→ Welcome : 5
→ Package, var, func: ▷
→ Flow control: if-else | sw | defer : 14 } 22/8/22

NOTES

22/08/22

fmt → Format package.

```
package main
import (
    "fmt"
    "time"  |  "math/rand"
)
func main(){
    fmt.Print("Hi\n")
    fmt.Println("T...", time.Now())
}
```

→ Time : time.Now()
→ math/rand : rand.Intn(n)
→ math : Sqrt(n)
→ Exported name begins with capital letter.
→ Data type comes after variable name.
→ a [3] int
→ p *int

```
func f1 (argc int, argv []string) int
```
Here "int" at the end is return type.

→ Closure :

```
sum := func(a, int b) int
    { return a+b } (3, 4)
```

** → Function to isolate global variable.

```
package main
import "fmt"
func newCounter() func() int {
    GFG := 0
    return func () int {
        GFG += 1
        return GFG
    }
}
int func main() {
    counter := newCounter()
    fmt.Println (counter ())
}
```

→ In above, if we dont do this, then GFG is a global variable and it can be accessed by anyone.

→ Pointers

```
var a[] int
x = a[1]

var p *int
x = *p
```

→ Function parameters with same data types can be omitted from all but last.

```
x int, y int <=> x, y int
```

→
```
func swap (x, y string) (string, string)
{   return  y, x }
func main() {
    a, b := swap ("hello", "world")
    fmt.Println (a, b)
}
```

? → Naked / Named return

```
package main
import "fmt"
func split (sum int) (x, y int) {
    x = sum * 4/9
    y = sum - x
    return
}
func main () {
    fmt. Println ( split (17))
}
```

? → 7 10

? Naked return statements should be used for short functions.

→ var - for declaring list of variables

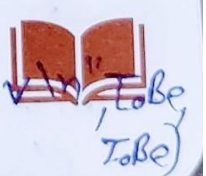
```
var i, j = 1, 2   // Valid
var i, j int = 1, 2 // Valid
```

→ Short variable declaration

```
i, j := 1, 2
```

This can be used inside a function

```
var (
    ToBe bool = false
    MaxInt uint64 = 1<<64-1
) func main() {
fmt.Println("Type: %T Val: %v", ToBe, ToBe)
}
```



# NOTES

16+144-4

(44+12

$\frac{156}{2}$ =78

import "math/cmplx"

→ complex128 = cmplx.Sqrt(+5+12i)

→ 2+3i

→ Zero values

Variables declared w/o initial val are given their zero val.

0 → Numeric

false Boolean

"" (Empty string) → String

→ T(v) converts value of v to Type T.

→ Constants are declared with const keyword.

Constants CANNOT be declared using ":=" syntax

NOTES

→ An int can store at-max 64-bit integer.

→ For Loop

```
for i := 1; i <= 10 ; i++{
    sum += i
}
```

Paranthesis are not these. Braces are always required.

→ Looping

Only loop in Go is "for"

```
for sum < 1000 { } ==while (sum
for ; sum < 1000; {}
    for {} // infinite loop
```

→ if v := Pow(x, n); v < lim{}

// Scope of v is inside braces.

→ Short variables declared inside if are available in else block as well.

→ Switch: Cases by default have break statements.
→ Also switch cases need not be constants, & values involved need not be integers.

23/8/22

→ Pointers has zero value nil.
→ Go has no pointer arithmetic.
→ Struct: Vertex does API handling REST.

```
type Vertex struct {
    X int
    Y int
}
```

# NOTES

→ Interface / Channels / Goroutine / Array ) Slices / Defer

---

→ Dereferencing is default
(*p).x == p.x

```
var (
    v1 = Vertex{1, 2}
    v2 = Vertex{X: 1} // Y = 0
    v3 = Vertex{} // X = Y = 0
    p = &Vertex{1, 2}
```

fmt.Println(p) ⇒ &{1 2}
fmt.Println(v1) ⇒ {1 2}

→ Arrays

```
primes := [6]int{2, 3, 4, 5, 6, 7}
fmt.Println(primes)
```

⇒ [2 3 4 5 6 7]

→ Slices

```
primes := [6]int{2,3,4,5,6,7}
s := primes[1:4]
fmt.Println(s)
```

⇒ [3 4 5]

First is inclusive. Last is exclusive.

* For out of Bounds

Error: invalid argument: index X out of bounds [0:size+1]

s := [0, size] ⇒ Entire array

→ Slice do not store any data. It is only reference. So changing data, does change org array.

# NOTES

- Slice literal
  Array literal w/o length

  For eg:
  [3] bool { true, true, false } //Arra
  [] bool { true, true, false } Slice lite
  s := [] struct { i int, b bool } {{2, true}, {3, false}}

- Slice defaults

  0 - Size default.

  a[0:10] == a[:10] == a[0:]
  = a[:]

Capacity : Potential len

s := []int{1, 2, 3}
s = s[:0] // Zero len / 3 cap

fmt.Print(cap(s), len(s)) ✓

Slice bounds Out of Range

→ Nil slice
len(s) = cap(s) = 0.
No underlying array.
→ Creating slice with make
a := make([]int, 5)
// len(a) = 5, cap = 5

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

→ Slice of slice

```
board := [][]string{
  []string{"_", "_", "_"},
  []string{"_", "_", "_"},}

board[0][0] = "X"
```

NOTES

→ Append slice

s = append (s, 10) //
Appends 10 to s.

→ Range

```
var pow = []int{1,2,4,8,16}

func main(){
  for i, v := range pow{
  fmt.Printf("2^%d = %d\n"
             , i, v)
}
```

2^0 = 1
2^1 = 2
:
2^4 = 16

9.9 Range Continue / Struct ... 17/27

Excercises 18/27 / 23/27 Left

Multi struct Multi map 9.9

strings.Field(s) => List of words separated by ~~commas~~ spaces.
?? Func for other delimeters

→ Use var for declaring variable outside a func.

** → Go does not have CLASSES

→ Methods

Function with special arg b/w func keyword & func_name

```
func (v Vertex) func1(a int) fl{}
    v.func1(3)
```

→ You can declare methods on non-struct types too.

```
type MF float64
f: MF(-math.Sqrt2
```

Struct | Array | Slice | Interface | Thread

# NOTES

→ Methods will interpret v as ~~v~~ &v on its own. While for pointer receiver u need to send &v.

→ Method with Pointer receiver is also invoked using Pointer

```
func (v Vertex) f1 (n int) float64 }
v1 := &p | v2 Vertex
v1.f1(2)  // ok.
v2.f1(2)  // ok
```

→ Excercise left 18/26 - Stringer