compiler and what it provides as a platform and support for more CPU architectures called gccgo.

It should be noted that the gccgo complex is somewhat late in implanting the full specifications of the newer version of Go. For example, in writing this lesson, gccgo provides version 1.4.1 of Go, while gc always provides the latest version (1.5.2) of the language.

I mentioned this for the sake of knowing, in short, if you are not interested in supporting more architectures or some details of the gcc platform you will not need gccgo and you will be satisfied with the standard gc which we will follow throughout this series.

Install Go

As mentioned in the previous lesson, Go is a multi-platform language par excellence, ready to download and install on your preferred system for both 32bit and 64bit architectures.

Install Go on Windows and Mac OS

Simply go to the language site and then download the version of your system. Its installation process is normal and is similar to the installation of any other regular program.

After installation, make sure the operation was successful by typing go version or just go on the command line. If you're on Windows, make sure to close any previously opened command-line window (if any) and then open a new window.

Install Go on Linux

I advise you to check your package manager before downloading and installing Go from anywhere else. If you find that your distribution package manager provides the latest version of Go, it will be much easier.

If you are on Ubuntu and the distributions based on it, you probably think that the Go version provided by the apt package manager is outdated, since the distribution is updated every 6 months or two years (if it is LTS). So I advise you to install by adding an etherneum repository.

sudo add-apt-repository -y ppa: ethereum / ethereum

sudo apt-get update

sudo apt-get install golang

The rest of the distributions are supposed to contain the latest version of Go, then it is enough to install from the package manager, for example on Arch Linux:

```
sudo pacman -S go
```

After installation, make sure that the installation was successful by typing go version or just go on the command line. You should receive a similar result to this:

```
go version go1.5.2 linux / amd64
```

Understand the work environment

The working environment here is two basic:

- A folder on your computer, created anywhere you want, will be a place for all Go projects.
- An environment variable called GOPATH refers to this folder.

Suppose you name the work folder as work, which should also contain three other subfolders:

- A sub-folder called src, where you will place all the code for all of your Go projects.
- A subfolder named bin, where the gc aggregator will transfer executables generated by your code directly to that folder.
- A subfolder named pkg, such as bin but not for executables, if you write a program that is a library and not an executable program itself, or fetch an external library via the go get command, it will be compiled as a library that is placed in that folder so that it can be easily fetched in other programs without reuse. Compile them each time (this speeds up the process of compiling your executable that uses these libraries).

Given the above, the Go environment will be similar to this tree division:

```
work folder
├ ── bin # The resulting executables folder
Kg pkg # The resulting program libraries folder
└── src # Our code folder
```

Is that all? of course not.

Since Go is a relatively recent language, it assumes that you will host your code in a place (repository), which is usually any professional programmer, fearing that the code will be lost or shared with other people. Of course, your code repository will be managed with a version management tool (VCS) such as Git or Mercurial. The code may be on your own server, or on a software hosting service such as Github and Bitbucket.

Since Github is the most popular platform for sharing code, I'll give it an example.

Go prompts you to place your code in a path similar to your github account, for example, and even close the image. Suppose your github account is called HsoubAcademy and your account link is:

https://github.com/HsoubAcademy

If you create a Go project named example for example and then upload it to Github using your HsoubAcademy account, the project link is:

https://github.com/HsoubAcademy/example

Go will prompt you to have the project folder on your computer similar to this link, ie under this path:

work / src / github.com / HsoubAcademy / example

Where:

github.com is a subfolder under the src folder.

HsoubAcademy is a subfolder under the above github.com folder, for example, change it to your Github account.

example is a subfolder that contains your project files under the HsoubAcademy folder.

Note: Folder names are case-sensitive except for Windows.

Why arrange the work environment like this?

Simply because organization in this way is very useful in two ways:

1. Any external library can be fetched by simply importing it from its repository path. For example, suppose that we want to use a library to decompress the Arabic text, we will use the goarabic library, which will be enough to include in our program.

```
go get github.com/01walid/goarabic
```

2. The go get command places the output of this library in the pkg folder with the same path as its link on github, ie it will be placed in a

subfolder named 01walid inside the github.com folder which is a subfolder within the working folder, so it can be included directly in our project. the shape:

import "github.com/01walid/goarabic"

Note that the path to import this library is the same as its link on github.

This makes it easy to share code libraries and enrich the language libraries themselves. It is very important that your project follows the same curriculum, especially if it is a library.

Configuring the work environment

For Linux or Mac OS, for example, you can create a folder named work inside your Home folder by typing a command:

mkdir $ HOME / work

Then make the latter a Go work environment by creating an environment variable named GOPATH (case sensitive) that indicates:

export GOPATH = $ HOME / work

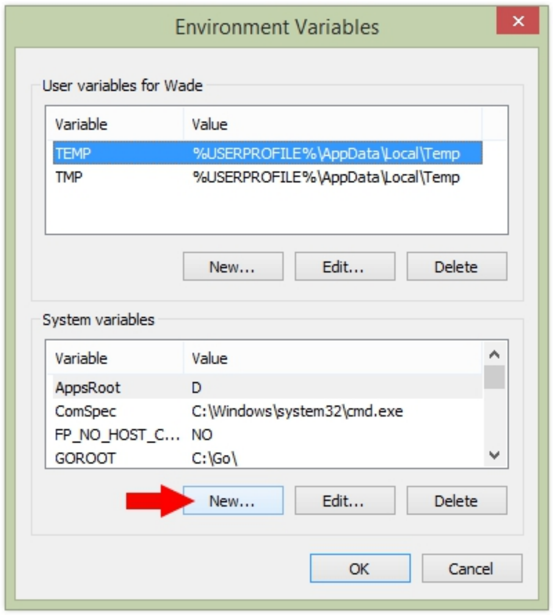
Of course to make this variable permanent even after rebooting, you will have to type it in bashrc. (If you are using a bash shell, or a zshrc file. If you are using a zsh shell, for example, files that are inside your home folder HOME $).

If you're on Windows, just add a variable called GOPATH by going to:

System Properties (System -> Advanced system settings)

Then click on Environment Variables

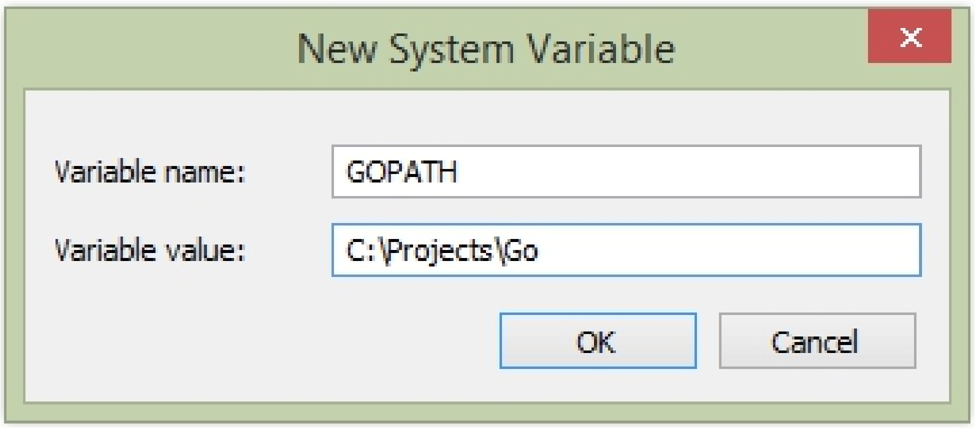
Then add the variable from the System variables section as shown below.



Variable	Value
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

Variable	Value
AppsRoot	D
ComSpec	C:\Windows\system32\cmd.exe
FP_NO_HOST_C...	NO
GOROOT	C:\Go\

For example, if your work folder in Windows is C: \ Projects \ Go, you should add it like this:



Variable name: GOPATH
Variable value: C:\Projects\Go

If the selected path for your work folder is shown, you have successfully installed and configured Go.

Bonus

What if we can make the executables we write or download via the go get command available directly to us in the command line window without browsing them every time?

For example, let's say we want to use the ha tool written in Go, to convert Markdown files into HTML ready to be pasted into another editor, we bring the tool:

```
go get github.com/HsoubAcademy/ha
```

Since the ha tool is an executable program, the go get command will move it directly to the bin folder within the work environment.

But we don't want to navigate to the bin folder inside the working folder to use it, just want to type ha in the command line directly to take advantage of the tool and use it even if we are in a free folder.

Since Go places the output of the executables in the bin folder under the working folder, it is enough to add the bin folder path to the PATH environment variable paths so that you have everything! Even your own executive programs you write yourself.

For Linux or Mac OS users, this is possible by adding the bin folder path within the working folder to the PATH environment variable:

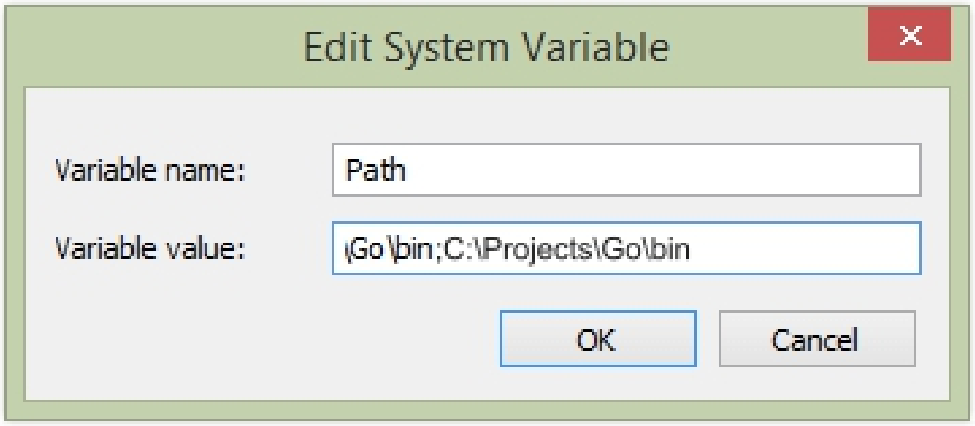
```
export PATH = $ PATH: $ GOPATH / bin
```

For Windows users, add the bin folder path within the working folder to the PATH environment variable by going to:

System Properties (System -> Advanced system settings)

Then click on Environment Variables

Find the Path value in the System variables section and press edit and add the path to the end of the power preceded by a semicolon (;) as shown below.



Variable name: Path

Variable value: \Go\bin;C:\Projects\Go\bin

Now you will find that everything in the bin folder of programs and tools is available to you, including the ha tool we mentioned earlier (try opening a new command line window and typing the ha command).

Chapter III

Writing the first program

What does Go programming look like? How do I run a program written in Go? This is what we will learn during this chapter, and before continuing the lesson, I like to make sure that you followed the previous two chapters first.

Welcome to the world

We will try to follow the habit of learning programming languages, which is simply to write the phrase "Hello World" (in English "Hello World") in the language to be learned, simply this is what the program will appear in the Welcome to the world in Go:

```
package main

import "fmt"

func main () {

    fmt.Print ("Welcome to the world \ n")

}
```

Place this in a file named helloworld.go inside a folder named hellogo, for example, under Path:

```
$ GOPATH / src / github.com / YOUR_USER /
```

Where GOPATH $ is the path to your work environment, see Lesson 2 to understand how it works.

Change YOUR_USER to your username on, for example, github.com or a code management service.

So the final path to the new file we created will be:

$ GOPATH / src / github.com / YOUR_USER / hellogo / helloworld.go

All you have to do now is navigate to the file path and type go run helloworld.go

```
$ go run helloworld.go
Welcome to the world
```

If your command line terminals do not support Arabic, you may find that the Arabic script is inverted and the letters are intermittent.This is fine at the moment, but note that Go has accepted the Arabic characters inside the code text and handled them normally, because everything you consider unicode by default.

Let's explain the program line by line:

```
package main
```

Simply:

Each Go code file must belong to a package.

Each Go package must belong to a folder.

Two packages cannot exist at the same folder level, but several files can belong to the same package (that is, a folder with multiple files), and a package can exist within another package but each in a subfolder separately.

In our previous example, we gave main as the name of our package, which is a special name, in which the compiler of the Go language treats this package as the program entry, that is, the instructions in this package are run first. We preceded the package name with the keyword complete package.

import "fmt"

import: another keyword meaning "import" or "collect" the Doe library or Doe package.

"fmt" is an abbreviation for format or formatting, a standard library that comes with the installation of Go, for building and printing text. Note that the name of the libraries or packages to be imported is always enclosed in quotation marks "".

func main () {

Here we created a function called main :

To create the function, we used the func key, a space, and the function name.

Like the main package, the main function is also treated specially by the compiler, where it is considered the entry point of the program (Entry point), that is, the instructions in this particular function, specifically within the main package, are run first, unless there is another function Called init (), which we will discuss in other lessons.

Like Javascript and other languages,} opens the function body.

fmt.Print ("Welcome to the world \ n")

Here we used the "fmt" package we imported above, to use any package in Go enough to write its name, then a dot, and then the name of the function you want to use from that package, in this case, we simply wanted to print the text "Hello world" and go back to the line, So we used the Print function and sent it the value "n \ hello":

We passed the value in quotation marks "" because it is a string.

We ended our text string with the \ n tag, a special tag in most programming languages, meaning "new line", which we used to return to the line.

We pass the values and coefficients of functions by placing them in parentheses directly next to the function name.

}

We closed the main function object using the {sign, which means that the instructions for this function have ended.

Notice the absence of the semicolon ";" After the end of each instruction, it is not needed in Go.

The concept of Packages in Go

Packages are a simple way to split your program into smaller pieces divided by purpose or function. Packages can be referred to as "libraries" or modules.

Go follows the following rules in how to split a program into packages:

A package is often a folder within your project that contains files named after the package.

You must have at least one package in any program or library.

If the program is an executable program (not a library), then a package called main (ie package main) should be the program's entry as seen in our first example.

As a reminder of the above:

Each Go code file must belong to a package.

Each Go package must belong to a folder.

Two packages cannot exist at the same folder level, but multiple files can belong to the same package (that is, a folder with multiple files).

A package can exist inside another package but each in a subfolder separately.

The main folder of your project can contain one package, only one, the rest of its packages can be located in subfolders.

The absence of a main package from a program makes it a library only and not an existing operating program itself.

When writing packages, any function, variable, or structure name that begins with a capital letter (Uppercase) means that this function / variable / structure is generally available to all who import this package via the import keyword. Any lowercase function / variable name means that it is private and will not be exported (will not be available) to other packages and programs.

In our previous example, the "fmt" package provided a print function to us, noting that Print starts with a capital letter U (Uppercase). It was just enough to write [...] fmt.Print

Welcome to the library

What if we want to say "Hello world" in more than one project? Perhaps it would be better to make "Hello world" printing in a separate library that we import into other projects. This is just an example of how to build a library in Go. Of course, you don't need a library. All you do is "Hello world."

Let's try to write this in a simple function, call it SayHello and leach it in a special package, call it sayhello.

Let's build this library in a separate project, to be a library that can be brought in more than one program or other project. We will call the project sayhello, ie the same package name to facilitate it. We will not need a package main because we will build a library only and not an executive program itself.

So create a new project folder under Path:

```
$ GOPATH / src / github.com / YOUR_USER /
```

The same way that we explained earlier.

Create a file named sayhello.go under the sayhello folder, so the file path looks like this:

```
$ GOPATH / src / github.com / YOUR_USER / sayhelo / sayhello.go
```

Now, open the sayhello.go file with your favorite text editor and then start writing our amazing function.You are not supposed to understand everything right now. The purpose is to explain the concept of packages and libraries, yet we will try to explain the instructions of this function:

```
package sayhello

// SayHello returns Hello World in Arabic.
func SayHello () string {
    return "Welcome to the world \ n"
}
```

Note that our function is not much different from the "Hello world" program we first wrote:

We used package sayhello instead package main.

We called the func () function SayHello instead of (func main). Note that we made the function name begin with a capital letter (Uppercase) because we want to export this function and make it available to other projects that retrieve the sayhello package.

We have declared that the function returns a text string by typing the keyword string to the right of the function.

We returned the text string "n \ hello world" via the keyword return instead of printing it using the fmt library as we did the first time.

We now have "Welcome to the World" in the form of a library! But how do we make it feasible? Ie how do we put it in the pkg folder so that the rest of the projects can bring and benefit from it?

All we have to do is type go install inside the sayhello project folder.

Of course you will not be able to run this library via the command go run sayhello.go like we did or once, because what we wrote this time is a library and not an executive program!

Check out the "Hello world" library

Does our function work well? How can we check our Go programs and packages? Does the "Hello world" library really print "Hello world"? Let's make sure by writing an automated check of this function!

In the Go language:

Unit testing can be written easily. It is enough to create a file with the same name as the file you want to test and add test_.

You can examine a particular function by simply typing Test and then the name of the function.

In this case, in the same path as the sayhello project, create a file named sayhello_test.go next to the same sayhello.go file to test the sayhello package.

Open your sayhello_test.go file with your favorite text editor. All we have to do to try our function is to fetch it and check if the print result is actually "Hello world!" As shown in the following code:

```
package sayhello

import "testing"

func TestSayHello (t * testing.T) {
    greeting: = SayHello ()
    if greeting! = "Welcome to the world \ n" {
        t.Error ("TEST FAILED!")
    }
}
```

In the above code:

By default, we imported the standard "testing" package provided by Go for purposes of creating automated checks.

Since we want to check the SayHello function, we have created the TestSayHello function with its pass-through test.That is, this is a "check state", you don't have to understand this expression right now, you can ignore it. The purpose is to understand how automated checks work in Go.

Within this function, we simply call our SayHello function and check if the value it returns is "n \ hello" if it is, the scan passes safely and we will not print anything, and if not, type "TEST FAILED" using the t operator Of the standard testing package using the Error function, which in turn tells that this scan state has failed.

Now, in the same folder you are in, simply type go test to check our sayhello package! You should receive a result similar to this:

```
PASS
ok github.com/01walid/sayhello 0.001s
```

You can add v- or cover-- to the previous command to print more information about the compiler checks, for example go test -v --cover

```
=== RUN TestSayHello
--- PASS: TestSayHello (0.00s)
PASS
coverage: 100.0% of statements
ok github.com/01walid/sayhello 0.001s
```

Note that we get coverage: 100.0%! This means that our tests cover all the functions we have written in this project (since we have only one function and one test, the result is normal), it is always advisable to keep the coverage rate of tests high in your programs, so as to ensure that their stability is good and reliable.

Create a documentation for the "Hello world" library

We finished our amazing library, we made sure its work was correct, what if we could generate and share documentation of it? You won't need much to do that!

All you have to do is type the command godoc -http =: 6060 and then visit localhost: 6060 on your browser! You'll need to navigate to document your library privately by increasing

http: // localhost: 6060 / pkg / github.com / YOUR_USER / sayhello /

Of course change YOUR_USER to your username, just like the library path in your $ GOPATH environment.

Using the "Welcome to the World" Library

Let's use this library in our first program.

```
package main

import "fmt"
import "github.com/01walid/sayhello"


func main () {
   fmt.Print (sayhello.SayHello ())
}
```

We just brought our new library by typing:

```
import "github.com/YOUR_USER/sayhello"
```

Then use it by calling the SayHello function inside fmt.Print instead of typing "Hello world" directly:

```
   fmt.Print (sayhello.SayHello ())
```

Now run the program again by typing go run helloworld.go, that's it!

Congratulations on your first program and library using Go!

Bonus

In adding this lesson, I chose to refer to some popular script editor plugins, which would make it easier for you to programmatically go with Go.

Here you will find a list of these add-ons.

Conclusion

In this lesson, we learned how to write a program using Go, and then make it a library rather than leaving it in its executive form.

It is important to make your programs a collection of libraries separate by purpose, rather than having everything nested in a single package or in an executable program. As we have seen, creating an executable program is as simple as writing a file with package main, then func main, and invoking the rest of the packages. Good and unit tests.

In this lesson, the library was completely separate from the executive program. Your future projects will most likely be divided into packages / libraries within the project folder itself and not necessarily in another folder / project, but the concept is the same.

We have also seen that comments in Go are an excellent way to generate documentation with minimal effort, so as a programmer you should not neglect it and pay attention to it and its details.

All this was easy! Perhaps you hear these things Vtzaf (tests, docs, cover .. Etc.) But I deliberately introduced these concepts in the first lessons to see that nothing worth running away from. High-quality programs are merely polite behaviors and practices that are in constant steps that you should get used to from now and make them a development / programming method.

In the next lessons we will look at the types of variables, structuring data, how to create a variable, loops and conditional statements.

Chapter IV

the rules

The Go programming language was created to get the job done easily.

Go has concepts similar to Imperative Languages and Static typing. It is also fast in compilation, quick to run and implement. Large and complex.

Go has a great standard library and an enthusiastic and active software community.

In this article, we will explain the basics of the Go language in an easy and simple way, and deal with some important concepts. The code in this chapter is interconnected, but we have broken it down into sections and have titles for these parts, and there are lots of direct comments on the code.

The main topics covered in this chapter are as follows:

- Write comments.

- Libraries and import.

- Functions.

- Data types.

- Named return values.

- Variables and memory.

- Control sentences.

- Generate functions.

- Deferred implementation.

- Interfaces.

- Multiple inputs.

- Error handling.

- Simultaneous implementation.

- Web

Write comments:

To write a one-line comment

```
// single line comment
```

Type a comment with more than one line

```
/ * Multi-
line comment * /
```

Libraries and import

Each source file starts with the keyword packag. The main keyword is used to identify the file as an operational file, not a library.

```
package main
```

To import an office package into the file, we use the Import instruction in the following way:

```
import (
    "fmt" // package in the standard language library
    "io / ioutil" // I / O functions are applied
    m "math" // We use the letter m to shorten the name of the mathematical functions library
    "net / http" // web server
    "os" // functions at the operating system level such as handling file s
    "strconv" // Text conversions
)
```

Functions

Functions are defined by the word func followed by the function name.

The main function is private, and is the entrance to the program's executable file (the Go language uses ornate braces {} to define parts / blocks).

```
func main () {
// To output text on the main output unit (stdout), we use the Println function in the fmt library
fmt.Println ("Hello world!")
// Call a function from the same current package
beyondHello ()
}
```

Functions need parentheses that receive Parameters, and even in the absence of parameters, parentheses are required.

```
func beyondHello () {
// Variable declaration (variable must be declared before using it)
    var x int
// Give value to the variable
    x = 3
// Short definition using: = Includes variable definition, specifying its type and giving it valu e
    y: = 4
// A function that returns two separate values
sum, prod: = learnMultiple (x, y)
// Print and output in a simple and direct
    fmt.Println ("sum:", sum, "prod:", prod)
    learnTypes ()
}
```

The function definition can have multiple coefficients and return values. For example, learnMultiple takes the coefficients x and y and returns two sum and prod values of type Int.

```
func learnMultiple (x, y int) (sum, prod int) {
// Separate the returned values with a regular comma
    return x + y, x * y
}
```

Data Types

```
func learnTypes () {
// Short tariffs usually perform the desired purpose
// Define a text variable using a double quotation mark
str: = "Learn Go!"

// Define a text variable using a single quotation mark
s2: = `A" raw "string literal can include line breaks.`

// Define a variable of type rune which is another name for the type
int32 and the variable of this type contains unicode
g: = 'Σ'

// Float
f: = 3.14195
```

```
// Definition of Complex Number (Complex)
c: = 3 + 4i
// Define variables using var
var u uint = 7 // natural number (positive integer)
var pi float32 = 22. / 7 // A decimal of 32 bits

// Short definition (byte is another name for uint8)
    n: = byte ('\ n')
// Arrays have a fixed and fixed size at compile time
// Defines an int array of 4 elements with an initial value of zero
    var a4 [4] int

// Define an array of 3 elements with values 3, 1 and 5
    a3: = [...] int {3, 1, 5}
```

Go offers a data type called Slices. Slices have dynamic size. Arrays and segments have advantages but use cases for segments are more common.

The following instruction defines a segment of type int

```
// Note the difference between the matrix and the chip definition, where when the chip is defined there is no number that determines its siz e
    s3: = [] int {4, 5, 9}

// Define an int type with four elements with zero values
s4: = make ([] int, 4)

// Define only, and there is no selection
var d2 [] [] float64
```

```
// Method to convert type of text to slide
bs: = [] byte ("a slice")
```

By the nature of dynamic slides, it is possible to add new elements to the slide by using the built-in append function. First pass the slide that we want to add and then the elements we want to add, see the example below.

```
    s: = [] int {1, 2, 3}
    s = append (s, 4, 5, 6)
// A slice will be printed with the following contents [1 2 3 4 5 6]
    fmt.Println (s)
```

To add a slide to another slide, we pass the two segments of the function instead of passing individual elements, and follow the second slide with three points as in the following example.

```
    s = append (s, [] int {7, 8, 9} ...)
// A slice will be printed with the following contents [1 2 3 4 5 6 7 8 9]
    fmt.Println (s)
```

The following instruction defines the p and q variables as Pointers on two int-type variables that contain two returned values from the learnMemory function:

```
    p, q: = learnMemory ()
```

When an asterisk precedes a cursor, it means the value of the variable to which the cursor refers, ie in the following example the values of the two variables returned by the learnMemory function:

```
    fmt.Println (* p, * q)
```

Maps in Go are dynamic, modifiable arrays that are similar to the type of dictionary or hashtags in other languages.

```
/ * Here we know a map whose key is text type, and the values of
numeric elements. * /
    m: = map [string] int {"three": 3, "four": 4}
    m ["one"] = 1
```

Unused variables go wrong. The underline in the following way makes you use the variable but ignores its value at the same time:

```
    _, _, _, _, _, _, _, _, _, _ = str, s2, g, f, u, pi, n, a3, s4, bs
```

This method is usually used to ignore a return value from a function. For example, you can ignore the error number returned from the os.Create file creation function, which states that the file already exists, and always assumes that the file is created:

```
    file, _: = os.Create ("output.txt" )
    fmt.Fprint (file, "By the way, this is the write function in a file")
    file.Close ()

    fmt.Println (s, c, a4, s3, d2, m)

    learnFlowControl ()
}
```

Named return values

Unlike other languages, functions can have named return values. Where the name is returned to the value of the function in the line of the definition of the function, this feature allows to return easily from any point in the function in addition to the use of the word return only without mentioning anything after:

```
func learnNamedReturns (x, y int) (z int) {
    z = x * y
// Here we just wrote the word return and implicitly means returning
the value of the variable z
    return
}
```

Note: The Go language relies heavily on garbage collection. Go has indicators but no calculations (you can mistake an empty cursor, but you can't increase the cursor).

Variables and memory

The variables p and q below are indicators of the int type and represent return values in the function. When defined, the cursors are empty; however, the use of the new built-in function makes the value of the numeric variable that p refers to zero, and therefore takes up space in memory; that is, p is no longer empty.

```
func learnMemory () (p, q * int) {
    p = new (int)
// Define a slice of 20 elements as a single unit in memory
s: = make ([] int, 20)

// Give value to an item
s [3] = 7
// Define a new local variable at the function level
r: = -2
// Returns two values from the function, which are memory
addresses for s and r variables, respectively.
return & s [3], & r
}

func expensiveComputation () float64 {
    return m.Exp (10)
}
```

Control sentences

Conditional sentences require ornate brackets and do not require parentheses.

```
func learnFlowControl () {
    if true {
        fmt.Println ("told ya" )
    }

    if false {
        // Pout.
    } else {
        // Gloat.
    }
```

We use the switch statement if we need to write more than one conditional sequence.

```
    x: = 42.0
    switch x {
    case 0:
    case 1:
    case 42:
```

```
    case 43:
    default:
    }
```

As a conditional sentence, the for clause does not take parentheses. The variables defined in the for clause are visible at the sentence level.

```
    for x: = 0; x <3; x ++ {fmt.Println ("iteration", x)
    }
```

The for statement is the only iteration statement in the Go language and has another form in the following way :

```
    for {// infinite repetition
// We can use break to stop repetition
        break
// We can use continue to go for the next iteration
        continue
    }
```

You can use range to pass over elements of an array, slide, text, map, or channel Channel range returns one value when using a channel, and two values when using a slide, matrix, text, or map.

```
// Example:
    for key, value: = range map [string] int {"one": 1, "two": 2,
"three": 3} {
// We print the value of each element in the map
        fmt.Printf ("key =% s, value =% d \ n", key, value)
    }
```

Use the underscore vs. return value if you only want the value, as follows:

```
for _, name: = range [] string {"Bob", "Bill", "Joe"} {
    fmt.Printf ("Hello,% s \ n", name)
}
```

We can use the short definition with the conditional statement so that a variable is defined and then checked in the condition statement.

Here we define a variable y, give it a value, and then place the sentence condition so that they are separated by a semicolon .

```
if y: = expensiveComputation (); y> x {
    x = y
}
```

We can define fake anonymous functions directly in the code ”

```
xBig: = func () bool {
// We defined the following variable x before the previous switch statement
    return x> 10000
}
x = 99999
// The xBig function now returns the true value
fmt.Println ("xBig:", xBig ())
x = 1.3e3
// After modifying the value of x to 1.3e3 which is equal to 1300 (that is, greater than 1000), the xBig function returns false
fmt.Println ("xBig:", xBig ())
```

In addition to the above, it is possible to define the phantom function and call it in the same line and pass it on to another function provided that it is called directly and the result type is consistent with what is expected in the function parameter.

```
    fmt.Println ("Add + double two numbers:",
        func (a, b int) int {
            return (a + b) * 2
        } (10, 2))
    goto love
love:

    learnFunctionFactory () // A function that returns a function
    learnDefer () // Snooze
    learnInterfaces () // Working with interfaces
}
```

Generate functions

We can treat functions as separate objects. For example, we can create one function and the return value is another.

```
func learnFunctionFactory () {
```

The following two methods of printing the sentence are the same, but the second method is clearer and more readable and common.

```
    fmt.Println (sentenceFactory ("summer") ("A beautiful",
"day!"))

    d: = sentenceFactory ("summer")
    fmt.Println (d ("A beautiful", "day!"))
    fmt.Println (d ("A lazy", "afternoon!"))
}
```

Decorators are found in some programming languages, and in the same concept in Go so that we can pass data to functions.

```
func sentenceFactory (mystring string) func (before, after
string) string {
    return func (before, after string) string {
        return fmt.Sprintf ("% s% s% s", before, mystring, after)
    }
}
```

Deferred execution

We can use the delay function in functions so that we perform an action before returning the return value, and if more than one is written, the execution of these actions is the opposite, as in learnDefer:

```
func learnDefer () (ok bool) {
// Deferred instructions are executed before the function returns the
result.
    defer fmt.Println ("deferred statements execute in reverse (LIFO)
order.")
    defer fmt.Println ("\ nThis line is being printed first because")
// Postponement is typically used to close a file after opening it.
    return true
}
```

Interfaces

Here we define a function called Stringer that contains one function called String; then we define a two-digit structure of int type named x and y.

```
type Stringer interface {
    String () string
}
type pair struct {
    x, y int
}
```

Here we define the String function as a pair, becoming a pair for the implementation of the Stringer interface. The variable p below is called the receiver. Notice how to access the pair structure fields by using the structure name followed by a period and then the field name.

```
func (p pair) String () string {
    return fmt.Sprintf ("(% d,% d)", p.x, p.y)
}
```

Semicolons are used to create an element of Structs. We use the short definition (using: =) in the example below to create a variable named p and specify its type in the pair structure.

```
func learnInterfaces () {
    p: = pair {3, 4}
// We call the pair's String function
fmt.Println (p.String ())
// We define a variable as i of the interface type previously defined
Stringer
var i Stringer
// This equality is correct, because pair Stringer is applied
i = p
/ * We call the String function of the variable i of type Stringer and
get the same result as before * /
fmt.Println (i.String ())

/ * When passing the preceding variables directly to the print and
output fmt functions, these functions call the String function to print
the representation of the variable. * /
// The following two lines give the same preprint result
fmt.Println (p)
fmt.Println (i)
    learnVariadicParams ("great", "learning", "here!")
}
```

Multiple inputs

It is possible to pass variable numbers of functions.

```
func learnVariadicParams (myStrings ... interface {}) {
/ * The following iteration passes on data input elements of the
function. The underline here means ignoring the cursor of the item
we're passing through. * /
    for _, param: = range myStrings {
        fmt.Println ("param:", param)
    }

/ * Here we pass the input of the variable-number function as a
parameter of another function (for Sprintln) * /
fmt.Println ("params:", fmt.Sprintln (myStrings ...))

    learnErrorHandling ()
}
```

Errors Handling

The keyword “ok,” is used to determine whether a statement is correct. If an error occurs, we can use err to find out more details about the error.

```
func learnErrorHandling () {
    m: = map [int] string {3: "three", 4: "four"}
    if x, ok: = m [1]; ! ok {
// ok Here it will be false because number 1 is not on the map m
        fmt.Println ("no one there")
    } else {
// x will be the value in the map
        fmt.Print (x)
    }
/ * Here we try to convert a text value to a number which will result in
an error, and we print out the error details if err is not nil * /
    if _, err: = strconv.Atoi ("non-int"); err! = nil {
        fmt.Println (err)
    }
    learnConcurrency ()
}
```

```
// given here is a channel type, which is an object to secure
concurrent connections

func inc (i int, c chan int) {

// When a channel type element appears on the north, the <- means
transmit

    c <- i + 1

}
```

Concurrent execution

We use the preceding function to make a numerical addition to some numbers in conjunction. We use make as we did at the beginning of the article to create a variable without specifying a value for it.

```
func learnConcurrency () {

// Here we create a channel-type variable named c

    c: = make (chan int)

/ * We start by creating three concurrent Go functions. The numbers
will be incremented simultaneously (in parallel if the device is
configured to do so). * /

// All transmissions will go to the same channel

// Go here means starting a new function

    go inc (0, c)

    go inc (10, c)

    go inc (-805, c)

// Then we make three readings from the same channel and print the
results.

/ * Note that there is no order of read access from the channel, and
also note that when the channel appears to the right of the operation
<- it means that we are reading and receiving from the channel. * /
```

```
  fmt.Println (<- c, <-c, <-c)

// New channel with text
cs: = make (chan string)
// Channel contains text channels
ccs: = make (chan chan string)

// Send a value of 84 to channel c
go func () {c <- 84} ()
// Send wordy to channel cs
go func () {cs <- "wordy"} ()

/ * The Select statement is similar to the switch statement, but in
each case it contains a process for a channel that is ready to
communicate with. * /
select {
// The value received from the channel can be saved in a variable.
    case i: = <-c:
        fmt.Printf ("it's a% T", i)
    case <-cs:
        fmt.Println ("it's a string")
// Empty channel but ready to communicate
case <-ccs:
        fmt.Println ("didn't happen.")
    }
// Web programming
    learnWebProgramming ()
}
```

Web

We can start a web server using one function from the http package. In the first parameter of the ListenAndServe function, we pass the TCP address to listen, and the second parameter is an http handler.

```
func learnWebProgramming () {
    go func () {
        err: = http.ListenAndServe (": 8080", pair {})
// We print errors if they exist.
fmt.Println (err)
    } ()

    requestServer ()
}
// Make the pair a http handler by applying its only function
called ServeHTTP
func (p pair) ServeHTTP (w http.ResponseWriter, r *
http.Request) {
// Follows the Write function of the http.ResponseWriter
package and we use it to return a reply to the http request
w.Write ([] byte ("You learned Go in Y minutes!"))
}

func requestServer () {
    resp, err: = http.Get ("http: // localhost: 8080")
    fmt.Println (err)
    defer resp.Body.Close ()
    body, err: = ioutil.ReadAll (resp.Body )
    fmt.Printf ("\ nWebserver said:`% s` ", string (body))
```

Chapter VI

Create Backend

Go programming development began with an experiment from Google engineers to avoid some of the complexities of other programming languages while taking advantage of their strengths. Go is constantly evolving with the participation of an increasingly open source community.

The Go programming language is intended to be easy, but Go code writing conventions can be difficult to understand. In this lesson, I will show you how to start all my software projects when I use Go, and how to use the expressions provided by this language. We will create a backend service for the web application.

Setting up the work environment

The first step is, of course, installing Go. Go can be installed from official repositories on Linux distributions; for example for Ubuntu:

```
sudo apt install golang-go
```

Go versions in official repositories are usually slightly older than those on the official website, but they do work; You can install newer versions on Ubuntu (here) and Centos (here).

Mac OS users can install the language via Homebrew:

```
brew install go
```

The official site also contains language installation executables on most operating systems, including Windows.

Make sure Go is installed by executing the following command:

```
go version
```

Example of the result of the above command (on Ubuntu distribution):

```
go version go1.6.2 linux / amd64
```

- Links to install on Windows and set up paths -

There are plenty of text editors and plugins available for writing Go codes. I personally prefer Sublime Text and GoSublime, but Go's writing method makes it easy to use regular text editors especially for small projects. I work with professionals who spend the whole

}

day programming in the Go language using the Vim text editor, without any addition to highlighting the syntax highlighting. Surely you will only need a simple text editor to start learning Go.

new project

If you have not created a working folder during the installation and setup of Go, the time is right.

Go tools expect all the code to be on the $ GOPATH / src path, so our work will always be in this folder. The Go toolkit can also address hosted projects on sites such as GitHub and Bitbucket if they are set up.

For the purposes of this lesson, we will create a new empty repository on GitHub and name it “hello” (or any name that suits you). We create a folder within the GOPATH folder to receive repository files (replace your-username with your GitHub username):

```
mkdir -p $ GOPATH / src / github.com / your-username
cd $ GOPATH / src / github.com / your-username
```

We copy the repository within the folder we created above:

```
git clone git@github.com: your-username / hello
cd hello
```

We will now create a file called main.go to contain a short program in Go:

```
package main

func main () {
    println ("hello!")
}
```

Run the go build command to drain all the contents of the current folder. An executable file with the same name will be produced; you can then request that it be run by mentioning its name as follows:

```
go build
./hello
```

The result:

```
hello!
```

Despite years of development in Go, I still start my projects in the same way: an empty Git repository, main.go file and a few commands.

Any application that follows the usual methods of organizing the Go code becomes easily installable with the go get command. For example, if you deposited the file above and pushed it into the Git repository, anyone with a Go work environment can perform the following two steps to run the program:

```
go get github.com/your-username/hello
$ GOPATH / bin / hello
```

Create a Web server

Let's make our previous simple program a web server:

```
package main

import "net / http"

func main () {
    http.HandleFunc ("/", hello)
    http.ListenAndServe (": 8080", nil)
}

func hello (w http.ResponseWriter, r * http.Request) {
    w.Write ([] byte ("hello!"))
}
```

There are a few lines that need to be explained.

First we need to import the net / http package from the Go library:

```
import "net / http"
```

Then, install the Handler function in the root path of the Web server. Http.HandleFunc handles the initial Http request prompt in Go, ServeMux.

```
http.HandleFunc ("/", hello)
```

The hello function is of type http.HandlerFunc that allows the use of regular functions as processing functions for HTTP requests. The http.HandlerFunc functions have a special Signature signature (the function signature is the passed data and the data types returned by this function) and can be passed in a parameter to the HandleFunc function that registers the passed function in the given parameter of the ServeMux router, thus creating a web server, each time it arrives A new request that matches the root path, creates a new copy of the hello function.

The hello function receives a variable of the http.ResponseWriter type that the wizard uses to generate an HTTP response and thus generates a response to the client request via the Write function provided by http.ResponseWriter. Since http.ResponseWriter.Write takes a generic type of [] byte or byte-slice, we convert the hello string to the appropriate type:

```
func hello (w http.ResponseWriter, r * http.Request) {
    w.Write ([] byte ("hello!"))
}
```

Finally, we run a Web server on port 8080 via the http.ListenAndServe function that receives two parameters, the first is the Port port and the second is a processing function. If the value of the second parameter is nil, then we want to use the default router DefaultServeMux. This Synchronous call, or Blocking, keeps the program running until the call is interrupted.

Run and run the program the same way as before:

```
go build
./hello
```

Open another command-line-terminal and send an HTTP request to port 8080:

```
curl http: // localhost: 8080
```

The result:

```
hello!
```

It's simple. There is no need to install an external framework, download Dependencies or create project structures. The executable itself is an authentic Native code without operational credentials. Furthermore, the Web server's standard library is geared to the production environment with defenses against common cyberattacks. This code can answer requests over the network directly and without arguments.

Add new tracks

We can do more important things than just say hello. Let the entry be a city name that we use to call the Weather API and redirect the answer - temperature - in response to the request. OpenWeatherMap provides a free and easy-to-use software interface for climate forecasting. Register at the site for an API key. Can query from OpenWeatherMap by cities. The API returns an answer as follows (we slightly edited the result):

```
{
    "name": "Tokyo",
    "coord": {
        "lon": 139.69,
        "lat": 35.69
    } ,
    "weather": [
        {
            "id": 803,
            "main": "Clouds",
            "description": "broken clouds",
            "icon": "04n"
        }
    ],
    "main": {
        "temp": 296.69,
        "pressure": 1014,
        "humidity": 83,
        "temp_min": 295.37,
        "temp_max": 298.15
    }
}
```

The variables in Go have static type, meaning that the type of data stored by the variables should be declared before use. So we will have to create a data structure to match the interface response format. We don't need to store all the information, just keep the data we care about. We are now content with the name of the city and the expected temperature that comes with the Kelvin unit. We will define a structure to represent the data we need from the climate forecast service.

```
type weatherData struct {
    Name string `json:" name "`
    Main struct {
        Kelvin float64 `json:" temp " `
    } `json:" main "`
}
```

The type keyword defines a new data structure called weatherData and declares it to be struct. Each field in variables of the struct type contains a name (for example, Name or Main), a data type (string or another anonymous), and a tag. The tags in Go are similar to the Metadata metadata, and enable us to use the encoding / json package to re-align the OpenWeatherMap response and save it in our data structure. More code is required than in languages with dynamic data types (i.e., a variable can be used as soon as we need it without having to declare the data type) like Ruby and Python, but it gives us the security of the data type.

We have defined a data structure. We now need a way to fill this structure with data coming from the API; we will write a function for this.

```
func query (city string) (weatherData, error) {
    resp, err: = http.Get
("http://api.openweathermap.org/data/2.5/weather?
APPID=YOUR_API_KEY&q=" + city)
    if err! = nil {
        return weatherData {}, err
    }

    defer resp.Body.Close ()

    var d weatherData

    if err: = json.NewDecoder (resp.Body) .Decode (& d); err! = nil {
        return weatherData {}, err
    }

    return d, nil
}
```

The function takes a string of characters representing the city and returns a variable of weatherData and error data structures. This is the basic way to deal with errors in Go. Functions encapsulate certain behavior, and this behavior can fail. For our example, the GET request we send to OpenWeatherMap can fail for several reasons, and the returned data may be the one we are waiting for. In both cases, we return a non-nil error to the customer who is expected to handle the error in the context of the request.

If the http.Get request succeeds, we postpone a request to close the response board to execute after exiting the Scope function (i.e. after returning from the HTTP request function), an elegant way to manage resources. In the meantime we reserve the weatherData structure and use json.Decoder to read the answer data and enter it directly into our structure.

When the rework of the answer data is successful, we return the weatherData variable to the caller with an empty error to indicate that the operation was successful. We now proceed to associate that function with the processing function of the request:

```
http.HandleFunc ("/ weather /", func (w http.ResponseWriter, r *
http.Request) {
    city: = strings.SplitN (r.URL.Path, "/", 3) [2]

    data, err: = query (city)
    if err! = nil {
        http.Error (w, err.Error (), http.StatusInternalServerError )
        return
    }

    w.Header (). Set ("Content-Type", "application / json; charset =
utf-8")
    json.NewEncoder (w) .Encode (data)
})
```

We define a processing function on the In-line line instead of defining it separately. We use the strings.SplitN function to take all that exists after / weather / in the path and treat it as a city name. We execute the request and if we encounter errors that the customer knows about using the http.Error utility, we stop the execution of the function to indicate that the HTTP request is complete. If there is no error, we tell the customer that we are sending JSON data to it and use the json.NewEncode function to directly encode the weatherData content in JSON format.

The blade is so elegant, it adopts a procedureural and easy to understand. They cannot be interpreted and cannot bypass common mistakes. If we move the "hello, world" processing function to the / hello path and import the required packages, we will get the following complete program:

```
package main

import (
    "encoding / json"
    "net / http"
    "strings"
)

func main () {
    http.HandleFunc ("/ hello", hello)

    http.HandleFunc ("/ weather /", func (w http.ResponseWriter, r *
http.Request) {
        city: = strings.SplitN (r.URL.Path, "/", 3) [2]

        data, err: = query (city)
        if err! = nil {
            http.Error (w, err.Error (), http.StatusInternalServerError)
            return
        }

        w.Header (). Set ("Content-Type", "application / json; charset =
utf-8")
        json.NewEncoder (w) .Encode (data)
    })
    http.ListenAndServe (": 8080", nil)
}

func hello (w http.ResponseWriter, r * http.Request) {
    w.Write ([] byte ("hello!"))
}

func query (city string) (weatherData, error) {
    resp, err: = http.Get
("http://api.openweathermap.org/data/2.5/weather?
APPID=YOUR_API_KEY&q=" + city )
    if err! = nil {
        return weatherData {}, err
    }

    defer resp.Body.Close ()

    var d weatherData

    if err: = json.NewDecoder (resp.Body) .Decode (& d); err! = nil {
        return weatherData {}, err
    }

    return d, nil
}

type weatherData struct {
    Name string `json:" name "`
    Main struct {
        Kelvin float64 `json:" temp "`
    } `json:" main "`
}
```

We run and execute the program in the same way as explained above:

```
go build
./hello
```

We open another terminal and request the path http: // localhost: 8080 / weather / tokyo (Kelvin thermometer):

```
curl http: // localhost: 8080 / weather / tokyo
```

The result:

```
{"name": "Tokyo", "main": {"temp": 295.9}}
```

Query from multiple software interfaces

Perhaps more accurate temperatures can be obtained if we inquire from several weather services and only average them. Most weather service interfaces require registration. We will add Weather Underground, so sign up for this service and find the authentication keys necessary to use the API.

Since we want to have the same behavior from all services, it is worthwhile to write this behavior in an interface.

```
type weatherProvider interface {
    temperature (city string) (float64, error) // in Kelvin, naturally
}
```

We can now convert the query function from the previous openWeatherMap to a data type that corresponds to the weatherProvider interface. Since we do not need to save any status to make an HTTP GET request, we will use an empty struct structure, and we will add a short line in the new query function to record what happens when you connect to the services for later review:

```
type openWeatherMap struct {}

func (w openWeatherMap) temperature (city string) (float64, error) {
    resp, err: = http.Get
("http://api.openweathermap.org/data/2.5/weather?
APPID=YOUR_API_KEY&q=" + city)
    if err! = nil {
        return 0, err
    }

    defer resp.Body.Close ()

    var d struct {
        Main struct {
            Kelvin float64 `json:" temp "`
        } `json:" main "`
    }

    if err: = json.NewDecoder (resp.Body) .Decode (& d); err! = nil {
        return 0, err
    }

    log.Printf ("openWeatherMap:% s:% .2f", city, d.Main.Kelvin)
    return d.Main.Kelvin, nil
}
```

We just want to extract the temperature (Kelvin) from the answer, so we can define the struct structure on the Inline line. Otherwise, the code is similar to the previous query function, but it is defined as the Method method of the openWeatherMap structure. This method allows us to use the Instance sample from openWeatherMap in place of the weatherProvider interface.

We will do the same for Weather Underground. The only difference with the previous service is that we will store the API key in a struct and then use it in the function.

Note that Weather Underground does not process identical city names as well as Open WeatherMap, which should be noted in actual applications. We will not address this in our simple example.

```
type weatherUnderground struct {
    apiKey string
}

func (w weatherUnderground) temperature (city string) (float64,
error) {
    resp, err: = http.Get ("http://api.wunderground.com/api/" +
w.apiKey + "/ conditions / q /" + city + ".json")
    if err! = nil {
        return 0, err
    }

    defer resp.Body.Close ()

    var d struct {
        Observation struct {
            Celsius float64 `json:" temp_c "`
        } `json:" current_observation " `
    }

    if err: = json.NewDecoder (resp.Body) .Decode (& d); err! = nil {
        return 0, err
    }

    kelvin: = d.Observation.Celsius + 273.15
    log.Printf ("weatherUnderground:% s:% .2f", city, kelvin)
    return kelvin, nil
}
```

We now have a weather service provider. Let us write a function that queries the two and returns the average temperature. For the sake of simplicity, we will stop querying if we have trouble getting data from both services.

```
func temperature (city string, providers ... weatherProvider) (float64,
error) {
    sum: = 0.0

    for _, provider: = range providers {
        k, err: = provider.temperature (city)
        if err! = nil {
            return 0, err
        }

        sum + = k
    }

    return sum / float64 (len (providers)), nil
}
```

Note that the function definition is very close to the definition of the temperature function defined in the weatherProvider interface. If we combine the weatherProvider interfaces into a data type and then define a function called temperature on this type, we can create a new type that combines the weatherProvider interfaces.

```
type multiWeatherProvider [] weatherProvider

func (w multiWeatherProvider) temperature (city string) (float64,
error) {
    sum: = 0.0

    for _, provider: = range w {
        k, err: = provider.temperature (city)
        if err! = nil {
            return 0, err
        }

        sum + = k
    }

    return sum / float64 (len (w)), nil
}
```

Fabulous! We will be able to pass multiWeatherProvider to any function that accepts weatherProvider.

We now associate the HTTP server with the temperature function to obtain temperatures when a path with a city name is requested:

```
func main () {
    mw: = multiWeatherProvider {
        openWeatherMap {},
        weatherUnderground {apiKey: "your-key-here"},
    }

    http.HandleFunc ("/ weather /", func (w http.ResponseWriter, r *
http.Request) {
        begin: = time.Now ()
        city: = strings.SplitN (r.URL.Path, "/", 3) [2]

        temp, err: = mw.temperature (city)
        if err! = nil {
            http.Error (w, err.Error (), http.StatusInternalServerError)
            return
        }

        w.Header (). Set ("Content-Type", "application / json; charset =
utf-8")
        json.NewEncoder (w) .Encode (map [string] interface {} {
            "city": city,
            "temp": temp,
            "took": time.Since (begin) .String (),
        })
    } )
    http.ListenAndServe (": 8080", nil)
}
```

Run the program, run it and request a web server link as we did before. In addition to the JSON format in the application window, you will find outputs from the server registrations we added above in the window from which you ran the program.

```
./hello
2015/01/01 13:14:15 openWeatherMap: tokyo: 295.46
2015/01/01 13:14:16 weatherUnderground: tokyo: 273.15

$ curl http: // localhost: 8080 / weather / tokyo
{"city": "tokyo", "temp": 284.30499999999995, "took": "821.665230ms"}
```

Make the queries run in parallel

We limit ourselves to the hour by querying the interfaces consecutively, one by one. There is nothing to stop us from querying both interfaces at the same time, which will reduce their response time.

We take advantage of Go's simultaneous playback capabilities across Go sub-units, goroutines and Channels. We will place each query in its own subunit and run it in parallel. We then combine the answers into one channel and then calculate averages when all queries are complete.

```
func (w multiWeatherProvider) temperature (city string) (float64, error) {
    // We create two channels, one for temperature and the other for errors
    // Each service provider adds a value to only one channel
    temps: = make (chan float64, len (w))
    errs: = make (chan error, len (w))

    // For each ISP, we call an anonymous function. An unknown
function calls the dependent name temperature and then redirects
the result obtained.
    for _, provider: = range w {
        go func (p weatherProvider) {
            k, err: = p.temperature (city)
            if err! = nil {
                errs <- err
                return
            }
            temps <- k
        } (provider)
    }

    sum: = 0.0

    // We collect temperatures - or errors, if any - from each service
    for i: = 0; i <len (w); i ++ {
        select {
        case temp: = <-temps:
            sum + = temp
        err: case = <-errs:
            return 0, er r
        }
    }

    // Return the heat as before
    return sum / float64 (len (w)), nil
}
```

The time required to execute all queries now equals the time required to get an answer from the slowest weather service; rather than the sum of the durations of all queries as before. All we needed to do was modify the behavior of multiWeatherProvider, which still satisfies the needs of the simple and unparallel weatherProvider interface.