Golang Course

Introduction

Compiled.

Go is a compiled language. This implies that if we want to execute our program we need compile before running. Different to compiled languages, it exists interpreted languages. This kind of languages uses an interpreter that takes the code lines, translate they and the generate instructions to the machine in his native language. As general rule, compiled languages are faster than interpreted languages due that last them requires to make translation operation by each execution. In the other side, translation is done one time only, during compilation stage.

Typing.

Golang are strongly typed, this has some advantages and constrains. First, strong typing reduce the amount of errors that could be raise in production stages, on the other side, it could be tedious to develop with static types.

memory.

In the general view, we could have two memory management situations. First, languages that makes manual handling of memory, this evolves that each developer is responsible of handling the memory used and free for his program. On the other hand, it exist programming langs that uses components to make the memory management. Principal example of this is JAVA with garbage collector. Golang has a similar structure like JAVA, however, it uses runtime code as garbage colector and also it is not dependant of a JVM.

go runtime.

Is a piece of code that is included in each golang program. Is the base any golang program

Types

Basic Types

Basic data types in golang are bool, string int, int8, int16, int32, int64, unit, uint8, uint16, uint32, uintptr, byte, rune, float32, float64, complex64 and complex128

int and number represents a signed number that could be represented though n bits. uint is the same, but it is only allow to represent positive numbers. float numbers are used to integers with decimal part numbers, they are also limited ny the amount of bytes.

byte numbers are used to represent binary data, however it is an alias for uint8 number that is constrained to take only 0 and 1 values. rune number refers to a unicode character, that's the same that a single character in a string.

complex data type it is used to represent complex numbers, they are numbers with two components, real and imaginary.

Declaring Variables

to declare a variable we use the reserved word “var”, after that we also specify the name and the data type of the new variable, something like this

```
var number int
```

As you can see in the previous example, this variable does not have a value assignation, by this way, data will take a default value according to the data type, for example, string will be assigned to “” value, integers will be assigned with 0 value, booleans will be assigned with false value, etc. In the other hand, you can assign a value during variable declaration, only need to assign a value using assignation operand =, some like this:

```
var pi float64 = 3.14159
```

Previous declaration way is called long variable declaration, however you can do this in a small way. To do this we use the short declaration variable, this is realized by mean of direct assignation of a value to a variable without data type declaration. Short declaration variable infers the data type based on the value of the variable, to understand this part, see the next code snippet.

```
empty := ""
```

In the previous example, Golang will infer the data type according to value, for this case will infer a string data type for the value empty. Then, the previous example is equivalent to:

```
var empty string = ""
```

Like other strongly typed programming languages, go permits value assignation after variable declaration. Then, if you declare a variable with type but it is not initialized with value, this could take value after. Some like this:

```
var num_00 int
num_00 := 758
```

Variable Cast

Like any language, variables could be cast, however, in Golang cast is called conversion. Then, we can convert some types to another one. Usually, convert is done using a call of the to cast variable as an argument of the datatype written as a function, as is shown as follows.

```
temperatureInt := 88
temperatureFloat := float64(temperatureInt)
```

Printing Variables

In Golang exists fmt package whose permits, between another features, interacts with strings. Most known method is Printf, this function prints a string in a formatted way in the standard out. In the other hand, Sprintf returns a formatted string without generate any kind of output and this return could be stored in a variable, taking account that you should to respect the variable type to avoid compiolation errors due assignation of a wrong type.

```
fmt.Printf("I am %v years old", 10)
```

In the previous code snippet, we're trying to show the age of a person in the standard output. So, we put a sort of template that is filled with the desired value. In this case, we want to put the number 10 in string format. To do this, we use the expression %v, which is used to put the value of a variable inside of a string that will be printed using Printf. This expressions are called formatting verbs, and they are used to call some class of format from a variable that follows the expression that it is in the first position of parameters of this function. If we grant that the data that will be printed is an string, we could use %s verb formater. Next we can see a list of formatters.

Formatter verb	Use Case
%v	generic formatter. It is recommended when you dont know what type is the variable that you want to print
%s	interpolates an string
%d	interpolates a decimal
%f	interpolates a float
%f.n	interpolates a float and format the output constraining the amount of decimals in the number.

Conditionals

Like another languages, Golang has conditionals that permits evaluate statements and execute or not code blocks. Syntax is similar to python as you will see above.

```
if height > 4 {
    fmt.Println("you are tall enough!")
}
```

Also, multiple conditions can be evaluate in a dependant way using else and else if statement. As you will see in the next code snippet, you can put multiple cases.

```
if height > 6 {
    fmt.Println("You are super tall")
} else if height > 4 {
    fmt.Println("You are tall enough")
} else {
```

```
    fmt.Println("You are not tall enough")
}
```

It is important to note that it exist multiple comparators operands, as you see below:

name	symbol
equal	==
different or not equal	≠
less than	<
greater than	>
less than or equal to	≤
greater than or equal to	≥

In an if conditional we can use an special syntax to reduce the amount of code lines. It is possible to declare the variable that will be used in the comparison in the first part of if statement and then use it as a parameter in this comparison, some like this.

```
if length := getLength(email); length < 1 {
    fmt.Println("Email is Invalid")
}
```

Note that we declare two statements, first one get the length and second (after semicolon) is the real comparison that will be taking account in the conditional if. It is so important to be clear that length variable in the previous example, it is only callable in the scope of the comparison, this is not callable above of the declared if.

Functions

As other programming languages, golang has functions, which serves as a way to split code in units that execute an specific part of the code and that could be reused. In Golang we define an specific amount of inputs and outputs for functions, also we declare the types of this inputs and outputs.

```
func sub (x int, y int) int {
    return x * y
```

```
}
```

Function defined above receives two integers and return a single integer according to function definition. In golang, function definition is known as function signature.

If a parameters has the same type, it is possible to specify only in the last one and then, Golang will assume that first n-1 parameters has the same type, some like this:

```
func mult (x, y, z int, msg string) int {
    fmt.Println(msg)
    return x * y * z
}
```

In a function we can use callbacks, that is a way to name the capability of a language to receives functions as a parameters to a function. However, how we see previously, a function has a signature, then, we need to grant that function signature be the same that declared in the function that uses callback.

```
func process_x(func(int, int) int, int) int {}
```

Previous function process_x receives a function which has two input paramters of type integer and a second integer parameter. Also, this function returns an integer as type.

IN GOLANG VARIABLES ARE PASSED BY VARIABLE NOT BY REFERENCE, SO, IT IS NOT PRESENTED THE CASE WHEN A VARIABLE STORES THE MEMORY LOCATION OF A VARIABLE TO REFERENCE THE VALUE.

Ignoring return values

A function can return a value that the caller doesn’t care about. We can explicitly ignore variables by using an underscore as you can see next:

```
func getPoint()(x int, y int) int {
    return 3, 4
}
// ignore the value
x, _ := getPoint()
```

As you can see in the previous example, getPoint function returns 2 values, however, we ignore the second using underscore.

Also, due that golang has output declared, functions can use this declaration to do implicit returns. This is a feature that automatically returns the output when we assign a name in the signature, some like this:

```
func getPoint()(x int, y int) (sum_x_y int, prod_x_y int){
    // some operations
    return
}
```

As you can see in the previous example, we are specify two integers inputs and two integer outputs, so, in the last line when we write the return word, it is not require to put the output variables in order to return. Golan will return automatically the variables declared in the signature as output in the return statement. So, previous code is equivalent to :

```
func getPoint()(x int, y int) (sum_x_y, prod_x_y int){
    //some operations
    return sum_x_y, prod_x_y
}
```

However, for readability, it is better to use explicit returns instead named returns. It is important to note that even implicit returns can be replaced for hardcoded values, some like this.

```
func getPoint()(x int, y int) (sum_x_y, prod_x_y int){
    //some operations
    return 0, 4321
}
```

In this case, implicit return will not take effect.

Structures

An struct is a collection of key values like JSON in JavaScript or dictionaries in python. It is important to take account that structs composed by keys with keys that could be represent by primitive types. A struct should be reusable and is static like any other type in golang, so if you want to use it you should to declare before use it. The declaration of an struct has the next syntax:

```
type car struct {
    Make string
    Model string
    Height string
    Width string
    MaxSpeed int
    MaxAceleration int
}
```

Also a struct could be use another structs in a nested way to represent complex entities, some like this:

```
type car struct {
    Make string
    Model string
    Height string
    Width string
    MaxSpeed int
    MaxAceleration int
    FrontWheel Wheel
    BackWheel Wheel
}

type Wheel struct {
    Radius int
    Material string
}
```

When we want to access to an struct object, we should to use the . operator to run through keys:

```
myCar := car{}
myCar.FrontWheel.Radius = 5
```

In previous code snippet we use dot operator to access the field Radius inside the nested struct Wheel inside of our struct car, and the assign a value for it. Also it is important to note that we instantiate the car struct using the first line in the previous example.

Anonymous Structs

They are structs that are defined without name and due this can’t be reference later in the code. This is useful if you want to create an instance of the struct one time only. This is no common, so you don’t find a lot of top defined struct instances defined like this:

```
myCar := struct {
    Make string
    Model string
}{
    Make: "Tesla",
    Model: "model 3"
}
```

Instead previous use, you will see anonymous definitions like:

```
type car struct {
    Make string
    Model string
    Height int
    Width int
    Wheel struct {
        Radius int
        Material string
    }
}
```

Embedded Structs

Go is not an object oriented language, so, things like inheritance, classes and this class of stuff not exists. However, advantages from OOP are presented in Golang, like embedded structs. If you are defined an struct, and then you want to define another struct that share the same features defined in the first struct, it is possible to embed the struct 1 into the new struct. Some like this:

```
type car struct {
    make string
    model string
}

type truck struct {
    car
    bedSize int
}
```

In the previous example we reuse the fields of car struct inside the truck struct, so it is useful to write a less repetitive code. To generate an instance of a struct that has an embedded struct you should use the next syntax, it is similar to anonymous structs:

```
ptb379 := truck {
    car: car{
        make: "peterbilt",
        model: "379",
    },
    bedSize:3212
}
```

When we use embedded structs, fields belonging to embedded structs can be accesed in the top level of main struct, so, we can do this:

```
model_truck := ptb379.model
make_truck := ptb379.make
```

Struct methods

Like in OOPL, Structs can also have methods associated with them. This methods are just functions that have a receiver. A receiver is a special parameter that goes before the name of the function.

```
type rect struct {
    width int
    height int
}

func (r rect) area() int{
    return r.width * r.height
}

r := rect {
    width : 5,
    height :10
}

fmt.Println(r.area())
```

As you can see in the previous code snippet, we define a struct called rectangle with the fields width and height. Then we define a method, this is defined with the same syntax that a function however we need to add the struct that is owner of this method before the name of the struct method. At this point you can see that this could be useful to access to a data of our struct.

Interfaces

Like embedded structs, it is possible to use a common element between different structs with some things in common to use the same methods. So, a type implements an interface if it has all of the methods that the interface has declared. To declare a interface we use the next syntax:

```
type employee interface {
    getName() string
    getSalary() int
}
```

In the previous code snippet we can see the employee interface, which has two signatures associated getName and getSalary, so the next structs could fulfill the interface and implement it.

```
type partTime struct {
    supervisor string
    workHours int
    payPerHour int
    name string
    lastName string
}

func (pt partTime) getName() string{
    return fmt.Sprintf("%s %s", pt.name, pt.lastName)
}

func (pt partTime) getSalary() int{
    return pt.payPerHour * pt.workHours
}
```

So, partTime struct will implement automatically the interface employee and all methods associated with this. It is so important to remember that Golang don’t need to explicit declare that a struct implements an interface, this is done automatically, AUTOMATICALLY.

Multiple Interfaces

In golang you could declare multiple interfaces so, this interfaces also could be implemented for the same type struct. For example if i have the type soccerPlayer who has the methods shoot, jump and run. Then the interface fieldPerson with the interfaces

run and the interface penaltyShooter whith the interfaces shoot also could be implemented by the type soccerPlayer.

Name interface parameters

Like in function input and output parameters, interfaces can specify a name for the parameters of the signatures that belong to them. So, in this case, we have better readability and clarity over what parameters would be passed in a signature to fulfill the interface. So, we pass from:

```
type Copier interface {
    Copy(string, string) int
}
```

To a named parameters signature like:

```
type Copier interface{
    Copy(SourceFile string, destinationFile string) (bytesCopie
}
```

This change give us a clear expectation about parameters in the interface signatures.

Type Assertion

A useful feature of interfaces is the type assertion among types underlied by an interface. This is a feature that permit us to take a type that fulfill a interface and cast to another method that fulfill the same interface. So, we can do things like:

```
type shape interface{
    area() float64
}

type circle struct {
    radius float64
}

c, ok := s.(circle)
```

```
if !ok {
    log.Fatal("s is not a circle")
}

radius := c.radius
```

In the previous example, s is an instance of shape interface, we want to cast this instance to circle type. Then we use the cast syntax and we get two elements in the return. The first element is the casted value and ok is the error variable that indicates to us if the cast action was successful. ok variable has a boolean type, if it is true, then cast was successful, otherwise was not. ok variable is useful to avoid errors when we try to use methods of the circle type in the casted variable.

This feature is similar to isinstance function in python.

Type Switch

type switch is a elegant way to do type assertion with an interface. So, instead of use a repetitive call for the cast method we use only one cast and check by each probably type as you can see follow.

```
func printNumericValue(num interface{}) {
    switch v:= num.(type) {
    case int:
        fmt.Println("%T\n",v)
    case strubg:
        fmt.Println("%T\n",v)
    default:
        fmt.Println("%T\n",v)
    }
}

func main(){
    printNumericValue(1)
    printNumericValue("1")
```

```
    printNumericValue(struct{}{})
}
```

As you can see, now we have a small way to do type assertion with a lot of probably types of an interface.

Clean Interfaces

This is a some tips to do code abstraction to avoid complex code that not need to be complex.

1. Keep Interfaces Small: It is the most important advice, interfaces in golang define the minimal behavior necessary to accurately represent an idea or concept. Concept are related with simple things like only one responsability, task, job. Not use interfaces to represent general groups or concepts.

   ```
   type File interface {
       read() uint8
       write() bool
       edit() bool
       open() bool
       close() bool
   }
   ```

   Previous example shows a well declared interface that represent all posible methods that could be meet any kind of file.

2. Interfaces should have no knowledge of satisfying types: An interface should have clear requirements to define when a type can belong or not to them. For example, if we want to define the interface car we can take features as Color, Speed, however, specific features like isSedan or isTank are some specific an will generate redundant an repetitive information. However, types should be aware of what interfaces satisfy them, this is important from the developer perspective.

3. Interfaces are not classes: are simples with respect to classes. It not exists constructors or deconstructors methods. They are not hierarchical. Interfaces define function signatures but no define underlaying behaviour. That means that we can

control the behaviour or methods of type members, we need to code each type method by individual type.

Errors

Errors in golang are treated different that in languages like JS, Java or Python. In common languages we usually see structure try catch with code blocks and exceptions inside the catch section. In Golang, we generate an error by each potential dangerous function at same time that we generates the result. So, we can check if all in our dangerous function goes well. Some like this:

```
user, err := getUser()
if err != nil {
    fmt.Println(err)
    return
}
```

As you see in the code snippet, our dangerous function is getUser, so we got the user response and the error. To know if our call of getUser function was success we check the err variable, if it is equal to nil we grant that call was successfully executed.

Error Interface

If we check the errors, is not more than an interface that returns a string. However, this string should be handful for developers and any other than will see the error string. So, we need to grant that this strings has a correct data and format, we need to use the correct formatting verbs and texts.

Custom Error Interface

It is possible to create our own error interface to returns custom errors. For example, we can create a interface devoted to handle the user Errors, some like this:

```
type userError stuct {
    name string
}
```

```
func(e userError) Error() string{
    return fmt.Sprintf("%v has a problem with their account", e
}

func sendSMS(msg, userName string) error {
    if !canSendToUser(userName){
        return userError{name: userName}
    }
}
```

Loops

As any other programming language, Golang has loops to develop iterative or repetitive process over iterable structures. So, the basic syntax is shown below:

```
for INITIAL; CONDITION; AFTER{
    //do something
}
```

When INITIAL is a statement that runs once at the beginning of the loop and can create the variables within the scope of the loop. Similar to if statements when we create a variable with limited scope. CONDITION is checked before each iteration, so we know if continue with the iteration. AFTER is the statement that runs after each iteration, generally, this part is reserved to increase the value of the counter variable that controls the loop. An example of basic loop in Golang is:

```
for i:= 0; i<10; i++{
    fmt.Println("current vaue of i: %d", i)
}
```

Omitting conditions in the loop definition

it is possible to omit some statements of the loop declaration to get a specific behavior. As the first step, we will omit the condition step to stop the loop execution. As you see

in the next function, the condition was not defined, however, the loop will stop after the return statement is executed.

```
func maxMessages(thresh float64) int {
    costComputed := 0.0
    for msgCount := 0; ; msgCount++ {
        costComputed += 1.0 + float64(msgCount)*0.01
        if costComputed > thresh {
            return msgCount
        }
    }
}
```

While Loop

Like the previous subsection, using the omission of certain statements it is possible to get the while loop behavior, just using the next syntax:

```
for CONDITION {
    // code block
}
```

Break and Continue

It exist the common in some languages words continue and break statements, so, use they.

```
for i:=0; i<limit; i++{
    if i == 5 {
        break
    }
    if i == 10 {
        continue
    }
}
```

Slices

They are the implementation of arrays in Golang. Arrays has an static size, this is defferent to another programming languages where they has dynamic size. Syntax to declare it is as shown below:

```
var myings [10] int
```

In previous example we declare an array with a size of 10 and filled with integer elements. to declare and initialize an array we use the next syntax:

```
primer := [6]int{2, 3, 5, 7, 11, 13}
```

Slice is a dynamic size flexible view into an array. I could create a slice that only takes some part of an array.

```
a := [6, [3,2,6], 5]
b:= a[1:4]
```

Slices are used over arrays due their capability of dynamic handling over arrays. At least, for get slices from an array we should use the next syntax:

```
primes := [6]int{2,3,5,7,11,13}
mySlice := primes[1:4]
mySlice[lowIdx:highIdx] ->
mySlice[:highIdx]
mySlice[lowIdx:]
mySlice[:]
```

Slices reference and how it works when they are passed to a function

Slice reference an array, so, it take an array associated. Multiples array could reference the same array. It is important to note that when we talk about pass slice to function, we pass the reference, so it is important to note that changes applied inside the function will modify the referenced array.

Slices creation using make

To create a slice we can use the built-in function make, it takes the type definition, the len and the capacity, so we can define it.

```
mySlice := make([]int, 5, 10)
```

The previous defined array will store only integer values, will have a len of 5 and have tha capacity to increase his value until 10 elements. In case that you don’t know the possible max length that will have your slice you can omit the capacity parameter, some like this:

```
mySlice := make([]int, 5)
```

By this way we can create slices. We should remark that cap is the maximum value that will be take an array before call reallocation. In the other side, len is the amount of addresses used by the array, this addresses are different for the reserved ones.

Variadic Functions

Similar to python * operator and JS … operator in Golang it exists a way to pass multiple parameters to the function and receive it as a single slice. This is done by mean of :

```
func sum(nums ...int)(cost int){
    // in this function nums behaves as an slice
    cost := 0
    for i:=0; i < len(nums); i++ {
        cost += nums[i]
    }
    return
}
```

Also we can use spread operator to pass slices to functions like the previous one, some like this:

```
func main(){
    myTestSlice := int[] {1,2,3,4,5,6,7,8,9}
    sumVal := costsum(...myTestSlice)
}
```

Append in slices

Append is a method associated to slices that permits handle dynamic increase of len and cap of the array by mean of adding a new element at the last position of the slices array. This work with the next syntax:

```
slice = append(slice, element) // append a single element to the
slice = append(slice, element, element1) // append multiple elem
slice = append(slice, anotherSlice...) // append another slice t
```

Slice of Slices

Slices can hold another slices, this is useful to represent data types like multidimensional matrix. It is important to note that to access data from slice we use the next expression:

```
data_ij := sliceOfSlices[i][j]
```

Common mistakes

It is not recommendable to use append and assign the return slice to another variable. This is due that we are using a memory address and not a value, then, when we create the new slice, we are passing the same memory address to two variables. This could be lead in strange behaviors like change in one slice due re assignation of value in the another one, and this kind of stuff. This behavior occurs when array capacity is not overpassed and reassignment is not occurs, so it is passed the same address direction. In the other hand, reassignment avoid to pass the same memory address. So, it is not recommended to do the next assignation, instead use always assign the return value of append to same slice.

```
newSlice := append(primarySlice, value)    // XXXXX dangerous wa
```

Range

Go provides a syntactic sugar to iterate easily over elements of a slice, this works with the next syntax and it produce a readable code:

```
for INDEX; ELEMENT := range SLICE {
    // stuff
}
```

Maps

Are a similar implementation of JavaScript Objects JSON, python dictionaries or ruby hashes. Maps are a data struct that works key → value mapping. The zero value of a map is nil. We can create a map using a literal or by using the make() function, some like this.

```
ages := make(map[string]int)
ages["john"] = 37
ages["Mary"] = 24
ages["Mary"] = 21 // overwrites 24
```

Another way to declare the map is:

```
ages = map[string]int{
    "john":37,
    "Mary":21
}
```

The len function works on a map, it returns the total amount of key value pairs.

```
ages = map[string]int{
    "John":37,
```

```
    "Mary":21
}
fmt.Println(len(ages)) // prints 2
```

Maps are highly valuable compared to arrays because they are optimized for efficient searching using keys. This is not possible in structs or slices, as they lack an index to directly access data.

Mutations over maps

In maps we can use the next operations to edit, access … and other mutations over the map.

Insert

```
m[key] = elem
```

Get an Elem

```
elem = m[key]
```

Delete an Element

```
delete(m, key)
```

Check if a key exists

```
elem, ok := m[key]
```

Key Types

In the definition of maps in language specification, key types should be comparables, so go maps should use keys types that could be compared and that comparison returns coherent values, think that comparing two maps according to an slice is a bad approach

due that values could be the same but keys are different. Then, always you shoud to select properly key types like integers, numbers, ……

Simpler

As JSON or dictionaries, in Go, maps could be nested in more than a level, however this is not recommendable due that a try to access data in non existent key will lead to a panic and also stop in the execution code. According with this, simpler is the definition of a single level of nesting that takes as map key type an structure, with this approach you could wrap more information by entity without create a multiple nested map. Some like this:

```
type Key struct {
    Path, Country string
}
hits := make(map[key] int)
```

This is a better approach than:

```
hits := make(map[string]map[string]int)
```

last implementation is tedious and for a simple add operation we will require a complex function like:

```
func add(m map[string]map[string]int, path, country string) {
    mm, ok := m[path]
    if !ok {
        mm = make(map[string]int)
        m[path] = mm
    }
    mm[country]++
}
```

Recommendations from effective GO

Like Slices, Maps hold references

If you remember the slices, they reference an array with a specific length. This occurs also with Maps, which reference to an underlying data structure. So, if we pass a map to a function this function modifies the map, then the original map also will be modified.

Map Literals

Maps could be constructed using the syntax of colon separated key-value pairs, so it is easy to build during initialization.

Missing keys

An attempt to fetch a map value with a key that is not present in the map will return a zero value for the type of the entries map. So, if was have a map with key type string and value type int and we try to get a non existent key in the map it will return 0 value.

```
attended := map[string]bool{
    "Ann": true,
    "Joe": true,
}

if attended[person] { // will be false if person is not in the m
    fmt.Println(person, "was at the meeting")
}
```

Sometimes we need to distinguish between a missing entry and a zero value, in other cases this situation will lead to an undesired behavior due to logic fault. For this case, we need to check the existence using the check if exists syntax:

```
var seconds int
var ok bool
seconds, ok = timeZone[tz]
```

Deleting Map entries

To delete a map entry, it is recommended to use the built-in function, this function is too safe, instead in the case where the provided key name to delete is not present in the

map.

Nested Maps

Maps can contain maps, creating a nested structure. For example

```
map[string]map[string]int
map[rune]map[string]int
map[int]map[string]map[string]int
```

Advanced Functions

First Class and Higher order functions

First Class function is a term using in programming languages when they languages can treat the functions as any other variable. So, this permits us to pass the function to another function as a variable and assign it to a variable as we shown in the next example.

```
func add(x, y, int) int{
    return x + y
}

func prod(x, y, int) int{
    return x * y
}

func aggregate(a, b, c int, arithmetic func(int, int) int ){
    return arithmetic(arithmetic(a, b,), c)
}

func main(){
    fmt.Println(aggregate(1,2,3,add))
```

```
    fmt.Println(aggregate(1,2,3,prod))
}
```

As you can see in the previous example, we passed our functions add and prod to aggregate function as a parameter. We only need to specify in the signature of the function when we need to pass our function a signature that match with the definition of the function to pass. In this case, the function to pass are prod and add and the function that we use function as arguments is aggregate. So what is a first class function and what is a High order function.

Higher order function

Higher order function are those functions that takes another functions as argument or as a return value. Like aggregate function in the previous example

First Class Function

A first class function is a function that can be treated as any other value in Golang. Go supports this feature. It is important to note that the functions type is dependent of the types of its parameters and return values, so the next function types are differents:

```
func()int
func(string)int
```

Function Currying

Function currying is a practice of writing a function that takes another function to return a new function. You could check an example next:

```
func oneBaseIndexCorrection(funIndex func(int)int) func(int) int
    return func(i int){
        j := funIndex(i)
        if j <= 0{
            return 1
        }
        return j
```

```
    }
}
```

In the previous function we create a function that receives any function with integer input and return 1 if value is less than 1. This function will fix index values in one base indexation systems.

Defer

The defer keyword is a fairly unique feature of Go, It allows a function to be executed automatically just before its enclosing function returns.

The deferred call's arguments are evaluated immediately, but the function call is not executed until the surrounding function returns.

Deferred functions are typically used to close database connections, file handlers and the like.

It is important to note that it not imports if function returns in any point of the program, after defer function call, it will executed immediately before wrapper function returns.

```
// CopyFile copies a file from srcName to dstName on the local f
func CopyFile(dstName, srcName string) (written int64, err erro

  // Open the source file
  src, err := os.Open(srcName)
  if err != nil {
    return
  }
  // Close the source file when the CopyFile function returns
  defer src.Close()

  // Create the destination file
  dst, err := os.Create(dstName)
  if err != nil {
    return
  }
  // Close the destination file when the CopyFile function retu
```

```
  defer dst.Close()

  return io.Copy(dst, src)
}
```

In the previous example, defer function could be executed in the defer after error in os.Create function or after return the value of io.Copy function but before return it from the CopyFile function.

Closures

A closure is a function that references variables from outside in its own function body. The body function may access and assign to the referenced variables.

In this example, the concatter() function returns a function that has reference to an enclosed doc value. Each successive call to harryPotterAggregator mutates that same doc variable.

```
func concatter() func(string) string {
    doc := ""
    return func(word string) string{
        doc += word + " "
        return doc
    }
}

func main(){
    HPAgg := concatter()
    HPAgg("Mr.")
    HPAgg("and")
    HPAgg("Mrs.")
    HPAgg("Dursley")
    HPAgg("of")
    HPAgg("number")
    HPAgg("four")
    HPAgg("Privet")
    fmt.Println(HPAgg("Mr."))
```

```
}
```

In the previous example we create the function using the currying and assign to value HPAgg. concatter function returns a function that appends strings to an existent string that persist and is defined outside of the returned function, in this case this value is doc. After each call of the returned function, doc will be modified by mean of the append of the passed string.

It is important to remark that closures can mutate variables that were declared outside of its own body function.

When a variable is enclosed in a closure, the enclosing function has access to a MUTABLE REFERENCE TO THE ORIGINAL VARIABLE. If there was a copy of the value like in normal functions, the sum or cumulative and memory effect will not take effect.

Anonymous Function

Anonymous functions are functions that are defined only using the reserved word func and that creates a function that could not be called more times, only once to be used or to create a quick closure.

```
// doMath accepts a function that converts one int into another
// and a slice of ints. It returns a slice of ints that have bee
// converted by the passed in function.
func doMath(f func(int) int, nums []int) []int {
    var results []int
    for _, n := range nums {
        results = append(results, f(n))
    }
    return results
}

func main() {
    nums := []int{1, 2, 3, 4, 5}
```

```
    // Here we define an anonymous function that doubles an int
    // and pass it to doMath
    allNumsDoubled := doMath(func(x int) int {
        return x + x
    }, nums)

    fmt.Println(allNumsDoubled)
    // prints:
    // [2 4 6 8 10]
}
```

In the previous example we can see the anonymous function defined inside the doMath High Order function. Our anonymous function works as a first class function in the previous context.

Pointers

As we learn along this process, variables are stored in memory address. Variables are also passed to functions as value and they were copied internally by Go. However, There exists a way to pass, and interact with a variable without passing, copy and return. To do this we use pointers, this is a feature that permits us to store the memory address in another variable that we will call pointer. it give us access to data and also give us the capability to mutate the variable that is stored. This feature is widely spread in languages like C or C++. Next example show us how to generate a single pointer to a variable that stores an integer.

```
// we have the next variable
a := 42
// if we want to store the memory address we need to use a point
// pointers are references to a certain variable with an specifi
var p * int
// pointers are initialized as null in some cases, like above, t
// then, to give the reference we uses anpersand operator to obt
p := &a
// by this way, we have the reference of variable a in the point
```

But.. Why use pointers ?

Pointers allow us to manipulate data in memory directly, without making copies or duplicating data. This can make programs more efficient.

Pointers are so useful, but they are also dangerous if they are not used properly, you could generate bugs due changes in memory by direct manipulation

Nil Pointers

Like any other variable in Golang, pointers also have a zero, also known as default value. In this case pointers has a nil value when they not point to any other variable. This is important to know that if pointer point to nothing, it will raise a panic state when we try to dereference with * operator. To avoid this, it is important to check if pointer in nil or not, in other case will go to panic.

Pointers Receivers

A type receiver on a method can be a pointer

Methods with pointer receivers can modify the value to which the receiver points. Since methods often need to modify their receiver, pointer receivers are more common than value receivers. Remember pointer receivers as the way that we pass the the struct instance.

```
type car struct {
    color string
}

func (c *car) setColor(color string){
    c.color = color
}

func main(){
    c:= car {
        color: "white"
    }
    c.setColor("blue")
```

```
    fmt.Println(c.color)
}
```

There are so useful due that permits in a single way to mutate the instance of struct, it is different when no pointer receivers that non pass a reference, in this case the value of the instance don’t mutate at least that we return from the method. It is great to note than you don’t need to specify in the method or pass a pointer, only with the signature definition we grant this.

Local Development

In this chapter we will cover the main topics related and that concerns when we develop with go in our local computers.

Packages and type packages

Packages are like in other programming languages, a set of functions and instructions of the language that solve specific problem. They are known also as libraries in other programming languages.

In go it exists two types of packages, main package and library packages.

Main Package

Main package contains the program that is executed standalone and also is the piece of code that will be compiled. Package has an entrypoint at the main function and execute all the instructions contained in this function. If a golang program has not the main package definition, it will not execute any piece of code due that it doesn’t contains an entrypoint.

Libraries Package

Libraries package are packages that has not executed standalone and also has not an entrypoint, instead, they only are pieces of code that export functionality that can be used for other packages, even the main package.

Package Naming

By convention, a package name is the same as the las element of its import path. For instance, the math/rand package comprise files that begin with:

```
package rand
```

That said, package names aren’t required to match their import path. For example, I could write a new package with the path github.com/mailio/rand and name the package random:

```
package random
```

While the above is possible, it is discourage for the sake of consistency.

One Package / Directory

A directory of Go code can have at most one package. All .go files in a single directory must all belong to the same package. If they don’t an error will be thrown by the compiler. This is true for main and library packages.

Modules

As we see previously, go is organized in packages. A package is a directory of Go code that’s all compiled together. Functions, types, variables, and constants defined in one source fiel are visible to all other source files within the same package directory.

A repository contains one or more modules. A module s a collection of Go packages that are released together.

A Go repository Typically contains only one module, located at the root of the repository

A file named go.mod at the root of a project declares the module. It contains:

1. The module path
2. The version of the Go language your project requires
3. Optionally, any external package dependencies your project has

The module path is just the import path prefix for all packages within the module. Here’s an example of a go.mod file:

```
module github.com/bootdotdev/exampleproject

go 1.20

require github.com/google/examplepackage v1.3.0
```

Each module's path not only serves as an import path prefix for the packages within but also indicates where the go command should look to download it. For example, to download the module golang.org/x/tools, the go command would consult the repository located at https://golang.org/x/tools

Import Path

As we said before, module could contains one or multiple packages in the same path. So, we also can import directly each one of these modules. To do this we use the import modules syntax. This syntax is:

```
modulePath + packageSubdirectory // syntax
// example module
module github.com/bootdotdev/exampleproject
// import path
github.com/bootdotdev/exampleproject/pckg1
```

GOPATH (outdated AVOID)

in the first golang implementations, we need to use a specific directory to run an interact with our go programs, this was replaced by the modules and packages implementation. So, don't put code or any file of your project in the GOPATH. This is solved by modules and packages.

Channels and Concurrency

First of all we need to define what is the concurrency. The concurrency is the ability to perform multiple tasks at the same time. Concurrency is only available when the tasks that we want to execute are not dependant each one from others. In case that task has a dependency from the others t is not possible to execute at same time due that is a

sequential process. As we seen in the previous examples along this process we check that we have a unique thread that executes code. In this case we want to execute tasks in a separate core from the main thread to do this we have the next syntax that generates a new goroutine.

```
x:= "data"
go doSomething()
// continue execution
```

In this example, doSomething() will be executed concurrently with the rest of the code of the function, something like this:

As you can see in the image,m it exist a moment when code behaves as a sequential process, after that go routine is created and a new process that is executed parallel to the main thread. It is important to note that doSomething function don’t return anything, this have sense due that it works to the main thread as a async function, it continues with the normal operation, so event if it returns something maybe, at the moment of return program will not wait this value or it will execute some lines ahead.

Channels

Channels are a typed, thread-safe queue. Channels all different goroutines to communicate with each other.

Create Channels

Like maps and slices, channels must be created before use. They also use the make keyword to do this.

```
ch := make(chan int)
```

Send data to channel

To send data to a channel we use the ← operator, who is called channel operator. Data flows in the direction of the arrow. This operation will block until another goroutine is ready to receive the value.

```
ch <- 69
```

Receive data from channel

To read and remove data from a channel and save this data into a variable v we use also the operator ←. This operation will block until there is a value in the channel to be read.

```
v := <- ch
```

Blocking and deadlocks

A deadlock is when a group of goroutines are all blocking so, the program will be wait indefinitely. This condition should occurs when go routines don’t response to the channel or it not exists go routines that receives the data from the channel.This is a common bug that you need to watch out for in concurrent programming.

```
func filterOldEmails(emails []email) {
    isOldChan := make(chan bool)

    sendIsOld(isOldChan, emails)

    isOld := <-isOldChan
    fmt.Println("email 1 is old:", isOld)
    isOld = <-isOldChan
    fmt.Println("email 2 is old:", isOld)
    isOld = <-isOldChan
    fmt.Println("email 3 is old:", isOld)
}
```

As you can see in the previous example, channel is created in the function and this is used after to get data. The problem with previous function is that sendIsOld function was called as a common function and not as a goroutine. Due this, data should be extracted directly or using a pointer not using a channel due that channel operation will block the execution until a goroutine returns data via the channel.

Tokens

Empty structs are often used as tokens in the channels. Sometimes, we want to know only if the channel are sending the data. So, to check data send in the channel we can send empty structs. Also, if we don’t care the value, only know if channel works we can use the next syntax that block the execution until something come from the channel.

```
<- ch
```

Buffered channels

A buffered channels is a channel implementation that receives multiple values and fill something like a list before execute the block and while receiving and send data. Buffered channels are defined with the next syntax

```
ch := make(chan int, 100)
```

In the previous case, this channel will receive 100 integers elements before send data. This is useful if we want to do batch operations, for example wait for the response of multiple instances of a service and then commit to a database.

It is important to note that Sending on a buffered channel **ONLY BLOCKS WHEN THE BUFFER IS FULL**

It is important to note that receiving from a buffered channel **ONLY BLOCKS WHEN THE BUFFER IS EMPTY**

Closing channels

Channels can be explicitly closed by a sender. Remember that a sender is the side that put data in the channel that will be read by the goroutine. So, when we know that channel will not be used anymore it is a good practice to close this channel. To close the channel we use the next syntax.

```
ch := make(chan int)
close(ch)
```

By this way, the channel ch was closed. Due that it is possible that code will call the channel to send data, it exists a way to check if channel is closed or active. The next

syntax presents the way to check this using the ok variable. When ok variable is false then we know that channel is empty and closed.

```
v, ok := <-ch
```

Previous check step is important due that sending data into a close channel will cause a panic in your program.

It is important to note that isn’t necessary to close the channels, the garbage collector will act normally over the channel, this is only a good practice to indicate explicitly to a receiver that nothing else is going to come across.

Range

Similar with the slices, it is possible to iterate over a channel buffered or none using the reserved keyword range.

```
for item := range ch{
    // item is the value received from the channel
    // ....
    close(ch)
    // .../

}
```

Values are received from the channel, this channel will be blocked at each iteration if nothing new is there and will only break the loop when the channel is closed

Select

In certain programs we can use a single goroutine that listen multiple channels and want to process data in the order it comes through each channel. A select statement is used to listen to multiple channels at the same time. It is similar to a switch statement but for a channel type.

```
select {
    case i, ok := <- chInts:
        fmt.Println(i)
```

```
    case s, ok := <- chStrings:
        fmt.Println(s)
}
```

In the previous example, the first channel with a value ready to be received will fire and its body will execute. If multiple channels are ready at the same time one is chose randomly. The ok variable in the example above refers to whether or not the channel has been closed by the sender yet. Also, we can add a default case to the select statement. This condition help us to avoid a block condition when channels has not inputs available. The syntax is:

```
select{
    case v:= <- ch:
        // use v
    //.....
    default:
        //default behaviour
}
```

Tickers

There are a golang build in functions that are useful to test channels and interact with them based on time variables.

time.Tick() is a standard library function that returns a channel that sends a value on a given interval.

time.After() sends a value once after the duration has passed

time.Sleep() blocks the current goroutine for the specified amount of time

These functions are very powerfull when we want to test channels, send data after certain operations or block a channel in order to avoid a third party service block or fault.

Read Only Channels

Until this time, we have work with channels with multiple purposes, send and receive data from the goroutine. However, it is possible to limit the behavior of a channel to a

single operation, read in this case. This is possible using the next syntax.

```
func main(){
    ch := make(chan int)
    readCh(ch)
}
func readCh(ch <- chan int){
    // ch can only be read from in this function
}
```

Write only channels

Same as the previous read only channel, we can limit the behavior to only permits write.

```
func writeCh(ch chan<- int){
    // ch can only be written to
    // in this function
}
```

Previous described implementations only limits the behavior inside the functions, after return data from the function channel will have both functionalities.

Additional considerations of channels and concurrency

A send or receive from or to a nil channel will blocks forever

As you can see in the previous example, we have use make function to create our channels, instead of declaration using reserved keyword var. This is due this initialization way avoids to return a nil channel. When we try to interact sending or receiving data from a nil channel the operation will block our execution forever. Then avoid things like:

```
var c chan string
fmt.Println(<-c)
//or
c <- "text that never works"
```

A send to a closed channel panics

As we mentioned previously, when you try to send data to a closed channel will generate a panic condition in you go routine. **Then Avoid things like:**

```
var c = make(chan int, 100)
close(c)
c <- 1 // this fu*k line will panic the program
```

A receive from a closed channel returns the zero value immediately

When we try to read data from a closed channel we will get a zero value according to the channel type value

```
var c = make(chan int, 100)
close(c)
fmt.Println(<-c) // returns a 0 value
```

Mutexes

Mutexes are a library implementation in Golang that allow us to lock access to data. Previous indicates that we can control which go routines can access certain data at which time. Library is accessible with the type sync.mutex type and its two method

- .Lock()
- .Unlock()

We can protect a block of code by surrounding it with a call to Lock and Unlock as shown in the next method below.

It’s a good practice to structure the protected code within a function so that defer can be used to ensure that we never forget to unlock the mutex.

```
func protected(){
    mu.Lock()
    defer mu.Unlock()
```

```
    // the rest of this function is protected
    // any other calls to 'mu.Lock()' will block
}
```

We should to say that maps are not thread safe in concurrency. Maps are not safe for concurrent use! if you have multiple goroutines accessing the same map, and at least one of them is writing to the map, you must lock your maps with mutexes.

Mutex is an achronym to mutual exclusion. Remember that mutex only locks one thread, the thread that callls Lock and Unlock functions.

Rw Mutex

Maybe you ask why multiple read over data structs is non safe? well, really multiple read over data structs like maps is safe. The problem is only limited by writing and reading or writing and writing operations by multiple goroutines. Then we can lock only the write operation using RW mutex. This means that we can have multiple RLock() calls simultaneously. However, only one goroutine can hold Lock and all RLock() will also be executed. This means that only can happens one call of Lock at same time while occurs multiple calls of RLock and RUnlock.

Generics

In the recent releases of Golang, it was added a feature that enhance the DRY principle, this feature is the generics. Generics is an implementation that permits use the same code for similar data types. For example:

```
func splitIntSlice(s []int) ([]int []int){
    mid := len(s)/2
    return s[:mid], s[mid:]
}

func splitStringSlice(s []string) ([]string []string){
    mid := len(s)/2
```

```
    return s[:mid], s[mid:]
}
```

As you can see in the previous example, code in both functions are the same and results will be the same without care data type, so, why we should to repeat the code and not write a single function that could be used by both data types slices ? Well, this is possible using generics. Basically, generics permits us to use variables to refer to specific types. This is useful due that permit us to write abstract functions that could be used by a variety of data types. Translating the previous functions, we get:

```
func splitAnySlice[T any](s []T)([]T, []T){
    mid := len(s)/2
    return s[:mid], s[mid:]
}
```

In the previous generic implementation, T is the name of the type parameter for the function. any keyword indicates that not exists a constrains with respect the datatype, so, any kind of datatype slice could be passed to this function and this will be admitted by Golang. We can call the previous function like:

```
firstInts, secondInts := splitAnySlice([]int{0,1,2,3})
fmt.Println(firstInts, secondInts)
```

It is important to note that generics variable has a value zero, so when we declare a variable using the var keyword it will take the default value of the inserted variable.

```
var myZeroInt int
var myZeroT T
```

Advantages of generics

Generics are so important and powerful due that permits reduce the amount of repetitive code that should be write when we use specific types. Generics are useful in

packages or modules due the abstraction, cohesion and decoupling required in this kind of code pieces.

Constrains

Sometimes, functions could not be available to receive any type. Instead, we need to know specifically what is the type that we received. Constrains are interfaces that allow us to write generics that only operate within the constrain of a given interface type. In the previous examples we use the constrain any that group all types. Now, we will see how to create a custom constrain.

We will implement a concatenate function. It takes slice of values and concatenates the values into a string. This should work with any type that can represent itself as a string, even if it's not a string under the hood. For example, a user struct can have a .String() method that returns with the user's name and age.

```
type stringer interface{
    String() string
}

func concat[T stringer](vals []T) string{
    result := ""
    for _, val := range vals{
        // this is where the .String() method is used
        // That's why we need a more specific constraint instead
        result += val.String()
    }
    return result
}
```

Interface Type List

If we want to create a generic based on specific types we can use interface type lists. This can be done adding a list of enable types with the next syntax:

```
type Ordered interface {
    ~int | ~int8 | ~int16 | ~int32 | \\ ...
```

```
\\ ...
    ~string | ~uint64
}
```

Parametric Constrains

If you have defined interface definitions, these can be later be used as constrains, can accept type parameters as well. This give us an additional layer to specify what kind of parameters should receive our interface methods.

```
type store[P product] interface{
    Sell(P)
}

type product interface{
    Price() float64
    Name() string
}

type book struct{
    title string
    author string
    price float64
}

func (b book) Price() float64{
    return b.price
}

func (b book) Name() string{
    return fmt.Sprintf("%s by %s", b.title, b.author)
}

type toy struct {
    name string
    price float64
```

```
}

func (t toy) Price() float64{
    return t.price
}

func (t toy) Name() string{
    return t.name
}

type bookStore struct {
    booksSold []book
}

func (bs *bookStore) Sell(b book){
    bs.bookSold = append(bs.BookSold, b)
}

type toyStore struct {
    toysSold []toy
}

func (ts *toyStore) Sell(t toy){
    ts.toysSold = append(ts.toysSold, t)
}

func sellProducts[P product](s store[P], products []P){
    for _, p := range products {
        s.Sell(p)
    }
}
```

AS you can see in the previous implementation, we define two interfaces store and product, this interfaces are combined to specify what type was used as parameter on the store interface. In the Store interface we specify that the method Sell only receives

types that fulfill the interface product. This is so important due that permits to create more abstract code in order to specify what will receive our interface methods.

Naming generic types

As you can see in our previous example we name our types using T letter.

```
func splitAnySlice[T any](s []T)([]T, []T){
    mid := len(s)/2
    return s[:mid], s[mid:]
}
```

We can change T for any other value, so, our previous function with specific type could be write as:

```
func splitAnySlice[MyAnyType any](s []MyAnyType)([]MyAnyType, []
    mid := len(s)/2
    return s[:mid], s[mid:]
}
```

however, T is a common convention to make readable code.

Go proverbs

- Don't communicate by sharing memory, share memory by communicating.
- Concurrency is not parallelism.
- Channels orchestrate; mutexes serialize.
- The bigger the interface, the weaker the abstraction.
- Make the zero value useful.
- interface{} says nothing.
- Gofmt's style is no one's favorite, yet gofmt is everyone's favorite.
- A little copying is better than a little dependency.
- Syscall must always be guarded with build tags.

- Cgo must always be guarded with build tags.
- Cgo is not Go.
- With the unsafe package there are no guarantees.
- Clear is better than clever.
- Reflection is never clear.
- Errors are values.
- Don't just check errors, handle them gracefully.
- Design the architecture, name the components, document the details.
- Documentation is for users.
- Don't panic.