**Golang**

Last Edited: October 2021

Go is a fast (C# level fast) language. It’s open source, clean and focuses on concurrency.

1. To setup go, install the client from their site and that’s it.
2. Every Go program is made up of packages. Where ‘package main’ with a function ‘main’ is the entry point.
3. Sample Go program:

In abc.go file,

package main

import (

"fmt"

"math/rand"

)

func main() {

fmt.Println("My favorite number is", rand.Intn(10))

}

Here import receives the path (in the pkg.go.dev ) of the packages, the package name (like ‘main’) is by convention same as the last value of path. Like ‘rand’ and ‘fmt’ here.

This is an alternate usage of import command (called factored import statement), it can also be used like ‘import “fmt”’ then ‘import “math/rand”’

1. Types:

bool

string

int int8 int16 int32 int64

uint uint8 uint16 uint32 uint64 uintptr

byte // alias for uint8

rune // alias for int32

// represents a Unicode code point

float32 float64

complex64 complex128

* 1. The default values for uninitialized variables are:

0 for numeric types,

“” for string

false for bools

* 1. Type Conversion is explicitly defined, unlike C it is not implicit.

var x int = 1

var y float = 1.0

x=float(y)

to convert it

* 1. Pointers: \*T is a pointer, where T is the type. Go has pointers but not pointer arithmetic.

&T generates the pointer to the value.

For example:

var i, j int= 1,2

p:= &i // generate the pointer for i and store it in p, p’s type is \*int

fmt.Println(\*p) //prints value of p, this is called dereferencing/indirecting value of p.

\*p=10 //stores 10 in value referenced by p, so i and \*p are now 10 instead of 1.

* 1. Structs: A collection of fields,

For example:

type Hello struct {

X int

Y int

}

v:= Hello{1,2} // stores 1 in X and 2 in Y

v.X=4

to generate instance and modify value of a field.

We can use pointers to store instances too,

p:= &v

(\*p).X=3

p.Y= 10

Both ways are valid to access values inside the struct referenced by a pointer.

More ways to initialize an instance

var (

v1 = Hello{1, 2} // has type Hello

v2 = Hello {X: 1} // Y:0 is implicit

v3 = Hello {} // X:0 and Y:0

p = & Hello {1, 2} // has type \* Hello

)

The ‘type’ keyword used in declaration declares that this is our type.

* 1. Arrays: They are 0 indexed.

[n]T where n is size and T is type

For example:

var x [20]string

or

y:= [20]string{“a”, “b”} //stores “” in the rest of the elements

Then access it like normal arrays. These are fixed size arrays and the size is part of their type.

* 1. Slices: Dynamically sized arrays.

a := [5]int{1,2,3,4,5}

var s [ ]int= a[1:3]

slices the array a and returns index 1 to 3

* + 1. Slice Syntax:

t[l:h] where t is the variable, l is the lower bound and h is the upper bound (exclusive) in the slicing. Default value for lower bound is 0 and length for the upper bound so a[:] is the same as a[0:10]

* + 1. Slices don’t copy, they reference the elements from the base array. So if the element changes in a slice then it does so in the base array too, and vice versa.
    2. Slices can be used to generate dynamically (dynamic as in runtime constant but compile time dynamic) sized arrays too,

b:= [ ]int{3,4,5}

the size of b is 3.

s:=[ ]struct{

i, a = bool

} {

{false,true },

{true, true}

}

the size of s is 2

* + 1. Length and Capacity: Length of a slice (len(obj)) is the size of the slice while capacity (cap(obj)) is the size of the base array. The latter is fixed while the former can be extended (up till the size of the base array, i.e., the len can only be <= cap) or reduced by re-slicing the sliced array or the base array.
    2. Nil Slice: An empty slice,

var a[ ]int

has len and cap 0 and a==nil is true.

* + 1. Slice of Slices: To do so,

board := [ ][ ]string{

[ ]string{"\_", "\_", "\_"},

[ ]string{"\_", "\_", "\_"},

[ ]string{"\_", "\_", "\_"},

}

* + 1. Append: This function takes a slice object and adds element to it, it increases the len and if the len has reached the cap then it grows the slice, hence it works on nil slices too.

var s [ ]int

s=append(s,0, 5,2) //appends 0 5 and 2 to s and increases len and cap to 3

* 1. Truly Dynamically sized arrays: We use the make function for this, the function allocates a zeroed array of given size (size can be given at runtime) and returns a slice that referes to that array

a := make([ ]int, 0, 5) //len 0 and cap 5

* 1. Maps: map[T]K where T is type of Key and K is type of value

For example:

var m map[string]int

or

m := make(map[string]int){

“yo”:2,

“yp”:3,

}

m[“aa”]= 2

zero value map is == nil

For user-defined types like structs,

var m map[string]Hello = {

“yo”: Hello{1,2},

“aa”:Hello{2,3},

}

or

var m map[string]Hello = {

“yo”: {1,2},

“aa”: {2,3},

}

If the top-level type is just a name then it can be omitted.

* + 1. Methods on maps:

insert/update

m[key]=value

get value

a := m[key]

delete key

delete(m,key)

test and retrieve value

elem, ok := m[key]

ok gets a bool which is true if key existed and false otherwise.

elem gets the value if it exists or gets the default value of type of the value the map has.

* 1. Function: Functions are types too. Explained in it’s own block.
  2. Method: Go doesn’t have classes, only structs. We can define ‘methods’ on types, these are like extension methods from c#.

Syntax:

func (v K) <name>(<params>) T {

… //body gets a copy of v instance

}

It cannot modify value of the original instance stored in ‘v’

For Example:

type Vertex struct {

X, Y float64

}

func (v Vertex) Abs() float64 {

return math.Sqrt(v.X\*v.X + v.Y\*v.Y)

}

func main() {

v := Vertex{3, 4}

fmt.Println(v.Abs())

}

The ‘v’ variable gets the instance to work with.

‘Abs()’ method is just a function so,

func Abs(v vertex) float64 {…} will be the same thing, we will call it with Abs(v)

* + 1. K type in the syntax can be any type that is declared in the same package only. So int, float64 etc. cannot be used. However,

type MyFloat float64

func (f MyFloat) Abs() float64 {…} works.

* + 1. K can be pointer type as well, this means it can modify value in the instance. It is called a pointer receiver.

func (v Vertex) Scale(f float64) {

v.X = 1

v.Y = 1

}

won’t result in anything and the instance doesn’t get modified.

func (v \*Vertex) Scale(f float64) {

v.X = 1

v.Y = 1

}

works.

We should call (&v).Scale(10) instead of v.Scale(10) when pointer receiver is used but Go automatically interprets the statement as the former. The same is not true for normal functions,

For example

func Scale(v \*Vertex, f float64) {

…

}

cannot be called with Scale(v,10). It needs to be called with Scale(&v,10)

The same is also true when the variable is stored in a pointer,

func (v Vertex) Scale(f float64) {

v.X = 1

v.Y = 1

}

…

p := &Vertex{3,4}

p.Scale(10) would work as it is automatically inferred as (\*p).Scale(10)

but Scale(p,10) wouldn’t.

Scale(\*p,10) would.

* + 1. It is recommended to use pointer recievers since it avoids copying and is faster.
  1. Interface: There are no classes but interfaces and their implicit implementations can exist. It works in a dependency injection sort of way. An interface in Go can only have method signatures.

To do so,

type MyInterface interface {

Abs() float64

}

func (v Vertex) Abs() float64 {

…

}

func main() {

var a MyInterface

v := Vertex{1,,2}

a = v

a.Abs() //will invoke the Abs of Vertex

}

a=v only works because Abs() is declared as a method on Vertex. If it were declared for any other struct or even \*Vertex it would not work. For \*Vertex we would use a= &v instead.

* + 1. Underlying type of an interface can be nil.

var a MyInterface

var b \*Vertex

a=&b //will work

a.Abs() //will still call Vertex’s abs but ‘v’ will be == nil

* + 1. Calling methods on a nil interface is a runtime error

var a MyInterface

a.Abs() //error

* + 1. Empty Interface: An interface that specifies 0 methods is called so.

The use of this interface is that it can hold a value of any type, that’s because every type implicitly implements an empty method.

var a interface{}

printer(a)

func printer(i interface{} ) {

fmt.Printf("(%v, %T)\n", i, i)

}

will print (<nil>,<nil>)

a= 2

printer(a)

will print (2,int)

and so on.

* + 1. Type Assertion: This allows acces to the interface’s value.

Syntax

t := i.(T)

where i is the interface with a value and T is the type of that value. If T is the same as the type inside it then value is returned to the ‘t’, but if it’s not then ‘panic’ exception is thrown.

Alternatively,

t, ok := i.(T)

can be used. This never throws ‘panic’, ‘t’ will get a value if it’s type T and ‘ok’ will get ‘true’. If it’s not of type T then ‘t’ will get default value of type T and ‘ok’ will be false.

For example:

var a interface { } = ‘hello’

b:=a.(string) //will store ‘hello’ in b.

* + 1. Type Switch: A regular switch statement but the cases are switching on ‘type’ instead of values.

To use it,

switch v:= i.(type) {

case int: …//v has value of type int.

case string: …

default: … // v has type of interface

}

* + 1. ‘Stringer’ interface in fmt package defines ‘String’ method.

This means we can define String() methods for our structs.

func (v Vertex) String() string {

return “yolo”;

}

a := Vertex {3,2}

fmt.Println(a) //will print yolo. This will have to be changed when pointer // receiver is used

a.String() //will print yolo

* + 1. ‘error’ interface in fmt package defines ‘Error’ method.

The signature is same as String() method.

Errors in Go are expressed with ‘error’ values. If an ‘error’ value is nil then that means success otherwise it means failure and ‘Error()’ method can be used to display the error.

For example:

type Vertex struct {

X,Y int

}

func (f Vertex) Error() string {

return "yonoo"

}

func run() error{

return Vertex{1,2}

}

func main() {

a := run()

fmt.Println(a == nil)

fmt.Println(a)

}

prints “false” then “yonoo”

In this code, ‘run’ returns an ‘error’ value hence ‘a’ isn’t nil and when we print ‘a’ we run the method we used to implement Error().

1. Declaration:
   1. Short variable declaration:

Variables can be declared and used in the same line, to do so

<varname> := <value>

like,

a := 5

The type is inferred automatically.

This works only inside functions, the ‘:=’ construct isn’t available in the package scope (outside functions)

* 1. We can use multiple declarations:

like,

a,b :=5

* 1. Normal Declaration: Works in every scope.  
     To declare a variable without defining,

var <varname> <type>

like

var b int

or

var a, b, c int

* 1. Variables can also be explicitly initialized using initializer,

To do so,

var a, b, c int = 1, 2, 3

or

the type can be omitted

var a, c = 1, 2

* 1. Factored Variable Declaration:

var (

ToBe bool = false

MaxInt uint64 = 1<<64 - 1

z complex128 = cmplx.Sqrt(-5 + 12i)

)

* 1. Constant: Using const in place of var to declare constants, they don’t support short variable declaration though.

1. Function:

Syntax:

func <name>(<param name then their type>) <return type>{…}

For example:

func add(x int, y int) int {

return x + y

}

can also be written like

func add(x,y int) int {…}

* 1. They can return any number of results:

func swap(x, y string) (string, string) {

return y, x

}

func main() {

a, b := swap("hello", "world")

fmt.Println(a, b)

}

* 1. Named Return Values: Return type can be given a name, if done so then the names act like declarations.

func split(sum int) (x, y int) {

x = sum \* 4 / 9

y = sum - x

return x,y

}

* 1. Naked Return: If using NRV then this can be used instead of normal return,

func split(sum int) (x, y int) {

x = sum \* 4 / 9

y = sum - x

return

}

The return returns ‘x,y’ here.

* 1. Exported names: A function is exported (available to other files) only if it’s name begins with a capital letter.
  2. Functions are types too, so they can be passed around.

hypot := func(x, y float64) float64 {

return math.Sqrt(x\*x + y\*y)

}

fmt.Println(hypot(5, 12))

We can pass this func to other funcs too,

func xyz (fn func(float64, float64) float64) int {

… //use fn

}

* 1. Functions can be closures, a closure is a function value that references variables from outside it’s body. With functions as types and closures some complex scenarios can be created, like

func adder() func(int) int {

sum := 0

return func(x int) int {

sum += x

return sum

}

}

func main() {

pos, neg := adder(), adder()

for i := 0; i < 10; i++ {

fmt.Println(

pos(i),

neg(-2\*i),

)

}

}

Here pos and neg don’t get adder function but the return value of adder which is a function in itself. But since the returned function is working on ‘sum’ variable which was retrieved from outside its scope the variable’s reference is used.

1. Print: 3 different types of print available in fmt package.

fmt.Println(“something”);

1. Loop: Go only has the standard for loop.

For example:

for a:=0; a<10; a++ {…}

The 3 sections are called, ‘init’ statement, ‘condition’ statement and ‘post’ statement.

* 1. initand post statement are optional,

a:=0

for ; a<10 ; {…}

or

for a<10 {…}

even Condition statement is optional which leads to an infinite loop,

for {…}

* 1. Range: Range form of for loop is used to iterate over a slice or map,

var pp= [ ]int {1,2,3}

for i, v := range pp {

…

}

Here, i is index and v is a copy of the element at that index and loop is iterating over pp.

We can choose to omit the index or the value by,

for i := range pp {…}

or

for i,\_ :=range pp {…}

Here i will get the index and value is discarded

for \_, v:= range pp {…}

Here index will be discarded.

1. Condition Blocks:
   1. If else:

if a < b {…}

else if a<c {…}

else {…}

* 1. Just like C++ we can declare variable in if block, scope visible to if and the elses

if v:= 2; v<4 {…}

else {…}

* 1. Switch: Just like C++ although unlike C++ the control automatically breaks after encountering it’s case.

For example:

switch a:= “abc”; a {

case “a”: fmt.Println(“yolooo”)

case “abc”: fmt.Println(“yo”)

default: …

}

will only print “yo”

* 1. Switch with no condition: An alternative to long if else chains,

a:=2

switch {

case a < 1: …

case a < 3: …

case a+5 < 3: …

}

1. Defer: defers the execution of a function until the surrounding function returns.

For example:

func main(){

a:=2

defer fmt.Println(“World %d”, a)

fmt.Println(“Hello”)

}

prints “Hello World 2”,

the arguments are processed immediately but the function itself isn’t executed until the main() returns.

* 1. Deferred functions are pushed onto a stack, i.e., they follow a LIFO execution.

For example:

defer abc()

defer bbb()

ccc()

runs ccc() then starts working on the deferred func stack, so it executes bbb() then abc()

1. Readers: The ‘io’ package contains a ‘Read’ method.

import (

"fmt"

"io"

"strings"

)

r := strings.NewReader("Hello, Reader!")

b := make([]byte, 8)

for {

n, err := r.Read(b)

fmt.Printf("n = %v err = %v b = %v\n", n, err, b)

fmt.Printf("b[:n] = %q\n", b[:n])

if err == io.EOF {

break

}

}

prints

n = 8 err = <nil> b = [72 101 108 108 111 44 32 82]

b[:n] = "Hello, R"

n = 6 err = <nil> b = [101 97 100 101 114 33 32 82]

b[:n] = "eader!"

n = 0 err = EOF b = [101 97 100 101 114 33 32 82]

b[:n] = "

1. The ‘image’ package contains functions for defining an image.
2. Goroutine: It’s a lightweight thread managed by the Go runtime.

It’s as simple as,

go <func>(<params>)

and the func will run in another thread

For example

func say(s string) {

…

}

go say(“hello”)

will start another thread and run say on it. The params and the function is evaluated in the current thread but the execution occurs on another thread.

For example:

func say(s string, times time.Duration) {

time.Sleep(times)

fmt.Println(s)

}

func main() {

go say("world", 8 \* time.Second)

go say("hello", 5 \* time.Second)

say("ni", 10 \* time.Second)

}

prints

“hello”

“world”

“ni”

The main thread does not wait for the other threads to complete, if say(“ni”) has a lower sleep time than the threads then it will finish up and close the process.

* 1. Channels: Channels are a ‘typed’ conduit used to send and receive values between goroutines.

We send values with ch <- v and receive values with v:= <-ch. (data flows in dirn of arrow)

where v is the value and ch is the channel.

Syntax:

chan T

where T is type of data that will pass between the channel

For example:

func sum(s int, c chan int) {

c <- s

}

func main() {

s := 2

c := make(chan int)

go sum(s, c)

s=3

go sum(s, c)

x, y := <-c, <-c // receive from c

fmt.Println(x, y, x+y)

}

prints “3 2 5”

Here, the values are added into the channel like a stack and comes out LIFO.

* 1. Buffered Channels: In normal channels if same thread tries to add values to the channel then it will give exception ‘all goroutines are asleep’ as the channel isn’t ready to pick the values up. We can use buffered channels to let the goroutines be ready.

To do so,

ch := make(chan int, 3) //creates a channel and gives a buffer of 3

ch <- 1

ch <- 3

ch <- 5

//then they can be retrieved from the channel.

Buffered channels don’t follow LIFO but FIFO.

* 1. Range and Close: For range can be used to iterate over elements being received and close(<ch instance>) can be used to close a channel. Closing isn’t necessary and should only be used when needed such as in when using range.

For example:

func fibonacci(n int, c chan int) {

x, y := 0, 1

for i := 0; i < n; i++ {

c <- x

x, y = y, x+y

}

close(c)

}

func main() {

c := make(chan int, 10)

go fibonacci(cap(c), c)

for i := range c {

fmt.Println(i)

}

}

Only a sender should close the channel, never the receiver as sending on a closed channel gives ‘panic’ exception.

* 1. Select: It lets a goroutine wait on multiple communication operations. It’s like switch but for goroutines. It is a blocking operation and blocks the thread until one of the case or default case is executed.

Syntax:

select{

case c1 := <- ch :

…

default: fmt.Println(“None started”)

}

For example:

func portal1(channel1 chan string) {

time.Sleep(3\*time.Second)

channel1 <- "Welcome to channel 1"

}

func portal2(channel2 chan string) {

time.Sleep(9\*time.Second)

channel2 <- "Welcome to channel 2"

}

func main(){

R1:= make(chan string)

R2:= make(chan string)

go portal1(R1)

go portal2(R2)

select{

case op1:= <- R1:

fmt.Println(op1)

case op2:= <- R2:

fmt.Println(op2)

}

}

prints “Welcome to channel1” and then exits.

An empty select blocks the thread forever.

If multiple channels are ready at the same time then it selects any case at random.

The default case is ran if no case is ready. It runs default case and exits, to use default we usually use an infinite for loop over select and then use a ‘return’ in a the desired case while the default case keeps on looping.

* 1. Mutual Exclusion: Mutex, this is used to lock and unlock an area of code so that only 1 thread can access it at any given time. The rest of the threads wait for it to finishand unlock the mutex.