[A very powerful language for data Science]

Go lang important points

① You can't Re-declare the same variable in the main function scope (same scope). But you can assign multiple value into one declared variable.

for ex:-

```
func main() {
    var i int = 42
    i := 13
}
```
→ This is wrong. ✗

```
func main() {
    var i int = 42
    i = 13
}
```
✓

② Go takes FLOAT64 as a default data type for any decimal variable value.

for ex:-

```
func main() {
    i := 23.0
    fmt.printf("%v, %T", i, i)
}
```

Output:-

23.0, float64

③ What is shadowing in Go?

Scenario: If you declare a variable twice in your application once at the package level and once inside of the main func. what happen in that case, the variable in the innermost scope takes precedence. So this is called shadowing.

for ex:-

```
Package main
import ("fmt")
var i int = 27
func main() {
    var i int = 42
    fmt.Println(i)
}
```

Output -

42.

Date: / / /

④ We can't leave any declared variable without used on go application.

for ex:-

```
func main() {
    var i int = 42        → In this case you will get a compile time error.
        j := 27
    fmt.Println(i)        j declared and not used.
}
```

⑤ Lower case ~~in the~~ variable ~~name~~ means this is scoped to that package.

Upper case variable → ~~Exposed~~ The Go-compiler to Exposed the varible to the outside world.

3 levels of visibility

1) Var on package level + lower case first letter → Scoped to the Package

(All are the src files that are in the same pkg have access to that variable)

2) upper case + package level → Exported from the package and globally visible.
first letter

3) Block scope → Never visible out of the block scope.

for ex:-

```
func main() {
    var i int = 42
        j := 13
    fmt.Println(i)
    fmt.Println(j)
}
```

→ Blocked to that Scope

NOTE:
Also, there is no private Scope, so you can't scope a variable to the src itself

⑥ Length of the variable name define the life of the variable.

[Naming convention as per GO] → Pascal for (upper case first letter) Camel case (lower case first letter)

A local variable should be having small names like i, j, etc.

globally accessible variable should be having proper variable name.

Date: / /

(7) Best practises :- use acronyms in upper case letter in Go. The reason is readability.

for ex :-

```
func main() {
    Var theURL string = "https://google.com"
    fmt.Println(theURL)
}
```

(8) How do you do the data type Conversion in Go

for ex :-

```
func main() {
    Var i int = 42
    fmt.Printf("%v, %T\n", i, i)
    Var j float32
    j = float32(i)
    fmt.Printf("%v, %T\n", j, j)
}
```

output :-
42, int
42, float32

This float32 data type works as a function out there

int datatype to float32 dT conversion

(9) 3 diff ways to declare the variable in Go

for ex :-

→ Var foo int (useful of you need to declare the var in a diff scope then you actualy gonna initialize it)

→ var foo int = 42 (use when compiler is not taking right data type)

→ foo := 42 (most of times)

(10) We can use strconv package while performing the int conversion in to str.

(11) (Primitives)

We can store 3 types of information in Go.

- Boolean Type
- Text Types
- Numeric Type
  - → integers
  - → floating point
  - → complex NO

NOTE:-

* Booleans are not an alias of any type (eg. int) * ie you can't convert back and forth into int like that"

Date / /

(12) Case study

Ex:-

```
func main() {
    var n bool
    fmt.Printf("%v, %T", n, n)
}
```

Other languages → In some languages that will be uninitialize memory and we will have no control over what printed out.

Analysis

Go → In Go, every time you initializ a variable it has a zero value, the zero value for boolean is value False

(13) Zero value for all numeric types is going to be 0 OR the equivalent of zero for that numaric type.

(14) Integer type numeric number.

Signed

int8	-128	-127
int16	-32768	-32767
int32	-2 147 483 648	-2 147 483 647
int64	-9 223 372 036 854 775 808	-9 223 372 036 854 775 807

Unsigned integer

for ex:-

```
func main() {
    var n uint16 = 42
    fmt.Printf("%v, %T", n, n)
}
```

output: 42, uint16

uint8	0-255
uint16	0-65535
uint32	0-4 294 967 295

Date: / /

(15)
```
func main() {
    a := 10
    b := 3
    fmt.Println(a/b)
}
```

Output
3

int / int = int (can't float point No)
→ So the type can't change while performing this operation

(16) In Go, we are not allowed to add different types of integers

for ex:-
```
func main() {
    var a int = 10
    var b int8 = 3
    fmt.Println(a+b)
}
```
You will get a compiler error, invalid operation: a + b (mismatched types int and int8)

⇒ In order to work you have to do the type conversion

for ex:-
```
func main() {
    var a int = 10
    var b int8 = 3
    fmt.Println(a + int(b))
}
```
Output
13

(17) Bitwise operator

& (AND) multiply the bits (1010 & 0011) = 0010 = 2

| (OR) One OR the other set (1010 | 0011) = 1011 = 11

^ (Exclusive OR / XOR) Either one has the bit set or the other does but not both (1010 ^ 0011) = 1001 = 9

&^ (AND NOT / Bit clear) → opposite of OR

(1010 &^ 0011)
0100 = 4

It is going to set only if neither one of no have the bit set

* NOTE:- This is called Bitwise NOT

^0101 (same as 1111 ^ 0101) = 1010

Date: / / /

(18) Bit shifting Operator:- "<<" is used for "times 2" and >> is used for "divided by 2" - and the number after it is how many times

for ex:-

n << x.

⇒ n times 2, x times

y >> z

⇒ y divided by 2, z times

1<<5 → 1 times 2, 5 times → (2)¹ x (2)¹ x (2)¹ x (2)¹ x (2)¹ → 32

32>>5 → 32 divided by 2, 5 times 32/(2)¹ x (2)¹ x (2) x 2 x 2 → 1

8<<3 → 8 times 2, 3 times → (2)^3 * 2 times 3 16 x 2 x 2 = 64

8>>3 → 8 divided 2, 3 times 8/(2)3 = 1

16<<4 → 16 times 2, 4 times → (16 x (2)⁴) (2)⁴ x (2)¹ times 4 → 32 x 2 x 2 x 2 → 256

16>>4 → 16 divided by 2, 4 times → 16/(2)4 = 1.

(19) Go store Floating point number-

float32 ±1.18E-38 - ±3.4E38

float64 ±2.23E-308 - ±1.8E308

Floating point follow IEEE-754 Standard.

zero value is 0

Ex:- n := 3.14 ✓

n = 13.7e72 ✓ All 3 are correct notation

n = 2.1E14 ✓

(20) Remainder operator is only available with int type (%)

* Also for floating point numbers, we don't have bitshifting operator and bitwise operator *

Date / / /

(21)
```
func main() {
    var n complex64 = 2i
    fmt.printf("%v, %T", n, n)
}
```
Output
(0+2i), comple

(22) With complex numbers we can do (+), (−), (*), (/) operation
Zero value of Complex NO − 0+0i

(23) How do you printed out only real and imaginary number
forex:
```
func main() {
    var n complex64 = 1+2i
    fmt.printf("%v, %T", real(n), real(n))
    fmt.printf("%v, %T", imag(n), imag(n))
}
```
Output
1, float32
2, float32

(24) Complex 64 − Turns out float 32
Complex 128 − Turns out and gives float 64
```
func main() {
    var n complex128 = 1+2i
    fmt.printf("%v, %T", real(n), real(n))
    fmt.printf("%v, %T", imag(n), imag(n))
}
```
Output
1, float64
2, float64

Date: / / /

(25) How will you be able to print complex128, complex64 as data type of a variable declared in a Go program.

for complex64

```
func main() {
    var n complex64 = complex(2,4)
        fmt.printf("%v,%T", n,n)
    }
```

Output

2+4i, complex64

for complex128

```
func main() {
    var n complex128 = complex(2,4)
        fmt.printf("%v,%T", n,n)
    }
```

Output

2+4i, complex128

(26) Text types are of two types → String (Any UTF-8 char)

- Rune (Represent any UTF-32 char)
- String
  1) That shows powerfulness
  2) But String can't encode every type of char that's available)

Date: / / /

(27) We can Treat String sort of like an array We can Treat String of text as collection of letters.

for ex:-

```
func main() {
    S := "This is a String"
    fmt.printf("%v, %T", S[2], S[2])
}
```

output -> 105, uint8

NOTE ✱ Strings in Go are actually an aliases for bytes.

uint8 is the type alias for bytes.

(28) How will you actually get String value along with String data type. for a specific letter like an array

```
func main() {
    S := "This is a String"
    fmt.printf("%v, %T", String(S[2]), String(S[2]))
}
```

output

105, String

Value of String

This will give you data type of String

(29) Strings are immutable

for ex:-

```
func main() {
    S := "this is a string"
    S[2] = "u"
    fmt.printf("%v, %T", S, S)
}
```

output -> this will throw an error.

1) cannot assign to S[2]

Explanation I can't assign string to a byte

2) can't use "u" (type string as type byte in assignment

Date: / / /

(30) How do you do sudo airthmetic operation with string, Concatination, add strings together.

for ex:-

```
func main() {
    S := "this is a string"
    S2 := "this is also a string"
    fmt.Printf("%v, %T\t", S+S2, S+S2)
}
```

output:- this is a string this is also a string

(31) How do you convert string into a collection of bytes.

OR

Slice of bytes / byte slicing

```
func main() {
    S := "this is a string"
    b := []byte(S)
    fmt.Printf("%v, %T", b, b)
}
```

output

[116 104 105 115 32 105 115 32 97 32 115 116 114 105 110 103]

[]uint8

ASCII Value OR UTF value of each charcter in that string.

The usecase for this is for many of the function supports byte slicing in Go.

For ex: If you want to ~~get~~ send the response back of an HTTP (Web service can) request then you get byte collection

Also, if you want to ~~get~~ send the files back, while a file on your HD is just a collection of bytes too

Date: / / /

(32) Any char in UTF-32 can be upto 32 bits long, but it doesn't have to be 32 bits long.

forex: Any UTF-8 char, which is 8 bits long, is a valid UTF-32 char.

(33) How to declare rune

```
func main() {
    r := 'a'        → with single quote
    fmt.printf("%v, %T", r, r)
}
```

output

97, int32

☆ Runes are just a type alias of int32 ☆

(34) Like in other languages we keep the constants name like

```
Const MY_CONST
```

But in Go the problem with that is ~~if you keep~~ the first letter is in uppercase and as remember from variable, if we get uppercase first letter that's gonna mean that the constant is gonna be exported, and we don't always want that.

So we name like Camelcasing (lowercase first letter)

```
Const myconst
```

(35) How do you declare the typed constant

```
func main() {
    Const myConst int = 42
    fmt.printf("%v, %T\n", myConst, myconst)
}
```

output

42, int

(36) We are not allowed to change the value of a constant

forex:

```
func main() {
    Const myConst int = 42
    myConst = 27
    fmt.printf("%v, %T", myConst, myConst)
}
```

output

Compiler error:

[Can't assign to myCon

Date: / / /

(37) Const has to be assignable at compile time.

for ex:-

```
Package main                                    Output
import (                        This will throw a compile time error-
    "fmt"                       Const initializer math.Sin(1.57) is not
    "math" )                        a constant.
func main () {
    const myconst float64 = math.Sin(1.57)
    fmt.Printf ("%v, %T", myconst, myconst)
    }
```

a) Explation :- In order to determine the Sin of that value, then you require function to execute which is not allowable at compile time; so can't set your constant equals to something that has to be determine at runtime And that includes things like setting it equals to flag that you are passing in your application when you run.

(38) Constant can be made up by any of the primitive types.

for ex:-

```
func main() {                                   Output
    Const a int = 14                            14
    Const b string = "foo"                      foo
    Const c float32 = 3.14                      3.14
    Const d bool = True                         true.
    fmt.printf ("%v", a)
    fmt.printf ("%v", b)
    fmt.printf ("%v", c)
    fmt.printf ("%v", d)
    }
```

Date: / / /

(39) Collection types are inherantly mutable

for ex:-

① you couldn't create an Array and declare that to be a constant type.

② Arrays are always gonna be variable types.

(40) Constants can't be shadowed.

for ex:-

```
Const a int16 = 27
func main() {
    Const a int = 14
    fmt.printf("%v, %T", a, a)
}
```

Output

14, int

```
Const a int16 = 27
func main() {
    const a = "hello"
    fmt.printf("%v, %T", a, a)
}
```

Output

hello, string

(41) We can do the airthmetic operations on constant + variable of same type.

for ex:-

```
func main() {
    Const a int = 42
    Var b int = 27
    fmt.printf("%v, %T", a+b, a+b)
}
```

Output

69, int

(42) What is implicit conversion feature with constants

```
func main() {
    Const a = 42
    Var b int16 = 27
    fmt.printf("%v, %T", a+b, a+b)
}
```

Example of untyped constants → Replace with 42 → 42+b, 42+b

output → 69, int16

Date: / / /

(43) What is iota →

It is a counter that we can use when we are creating Enumerated constants.

```
Const (
    a = iota      *
    b  ──→ Compiler automatically inferred the pattern out there. *
    c  ──→
)
```

Output
0
1
2

```
func main() {
    fmt.Printf("%v", a)
    fmt.Printf("%v\n", b)
    fmt.Printf("%v\n", c)
}
```

(44) What is write only variable in Go

```
Const (
    _ = iota
)
```

→ This tells the compiler that I know you are going to generate a value here but I don't care what it is, go ahead and throw that away.

(45) Typed constant work like immutable variables
→ Can interoperate only with same type

(46) Untyped constant work like literals
→ Can interoperate with similar types.

Date: / / /

(47) How to declare an array and why do we need them:-

```
func main() {
    grade1 := 97
    grade2 := 85
    grade3 := 93
    fmt.printf("Grades: %v, %v, %v", grade1, grade2, grade3)
}
```

Output:-
Grades: 97, 85, 93

using Array

```
func main() {
    grades := [3] int {97, 85, 93}
    fmt.Printf("Grades: %v", grades)
}
```

[3] → Size of the array (so here, 3 elements can be held by Array)

int → Type of data, that array is designed to store.

{97, 85, 93} → Initializing the syntax on array

Output
Grades: [97 85 93]

NOTE:- Array can only store one type of data

(48) Array allocates the elements into contingous memory, which help an application to access those elements, a little bit faster.

(49)

```
func main() {
    grades := [...] int {97, 85, 93}
    fmt.printf("Grades: %v", grades)
}
```

Explanation: Basically that says create an array that's just large enough to hold the data that I am gonna ~~paste~~ pass to you in literal syntax.

~~[...] → An array 3 elements in it.~~

Date: / / /

(50)

```
func main() {
    var students [3]string
    fmt.Printf("Students: %v", students)
}
```

Output:-
Students: [ ]
↑ Empty array

(51) How to print out the size of the array

```
func main() {
    var student [3] string
    student[0] = "Lisa"
    student[1] = "Jack"
    student[2] = "tom"
    fmt.Printf("Number of students: %v\n", len(students))
}
```

Output
Number of students: 3
using length function

(52) How do you declare array of arrays:

```
func main() {
    var identityMatrix [3][3] int
    identityMatrix[0] = [3]int{1,0,0}
    identityMatrix[1] = [3] int{0,1,0}
    identityMatrix[2] = [3] int{0,0,1}
    fmt.Println(identityMatrix)
}
```

Output
[[1 0 0] [0 1 0] [0 0 1]]

Another way

```
func main() {
var identityMatrix [3][3] int = [3][3] int{ [3] int{1,0,0},
[3] int{0,1,0}, [3] int{0,0,1}}
    fmt.Println(identityMatrix)
}
```

Date: / / /

(53) When you copy the array in Go, it copies the entire length of array

for ex:-

```
func main() {
    a := [...]int{1,2,3}
    b := a        -> it copies the entire length over there.
    b[1] = 5
    fmt.Println(a)
    fmt.Println(b)
}
```

output

[1 2 3]

[1 5 3]

[Copying refer to different underlying data / means onto a different memory location]

Explanation → The value b is assigned to the copy of array a,

(54)

```
func main() {
    a := [...]int{1,2,3}
    b := &a
    b[1] = 5
    fmt.Println(a)
    fmt.Println(b)
}
```

output

[1, 5, 3]

& [1, 5, 3]

Means there is no new memory being allocated for b.

Explanation → b is going to point to the same data that a has.
So a is the array itself and b is pointing to a.
↓
We are actually changing the underlying data for both.

(55) Arrays are fixed ~~in~~ size that has to be known at compile time definitly limit there usefulness.
In Go the most common use case of array is to back some thing called a slice.

Date: / / /

(56) How do you declare slice - data that store

```
func main() {
    a := []int{1,2,3}
    fmt.Println(a)
}
```

literal

Type of data we wanna store

pass the initialize data

Output

[1 2 3]

(57) ~~Slice~~ Slice is similar to array other than a couple of feature for ex:-

```
func main() {
    a := []int{1,2,3}
    b := a
    b[1] = 5
    fmt.Println(a)
    fmt.Println(b)
    // fmt.Printf("Length: %v\n", len(a))
    fmt.Printf("Capacity: %v\n", cap(a))
}
```

→ In slice it directly points to the data neither using (&)

All pointing to same underlying array

Output

[1 5 3]

[1 5 3]

Length: 3

Capacity: 3

↳ newly function in Slice-

(58)

```
func main() {
    a := []int{1,2,3,4,5,6,7,8,9,10}
    b := a[:]
    c := a[3:]
    d := a[:6]
    e := a[3:6]
    fmt.Println(a)
    fmt.Println(b)
    fmt.Println(c)
    fmt.Println(d)
    fmt.Println(e)
}
```

b := a[:] → Slice of all data

c := a[3:] → 4th element staring

d := a[:6] ← Until 6th Element

e := a[3:6] → B/w 4th to 6th Element Slicing

Output

[1 2 3 4 5 6 7 8 9 10]

[1 2 3 4 5 6 7 8 9 10]

[4 5 6 7 8 9 10]

[1 2 3 4 5 6]

[4 5 6]

☆ Inclusive index

☆ Exclusive Index

⁂ Slices are true for primitives

length of slice

Date / / /

(59) What is built-in make function. This takes two or three argumen

```
func main() {
    a := make([]int, 3)
    fmt.Println(a)
    fmt.Printf("Length: %v\n", len(a))
    fmt.Printf("Capacity: %v\n", cap(a))
}
```

make → function

3 → length of slice

[]int → Type of obj that you wanna create

in this ex we are gonna make slice of int

this will print out the value of the slice

length of the slice

print the capacity of ~~slice~~ array

Output

[0 0 0]

Length: 3

Capacity: 3

when you create a slice everything set of zero value

```
func main() {
    a := make([]int, 3, 100)
    fmt.Println(a)
    fmt.Printf("Length: %v\n", len(a))
    fmt.Printf("Capacity: %v", cap(a))
}
```

capacity

Slice has 3 elements in it

Output

[0 0 0]

Length: 3

Capacity: 100

underlying array has 100 elements in it

⁂ NOTE:- The slice has an underline array and they don't have to be equivalent ⁂

Reason: Unlike arrays, slices don't have to have a fixed size of their entire life. We can actually add elements and remove elements from them.

(60) append function

```
func main() {
    a := []int{}
    fmt.Println(a)
    fmt.Printf("length: %v\n", len(a))
    fmt.Printf("capacity: %v\n", cap(a))
}
```

with zero element

with 1 Element assigned

```
func main() {
    a := []int{}
    fmt.Println(a)
    fmt.Printf("length:
    fmt.Printf("capacity:
    a = append(a, 1)
}
```

Date: / / /

output

[]
length: 0
capacity: 0

output

[]
length: 0
capacity: 0
[1]
length: 1
capacity: 2 (why because memory was already allocated for zero element, So it has to create a new array with size 1 so the capacity turned to 2)

(61) How do you concate the slice function together.

```
func main()
    a := []int{}
    fmt.println(a)
    fmt.printf("length: %v\n", len(a))
    fmt.printf("capacity: %v\n", cap(a))
    a = append(a, []int{2,3,4,5}...)
}
```

output

[]
length: 0
capacity: 0
[2 3 4 5]
len: 4 capa: 4

In JavaScript they would call it Spread operator But in Go not sure.

This is going to spread the slice out into individual argument.

So these work exactly like-

a = append(a, ~~[]int~~ 2,3,4,5)