**Learning Go Programming Language/GoLang** **Background**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

## Creators: small team at google, Robert Griesemar, Rob Pike, and Ken Thompson

* Python was easy but slow, Java: increasingly complex type system and c/ c++ was quick but also a complex type system and slow compile times. So created go.
* Strong and Statically typed means the type of variable cannot change over time so an int cannot be changed to a bool.
* Features to get around
* Excellent community
* Key features
  + Simplicity:
  + Fast compile times: Go focuses on keeping time down
  + Garbage collected: Do not have to manage your own memory.
  + Built- in concurrency.
  + Compile to standalone binaries: everything bundled into single binary

**Useful resources:**

* golang.org
* effective go from documents on golang.org
* golangbridge.org
* play.golang.org

**Notes for GO**

* Import (“fmt”) allows us to format strings
* Func main () {} always the entry point
* Fmt.Println(“Hello, playground”)
* Go run path to code
* Go build: takes the package path
* Go install expected to be pointed into a package. Package path and creates a bin folder.

**Agenda**

* Golang provides the gofmt tool to encourage and safeguard that the Golang code is formatted in line with an expected convention.

Variable declaration

* + Var car string: Var (name of the variable) (variable type) then car = “Hyundai” to set the variable
  + **Can only print inside a function**
  + **Will not run if you have a variable set but not used**
  + **Lower case package level variables are scoped to just the package.**
  + **Upper case package level exported from the package to be global scoped.**
  + **Blocked scope**
  + **Exported or package level variables should be clear and concise**
  + **Short names are ok for variables that are used immediately. Variables should be named concisely**
  + **Keep acronyms all upper case for example Var theURL string = “https;//google.com” for readability**

Example: var car string= "Hyundai"

**or**

var car string

car = "hyundai"

**or**

car := "Hyundai"

* Can only be used inside of a function.

**A Variable Block(at the package level)**

var(

cartype string = "Truck"

carman string = "GMC"

caryear int = 2020

carnum int = 3

)

var(

A = 7

B = "hi"

)

**Checking the type of a variable**

fmt.Printf("%v,%T",car,car)

* + **%V =Value**
  + **%T=Type**
* **Declare a variable but not ready to initialize.**
* **Float32 to set float**
* **At package levels variables can be set**
* **At package level a block of variables can be set**
* func main() {
* var i int = 42
* fmt.Printf("%v,%T\n",i,i)
* // "%v,%T prints the value and type.
* var j string
* j= strconv.Itoa(i)
* //strconv.Itoa = converts other data types to strings
* fmt.Printf("%v,%T\n",j,j)
* }
* Will print the last variable that was set. If you say car = “Hyundai” and then car = “Audi” car will be Audi as it was set last.
* Redeclaration and shadowing
* Visibility
* Naming Conventions
* Type Conversions

Summary

* Cannot redeclare variables but can shadow them.
* All variables must be used
* Visibility rules
  + Lower case first letter for package scope
  + Upper case first letter to export globally
  + No private scope
* Naming conventions
  + Pascal (uppercase first letter) or camelCase
  + Capitalize acronyms (HTTP,URL)
  + As short as reasonable
  + Longer names for longer use
* Type Conversions
  + DestinationType(variable)
  + Use strconv package for strings

Primitives Agenda

* Boolean Types
  + == to compare items
* n := 1 ==1
* m := 1 ==2
* fmt.Printf("%v,%T\n",n,n)
* fmt.Printf("%v,%T\n",m,m)
  + (Var n bool = true) to set the datatype as a Boolean
  + **Whenever you initialize a variable without stating the bool value it = false**
* Numeric Types

Int8: goes from -128 to 127

Int16: goes from -32768 to 32,767

Int32: goes from -2,147,483,648 to 2,147,483,648

Int64: goes from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,808

Any other larger numbers need to use the big package.   
A picture containing drawing

Description automatically generated

Float 32bits: -3.4E+38 to +3.4E+38 about 7 decimal digits

Float54 bits: -1.7E+308 to +1.7E+308 about 16 decimal digits

* + Integers: Uint8,16,32 unsure int
    - Subtract = -
    - Divide = /
    - Add = +
    - % = remainder
    - Cannot do math on mitch match int types. Will have to convert integer to the same type.
    - & and adds the items using binary: **10 = 1010(in binary) 3 = 0011(binary)** add them and get 0010 which is 2
    - **|** = or which means if one or the other has a bit set 10 | 3 = 1011 which is 11
    - **^** = exclusive or means one or other has a bit set 10 ^ 3 = 1001which is 9
    - **&^** = and not only true if neither of the numbers have the bit set. 10 &^ 0100 which is 8
  + Bit shifting
    - **A := 8 also 2^3**
    - **A<<3 =64 2^3 \* 2^3 = 2^6 basically adding to the exponent = 64**
    - **A>>3 = 1 take original number and divide it by however many we are shifting so 2^3 / 2^3 = 2^0 which = 0**
  + Floating Point
  + Complex numbers
    - Complex64 =
    - Complex128 =
    - Go understands the I literal as an imaginary number
    - Imag looks at the complex number you provide and pulls out the real part or the imaginary part.
* Text Types
  + String: any UTF8 character
    - Immutable
  + Byte
    - Converting strings to the ascii value
* Rune are set with single quotes R := ‘s’

Summary

* Boolean Type
  + Values are true or false
  + Not an alias for other types (e.g. int)
  + Zero value is false. If you initialize a Boolean and do not set true or false it is false
* Numeric Types
  + Integers
  + Signed integers
  + Int type has varying size but min 32 bits
  + 8bit (int8) through 64bit (int64)
  + Unsigned integers
    - Can only be positive
    - 8bit (byte and unit8) through 32bit (uint32)
* Floating point numbers
  + Follow IEEE -754 standard
  + Zero value is 0
  + 32 to 64 but versions
* Literal styles
  + Decimal (3,14)
  + Exponential (13e18 or 2E10)
  + Mixed (13.7e12)
* Arithmetic Operations
  + Add +
  + Subtract -
  + Multiply \*
  + Divide /
  + Remainder %
* Bitwise operations
  + And &
  + Or |
  + Xor ^
  + Not &^
  + Zero value is zero
* Complex numbers
  + Zero value is (0 +0i)
  + 64 and 128 bit versions
  + Built in functions
    - Complex – make complex number from 2 floats
    - Real – get real part as float
    - Imag – get imaginary part as float
* Text Types
  + Strings
  + Utf8
  + Immutable
  + An be concatenated with +
  + Can be converted [] to byte
* Rune
  + Utf-32
  + Alias for int32
  + Special methods normally required to process
  + E.g. strings.Reader #readrune

Constants

Agenda

* Naming Convention
* Typed Constants
* Untyped Constants
* Enumerated constants
* Enumerated expressions
* Name constants the same way we name variables.
* Upper case first letter means it will be exported
* Inner declaration of the constant wins
* Package level constant wins
* Not allowed to change const once they are created. You will get an error message.
  + Example: Const my\_Const
* A const + variable = a variable

Enumerated Constants

* Iota: a counter we can use when we create enumerated constants
* Iota changes its value as it is assigned
* \_ = right only variable
* Raising powers in GO is in the math package.
* Bit shifting: A bit-shift moves each digit in a number's binary representation left or right. Within right-shifts, there are two further divisions: logical right-shift and arithmetic right-shift. A left-shift is represented by the << operator, while a right-shift is represented by the >> operator

Review

* Constants are immutable but can be shadowed
* Constants are replaced by the compiler at compile time.
  + Value must be calculable at compile time
* Named like Variables
  + PascalCase for exported constants
  + camelCase for internal constants
* Typed constants work like immutable variables
  + Can interoperate only with same type
* Enumerated constants
  + Special symbol iot allows related constants to be created easily
  + Iota starts at 0 in each const block and increments by one
  + Watch out of constant values that match zero values for variables.
* Enumerated Expressions
  + Operations that can be determined at compile time are allowed
    - Arithmetic
    - Bitwise operations
    - Bitshifting

Arrays and Slices

* Declare an array by starting with the size of the array and then the type of data the array will store.
* Can only have one type in an array.
  + Example: grade: = [3] int {97,85,93}
  + Example: grade: = […] int {97,85,93}
* Len(array\_name) will print the size of the array
* Identity matrix
* & = address of value/ points to specific value
* Slice: naturally reference types
  + Example: a: = [] int{1,2,3}
* Slice of all elements
  + Example: b:= a[:]
* Slice from 4th element to end
  + Example: b := [3:]
* Slice first 6th elements
  + Example: a[:6]
* Slice the 4th,5th and 6th elements(first number inclusive but second number exclusive
  + Example: [3:6]
* Built in Make function allows you to make slices
  + Example: a:= make([]int, 3,100)
  + 3 arguments are: Type, Size, capacity
* Append (slice name, value) will add a value to the slice
* Remove first element from the slice
  + Example: b:= a[1:]
* Remove last element from the slice
  + Example: a[len(slice\_name)-1]
* Remove element from the middle
  + Example: append(slicename[:2],a[3:]…)
* Capacity

Arrays and Slices Summary

* Arrays:
  + collections of items of the same type
  + fixed size have to state the size
  + Declaration styles
    - A := [3]int{1,2,3}(literal style, declare the number of elements)
    - A :=[…]int{1,2,3}(more robust in design as it updates array size for you as you add items)
    - Var a [3] int (each int starts with zero value)
  + Access via zero based index
    - A := [3]int {1,3,5} // a[1] ==3
  + Len function returns the size of the array
  + Copies refer to different underlying data.
* Slices:
  + Backed by array. Each slice you see under the cover go has an array
  + Creation styles
    - Slice existing array or slice
    - Literal style
    - using make function
      * a := make([]int,10) // create slice with capacity and length == 10
      * a:= make([]int,10,100) // slice with length == 10 and capacity == 100
      * len returns length of the slice
      * cap function returns the length of the underlying array
      * append function to add elements to slice
        + may cause expensive copy operation if underlying array is too small.
      * Copies refer to the same underlying array so if you make a change to one copy it will affect all copies.

**Maps and Structs**

**Maps(dictionary)**

* **What are they?** 
  + Takes a key and value
  + Example:
  + statePopulations := map[string]int {

“California” : 3950,

“Texas” : 4950,

“Florida” : 2069,

**}**

* + variable\_name := map[key\_type]value\_type {

“Key” : value,

“Key” : value,

“Key” : value,

**}**

* + statePopulations := make(map[string]int) {

“Key” : value,

“Key” : value,

“Key” : value,

**}**

* add a key and value: dictionary\_name[“key\_value”] = value
  + statePopulations[“Georgia”]=9087
* delete entry in a map: delete(dictionary\_name,”key”)
  + delete(statePopulations, “Georgia”)
* check if key is found use, ok
  + example: \_,ok := statePopulations[“keyname”]

fmt.Println(ok)

* len(dictionaryname) : Gives the length of the dictionary

**\*\* When copying and pasting into text editor be careful of the quotation marks.**

**Structs**

* **What are they?** 
  + **A collection type**
* **Creating**

**Type Doctor struct {**

**Number int**

**actor\_name string**

**companions []string{**

**}**

**},**

* **Naming conventions: lower case internal, uppercase exported**
* **Capitalize all field names if you want other functions to be able to reach the struct.**
* **Should NOT have underscore in struct name or field name**
* **Anonymous structs cannot be seen anywhere.**

**Embedding (needs review)**

**Tagging (needs review)**

* **Use interfaces when you want to describe common behavior**

**Summary of Maps/Structs**

* Maps
  + Collections of value types that are accessed via keys (Dictionary)
  + Created via literals or via make function
  + Members accessed via [key] syntax
    - myMap[‘key”] = “value”
  + check for presence of item using “value,ok” form of result
  + multiple assignments refer to the same underlying data so any changes will affect all items pointing to the same underlying data.
* Structs
  + Collections of disparate data types that describe a single concept
  + Keyed by named fields
  + Normally created as types, but anonymous structs are allowed
  + Structs are value types
  + No inheritance, but can use composition via embedding
  + Tags can be added to struct fields to describe field

**Control Flow**

**(If and Switch Statements)**

* **If Statements** 
  + Operators
    - || means or
    - && means and
    - ! means not
  + Short circuiting
    - When only some of the parameters are ran and the test fails because it needed to pass all tests
  + If -else and if -else if statements
* **Switch Statements**
  + Simple cases
    - Cannot have over lapping cases will get syntax order.
      * Example case 1,2,3 and case 3,4,5 will get error because 3 used twice
  + Cases with multiple tests
  + Falling through
  + Type Switches: running a switch to determine its type
    - A screenshot of a cell phone

      Description automatically generated
      * Same data type and length for arrays to be alike
      * Use breaks to get out of a switch early

**Summary**

* If statements
  + Initializers and how it allows us to execute a statement that sets up variables to use to run our tests against
  + Comparison Operators (6)
    - Less than <
    - Greater than >
    - Less than or equal to <=
    - Greater than or equal to >=
    - Equivalent ==
    - Not equal !=
  + Logical Operators
    - || or
    - && and
    - ! not
  + Short Circuiting
  + If-else statements
  + If-else if statements
  + Equality and floats
  + Fallthrough- implicit; use if you need to fall through to the next case which executes to the next case no matter what.

**Looping**

* **For statements**
  + Simple loops
    - For I:= 0; i<5; i++ {
      * Fmt.Println(i)
    - Start with an initializer (For I)
    - Some kind of statement with a Boolean result (i<5)
    - Incrementor (i++)
  + Exiting early
  + Looping through collections

Text, letter

Description automatically generated

**Summary**

* There is no “while or Do” Keyword in go for looping only for
* Simple loops
  + For initializer; test; incrementor {}
    - Leaving out the semi colon creates an error message
  + For test {}
  + For {}- runs indefinitely until break
  + Exiting early
    - Break
    - Continue
    - Lables
  + Looping over collections
    - Arrays, slices, maps, strings,channels
    - For k,v := range collection {}

Control Flow

(Defer, Panic and Recovery)

* **Defer**
  + - * Keyword
      * A defer statement defers the execution of a function until the surrounding function returns.
      * Last function deferred will be the first to be executed. (lifo)
* **Panic**
  + - * Does not have exceptions because a lot of cases that are exceptional in other languages isn’t exceptional in go.
      * built-in function that stops the ordinary flow of control and begins *panicking*.
      * Panic is fatal when the panic takes longer than the run time.
      * Execute function then defer then panic
* **Recovery**
  + - * a built-in function that regains control of a panicking goroutine. Recover is only useful inside deferred functions.

**Summary**

* **Defer**
  + Used to delay execution until function ends
  + Useful to group “open” and “close” functions together
    - Be careful in loops (can cause memory issues)
  + Run in LIFO (last-in, first out order)
  + Arguments evaluated at time defer is executed, not at time of call function execution.
* **Panic**
  + Occur when program cannot continue at all
  + Don’t use when file can’t open, unless critical
  + Use for unrecoverable events- cannot obtain TCP port for web server
  + Function will stop executing
    - Deferred functions will still fire
  + If nothing handles panic, program will exit
* **Recover**
  + Used to recover from panics
  + Only useful in deferred functions
  + Current function will not attempt to continue, but higher functions in call stack will.

**Pointers**

* Creating pointers

Text, letter

Description automatically generated

* + Copies data stored in a and assigns it to b. **NOT** pointing to the same memory

Text, letter

Description automatically generated

* + Var b \*int = &a creates the pointer to a. &a means the address of
  + Dereferencing operator \*b = shows the data the pointer is pointing at.
* The new function working with nill

Text, letter

Description automatically generated

* + A pointer you do not initialize or give a value will point to nil
* Types with internal pointers
  + Maps have a pointer to the underlying data

Text

Description automatically generated

Summary

* Creating pointers
  + Pointer types use an asterisk (\*) as a prefix to type pointed to
    - \*int is a pointer to an integer
  + Use the address of operator (&) to get address of variable
* Dereferencing pointers
  + Dereference a pointer by preceding an asterisk (\*)
  + Complex types (e.g. structs) are automatically dereferenced
* Create pointers to objects
  + Can use the address of operator (&) if value type already exists
    - ms := myStruct {foo:42}
    - p := &ms
  + Use addressof operator before initializer
    - &myStruct{foo:42
  + Use the **new** keyword
  + Can’t initialize fields at the same time
* Types with internal pointers
  + All assignment operations in Go are copies
  + Slices and maps contain internal pointers, so copies point to the same underlying data

Functions

* Passing in a pointer is more efficient than passing in an object.
* Passing in pointers is a performance benefit
* Maps and slices act as if you are passing in pointers
* Graphical user interface, text, application, email

  Description automatically generated
  + It is bad practice to use panic when there is an error in your function. You should pass in what you want to return and also an error in the return statement.
  + Return value, nil if nothing went wrong
* Fmt.Errorf(“Value”)
* Anonymous functions are functions that are not named usually used when using functions as a type.

Summary

* Basic syntax
  + Example: func foo() {

…….

} (on its own line after the final line of code for the function)

* + Func is a keyword
  + Foo is the function name
* Parameters
  + Allow us to pass data into the function
  + Comma delimited list of variables and types
    - Example Func foo(bar string, baz int)
  + Parameters of same type list type once
    - Example Func foo(bar, baz int)
  + When pointers are passed in, the function can change the value in the caller.
    - This is always true for data of slices and maps
  + Use variadic parameters to send list of same types in
    - Must be last parameter
    - Received as a slice
    - Fun foo (bar string, baz ….int)
* Return Values
  + Single return values just list type
    - Func foo() int
* Multiple return value list types surrounded by parentheses
  + Func foo() (int,error)
  + The (result type,error)paradigm is a very common idiom
* Can use named return values
  + Initializes returned variable
  + Return using return keyword on its own
* Can return addresses of local variables
  + Automatically promoted from local memory(stack) to shared memory (heap)
* Anonymous functions
  + Functions font have names if they are
    - Immediately invoked
    - Assigned to a variable or passed as an argument to a function
* Functions as types
  + Can assign functions to variables or use as arguments and return values in functions.
  + Type signature is like function signature, with no parameter names
* Methods
  + Functions that executes in context of a type
  + Format
    - Func (g greeter) greet() {

……

}

Interfaces (Needs Review)

* Basics
  + An interface type is defined as a set of method signatures.
  + Start by typing type (the interface type) interface
  + A value of interface type can hold any value that implements those methods.
* Composing interfaces
* Type conversion
  + The empty interface
  + Type switches
* Implementing with values vs. pointers
* Best Practices
  + Use many, small interfaces
  + Many small interfaces is preferable and faster
  + Single method interfaces are some of the most power and flexible
    - Io.writer, io.Reader
  + Don’t export interfaces for types that will be consumed.
  + Do export interfaces for types that will be used by package.
  + Design functions and methods to receive interfaces whenever possible

Summary

* Basics
  + Type Writer interface {
    - Write(byte)(int,error)

}

* Type ConsoleWriter struct {}
* Func(cw ConsoleWriter) Write(data byte)(int,error){
  + N,err := fmt.Println(string(data))
  + Return n,err

}

* Composing interfaces
  + Type Writer interface {
    - Write(byte)(int,error)

}

* Type Closer interface {
  + Close() error
  + Type WriterClose interface{
    - Writer
    - Closer

}

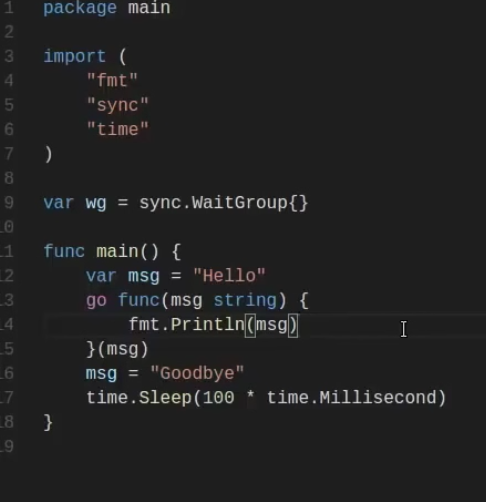
GoRoutines

* Creating Goroutines

Text

Description automatically generated

* Synchronizations
  + WaitGroups



* Wait Groups synchronizes multiple GoRoutines
  + Mutexes
    - Locked has to wait until the mutex is unlocked.
* Parallelism
* Best Practices
  + Don’t create goroutines in libraries
    - Let consumer control concurrency
  + When creating a goroutine, know how it will end
    - Avoids subtle memory leaks
  + Check for race conditions at compile time
    - Go run -race filename

Summary

* Creating goroutines
  + Use go keyword in front of function call
* When using anonymous functions, pass data as local variables
* Synchronization
  + Use sync.WaitGroup to wait for groups of goroutines to complete
  + Use sync.mutex and sync.RWMutex to protect data access
* Parallelism
  + By default, go will use CPU threads equal to available cores
  + Change with runtime.GOMAXPROCS
  + More Threads can increase performance but too many can slow it down.

Channels

* Channel Basics
  + Channels are a typed conduit through which you can send and receive values with the channel operator
  + Created with the built-in make function
  + Example ch: = make (chan int)
* Restricting data flow
* Buffered Channels
* Closing Channels
* For range loops with channels
* Select statements

**To zip a go file**

GOOS=linux GOARCH=amd64 go build -o main main.go

zip main.zip main

**to debug for golang locally**

GOARCH=amd64 GOOS=linux go build -gcflags='-N -l' -o <output path> <path to code directory>