Go / Golang

- Packages vs Modules
- Import packages / modules
- Structs exported fields
- Go with Examples
    - Introduction: Hello World, values, and variables
    - If/else
    - Switch Statement
    - Arry and Slices
    - Loops and range
    - Functions
- Embed Modules
- make vs new

Packages vs Modules

Preliminary

Forget about the packages and modules that you know from Python, it is no related, yes it is about organizing files but not in the same way.

Packages

A package is made up of Go files that lives under the same directory. You can think about the directory as the package name.

Every Go file must be under a package with the first line being

```
package <package name>
```

If the Go file is under the main package (i.e. just under src directory) then the package name will be **main**.

Same Go files that are under the same directory will have the same package name, so they will all have the same `package` line.

Modules

Modules are like npm in Node.js used for dependency management. You can build your own module and then share it with others much like packages with npm.

In essence a module is just a collection of related packages. You would include a `go.mod` and `go.sum`. The `go.mod` file gives the module a name and name it's dependency that is required for this module.

You would create a module with a new Go project by running `go mod init <module_path>`.

`module_path` is compose of the repository URL + the module name. It tells Go where it will be able to find the module. i.e. your `module_path` can be `github.com/tamaarine/testgoproject`. Where Go will know to find the module it will have to github and under that link to find the module dependency.

Import packages / modules

I have a helper file that I would like to use within the same project.

If say your directory layout is like below:

```
src
├main.go
    helper.go
```

A very simple directory, you wrote some helper function in `helper.go` and you would like to use it in `main.go`. Now because they are under the same *package* you do not need do anything importing to use the functions / global variables that's defined in `helper.go` in `main.go`, you can just call it as if they are already in the same file.

> However, when you compile you would need to provide either `build / run` with both of the files that you use.
> If you just run `go build main.go` or `go run main.go` it will not work. You must run `go build main.go helper.go` or `go run main.go helper.go`.
> To link all the files you need to explicitly mention it.

I have a package that I would like to use within the same project

So instead of putting the helper file in the same package you decided to be a little fancy and include it in another directory (package) on it's own like below:

```
src
├main.go
    - helper
   ├├helper.go
```

Now how can I use it in `main.go`? Since it is in another package now you would need to explicitly import it in `main.go` in order to use it. The way that you would import it is by importing it using the

`module_path` name.

For example, if this local project's `module_path` is github.com/tamaarine/testproject, or whatever, it doesn't matter if it is published under that URL, you would import the `helper` package by writing

```
import (
	"github.com/tamaarine/testgoproject/helper"
)
```

Then you can access the functions or variables that's defined in that package by using `helper.<function/variable name>`.

> Do keep in mind that those functions or variables that you decide to export through this way need to be explicitly exported, by making their name capitalized. So instead of function name add, it needs to be Add. To tell Go that this function is going to be exported and be used.

> Doing it this way you do not need to add any files to the `build / run` CLI. You can just do `go run main.go` and it will compile without any complains.

I made a module that I haven't published yet to a repository, but I want to use it in another local project, how do I do it?

Okay cool, say you got yourself a module under the `module_path=github.com/tamaarine/testgoproject` but you haven't published it into any repository but your local project wants to use it because you want your stuff to be modularized. How do you do it?

Well you would use the replace directive within your `go.mod` file.

```
module local/project2

go 1.20

replace github.com/tamaarine/testgoproject => /home/tamarine/GoProject/Project1
```

In this case, say we are making a new project under the `module_path=local/project2` and we want the module that haven't been published to a repository yet. By using the replace directive you can

tell Go to find the module in another place, in this case in the local file directory.

Left side consist of the module that you would like to replace, and the right side is the local directory to the module root directory where it contains the `go.mod` file.

Using replace directive basically substitute the module path with another. This can be useful if you're developing a new module but you haven't published the code to a repository yet but just want to test it locally. Or you found an issue with the dependency, so you cloned the repo and fixed it and wanted to test it without pushing the change yet.

Structs exported fields

Exported structs and fields

Like functions if you would like to export a struct from a package for another package or module to use then the first letter of the struct name must be capitalized, otherwise it is not exported and cannot be used.

```
type ComplexNum struct {
	Real int
	Complex int
}
```

When you are initializing a struct you can use named fields to assign value to the fields such as:

```
c1 := structs.ComplexNum{Real: 10, Complex: 20}
```

Keep in mind that the field must also be exported in order for key-value assignment, otherwise, you cannot assign to it and will receive the default value.

Special case: Struct are in the same go file

If the struct definition is within the same go file then whether or not the struct / field itself is exported it doesn't matter, you can use it within the same go file. Assign the key-value pair regardless of whether the field is actually exported.

```
type complexNum struct {
	real int
	complex int
}

func main() {
	c1 := complexNum{real: 10, complex: 30}
	fmt.Println(c1)
}
```

Special case: Struct are in the same package

Very similar to the previous case, if the struct are defined within the same package, you can just use it whether or not the struct / field itself is exported. You can also assigned to any unexported field, it doesn't not matter.

Go with Examples

Introduction: Hello World, values, and variables

Go code layouts

A Go project is also called a module. A module is just a collection of packages.

A package is just a group of related .go files. You would declare the .go files that belong in the same package with the line

```
package <package name>
```

For example: If you use the `main` package it is used to make the package an executable program (you get a binary) because it contains your main function. The `main` package tells the Go compiler that the package will be compiled as an executable program rather than a library which will not produce an executable.

Otherwise, the package name can be whatever you want. However, keep the package name that you are declaring the same as the directory that it is under. For example:

```
src
helper
    - helper.go
    main.go
```

If you have a directory like such keep the package name that you use in `helper.go` as `helper` because if do `package lol` which doesn't match the directory name. You would be importing the helper package in `main.go` as

```
import (
"module/path/helper
)
```

But when you want to call the function from the `lol` package it would be

```
lol.helperFunc()
```

So keeping the directory name and the package name the same would make it easier for yourself and for others to maintain.

Hello World

```
package main

import "fmt"

func main() {

    var a = "initial"
    fmt.Println(a)

    var b, c int = 1, 2
    fmt.Println(b, c)

    var d = true
    fmt.Println(d)

    var e int
    fmt.Println(e)

    f := "apple"
    fmt.Println(f)
}
```

As you can see the `fmt` module that is imported is the built-in module in Golang for printing things out to the consoles.

`Println` is just one of the functions inside `fmt` module to print things, there are many others.

> Notice that the function is capitalized. This is not arbitary but rather telling Go that the Println function from the `fmt` module is exported for others to use. If it is lower-case then it cannot be used by others.

Variables

Couple of ways of declaring a variable:

1. This is the most verbose way of declaring and assigning it, you have the type of the variable on the right and then assignment to the variable of the expected type to the right of the assignment operator

```
var x int = 10
```

2. A simpler variable declaration and assignment of the previous method, you can skip out on the type if the type on the right is what you want of the assignment. This is possible because the compiler can infer the type in compile time

```
var x = 10
```

3. If you just want to declare the variable without assigning it here it is. The variable will have it's zero value assigned.

```
var x int
```

4. You can also declare multiple var in a block. Also provide them with initialization as well

```
var (
	x int
	y int
	z string = "default"
)
```

5. Go also have a short hand declaration that you can only use within a function. This method cannot be used outside of the function, but `var` can be. You can just skip out the `var` keyword and add in a colon. The compiler will use type inference to infer the type on the right.

```
x := 10
```

6. There is also constants in Go but they are pretty limited. You can only assign them the values that compiler can figure out at compile time. Type with constant is allowed as well.

```
const y = "hello world"
```

var vs :=

There are limitation on `:=`. Such as you cannot use it outside of functions to declare a global variable like you can with `var`.

You cannot have a zero value if you are using `:=`. You must assign something to it in order to make it work.

Zero value

Type	**Zero value**
Boolean	false
Numeric types	0
Float	0
String	"" (Empty string)
Structs	Each of struct's key is set to their respective zero-value

If/else

If/else

If else is very similar to C except you just take out the parenthesis.

```
func main() {
	x := 10

	if x > 10 {
		fmt.Println("It is greater than 10")
	} else if x < 10 {
		fmt.Println("It is less than 10")
	} else {
		fmt.Println("It is equal to 10")
	}
}
```

Notice that the `else` and `else if` MUST be after the closing bracket of the previous statement. Otherwise, it will not compile.

Temporary variable

If/else generate it's own block, with this you can do something very interesting. You can declare and assign a variable that is only valid within that particular if/else block like such:

```
func main() {
	if x := 10; x > 10 {
		fmt.Println("It is greater than 10")
	} else if x < 10 {
		fmt.Println("It is less than 10")
	} else {
		fmt.Println("It is equal to 10")
	}
}
```

Ternary operator?

No ternary operator, must use a full if/else branch for assignment of the variable.

Switch Statement

Switch Statement

Switch in Go is pretty nice, it function similar to the switch statement in C but better.

```
x := 20

switch x {
  case 10, 20:
    fmt.Println("It is either 10 or 20")
  default:
    fmt.Println("It is not 10 and is not 20")
}
```

You can put case on multiple values rather than only just one. In addition, the default case is not mandatory, you can leave it out of your program.

Switch on conditions

Furthermore, instead of switching on a value, in the previous example it was switching on the variable `x` however, if you leave the value to switch on out, you can switch on other variables conditionally.

```
x := 22

switch {
  case x > 10:
      fmt.Println("It is greater than 10")
  case x < 10:
      fmt.Println("It is less than 10")
  case x == 10:
      fmt.Println("It is equal to 10!")
}
```

Arry and Slices

Array

An array can be declared with the following syntax:

```
var <arr_name> [length]type
```

For example if I want to create a list of array with length of 5:

```
var a [5]int
```

The array is initialized with zero value of the type. So since it is integer it will be initialized to be 0. If it were `bool` it will all be `false`.

Indexing

Indexing the array is just as the same as with other array

```
arr_name[index]
```

There are no negative indexing like Python to prevent unintentional bugs.

Slices

Array is fine on it's own, but it comes with a down side, in the sense that it is very rigid. If you want to pass an array to a function, the size must be specified, otherwise, it cannot be passed.

Therefore, slices which are references to array are created to combat that issue. Passing slices of array allows the size to vary depending on the array.

The function below can only take in an array of 3 elements, if you want this function to work with any array, then you will have to make it to take in a slice.

```
func printArray(nums [3]int) {
	i := 0
	for i < len(nums) {
```

```
    fmt.Printf("The number is %d\n", nums[i])
    i += 1
  }
}
```

This variation which just took out the number becomes a slice, which will allow it to take in any slice of varying sizes.

```
func printArray(nums []int) {
  i := 0
  for i < len(nums) {
    fmt.Printf("The number is %d\n", nums[i])
    i += 1
  }
}
```

Loops and range

For loops

The only looping construct in Go is the for loop. There is no while loop, but it is actually just merged into the for loop.

Loop on condition

Basically while loop except you replace the while with for

```
i := 1

for i <= 3 {
  fmt.Println(i)
  i = i + 1
}
```

Normal for loops

The same for loop that you see in C, Java, or JavaScript without the parenthesis.

```
for j := 0; j < 5; j ++ {
  fmt.Prinltn(j)
}
```

You can break on condition as well.

Range

The range keyword is used to iterate through an iterable object such as an Array or Map.

```
nums := []int{1, 2, 3, 4}
for i, num := range nums {
  fmt.Println(i)
  fmt.Println(num)
```

```
}
```

If you don't want the index then you can replace it with `_` to ignore that variable.

The `range` keyword can also be applied on Maps to iterate through the key-value pairs.

Functions

Functions

To write a function you would need to use the `func` keyword, provide the name of the funciton, provide in the argument of the function, and finally the return type of the function:

```
func plus(a int, b int) int {
  return a + b
}
```

Multiple return values

You can return multiple values in a function, however, you would need to explicit state what those return values are:

```
func vals() (int, int) [
  return 10, 20
}
```

In this case, the function `vals` will return two return values which are just simply two integers. Notice that the return types are surrounded by parenthesis, this is needed, if you only have one return value then you skip it.

Functions that takes in variable number of arguments

In Python, Java, JavaScript there is the concept of making function accept in a variable number of arguments, in Go there is that concept too.

And surprisingly it is also done with the same operator `...` in most languages.

```
func sum(nums ...int) int {
  total := 0
  for _, num := range nums {
    total += num
  }
  return total
```

```
}
```

The type of `nums` will be `[]int`.

Embed Modules

Embed Module

This module is a super duper cool module in the sense that you are able to pack static files into your binary, rather than having your program relying on the actual file being present in your filesystem when it is ran.

Normally when you do deployment with a Go server, you would have the server binary, the static frontend files that is open and serve by the Go server. You can pack everything into a Docker image so the server that's running your web application can just spin up the container which will have everything. However, in Golang 1.16 you can directly embed your static frontend files into the binary so you do not need a docker image to pack your static files in. You can just embed it directly into your binary.

Embed directives

To get started with using the Embed module you would actually use the embed directives which instructs compiler at compile time what to do.

For example:

```
//go:embed $PATH
var content $WANTED_DATA_TYPE
```

1. `$PATH` is the file or directory that you want to include. If you use dataa type `embed.FS` then you can put multiple files or multiple directories
2. `$WANTED_DATA_TYPE` needs to be replace with one of the followings
    1. `string` = Accepts a single file and when the binary is built, it wil lread the content of that entire file into the variable as a string. If you use this you can only embed in ONE file. No multiple files or a directory
    2. `[]byte` = Same as String, but it will be read in as `[]byte`
    3. `embed.FS` = Using this data type you are allowed to embed multiple directories and even files. This struct implements the `io/FS` interface, which means that `net/http` package can use it as part of the handler

`embed.FS` after you embed the files into the binary, you can open files that are included as part of the binary. Even if the files / directory doesn't exist after the program is ran. Magic. Power of embedded files.

Note about `embed.FS`

Do keep in mind that the directory that you embed, the variable that it is assigned under refer to the root directory that contains the directory or file. For example:

```
//go:embed static/
var embeddedFS embed.FS
```

The variable `embeddedFS` refers to a "root directory" that contains the "static" directory (I put it in quotes because it is in the binary not in the actual OS filesystem

Thus if you want to say open a file you need to do:

```
embeddedFS.Open("static/index.html")
```

And not this:

```
embeddedFS.Open("index.html")
```

In terms of `net/http` package, that means the files will be serve the subdirectory that you embed as. For example, if you embed `static/index.html`, and you add a handler `"/"` that references the particular `static/index.html` file, you will have to visit `localhost:80/static/index.html` or `localhost:80/static` in order to see the file rather than just `localhost:80`.

To solve that problem you would need use `fs.Sub` which:

```
Sub returns an FS corresponding to the subtree rooted at fsys's dir.
```

Basically returns you a filesystem that is rooted at the provided filesystem, making an "alias" per se.

make vs new