05

Wednesday • WK 06 (036-330)

February

Go lang :→ by Vineet Tumbe 202

Variables → variables has to be used

1. var <variable_name> <variable-datatype>

2. var <variable_name> <variable_datatype> = <value>

3. <variable_name> := <value>

printing statements

Println(" ")

Printf(" ",)

%v → value

%T → datatype

Scanned with CamScanner

var block -> when declaring multiple related variables

ex: var (
actorName string = "A"
companion string = "B"
doctorName int = 3
season int = 11
)

redeclaration of variables -> shadowing

redeclaration works only with the syntax

var <name> <type> = <somevalue>

or

var <name> <type>

levels of visibility in Go

lowercase at package level -> in package only
uppercase at package level -> global
block level

Scanned with CamScanner

primitive data type

Boolean type

Numeric type

- int → int 8, int 16, int 32, int 64, int 128
- float → float64, float32
- complex → complex128, complex64

uint ~~8~~ → byte
uint16
uint32

1+2i

text
- string
- rune

immutable

bit operator

& → and
| → or
^ → ex or
&^ → and not = opposite of or
>> → right shif
<< → left shift.

b := [] byte(s)

Scanned with CamScanner

Constants

Typed constant.

Untyped —"—

Enumerated —"—

Enumeration expressions.

Arrays —

<name> := [<size>] <datatype> {<values> }

var <name> [<size>] <datatype>

slice

<name> := [ ] <datatype> {<values> }

<name> := make([ ], <length>, <capacity>)

int int

<name> = append(<name>, <value>)

Scanned with CamScanner

xyz := map [string] int {"hello": 1}

Map

key? value?

<map name> := map [<datatype>] <datatype> {
    <key> : <value>,
    :
}

<map name> := make(map[<key dT>]<value dT>)

pop, ok := <mapname> ["key"] ← to check if key present or not.

if ok {

Struct

type <name> struct {
    number int
    name string
    :
}

tag →
type Animal struct {
    Name string ['required max : "100"] ← back tick
}

Scanned with CamScanner

has a

Composition → embedding

Go does not support OOP → is a model

if-else statement :→

```
if <condition> {
    :
}

if initializer; <cond1> {
    :
}
else {
    :
}
if <> {

}
else if <> {

}
else {

}
```


ARIMA 97.1413% → new cases
98.58% → total cases

Scanned with CamScanner

09 switch case : → Note → No break keyword
→ fallthrough
→ break

10 switch 5 {

11 case 1, 3, 7, 8 :
fmt.Println ("xyz")

12 case 2, 4, 6 :
fmt.Println ("uvw")

01 default :
fmt.Println ("abc")

02 }

03 i := 10

04 ~~case~~ switch {

05 case i <= 10 :
fmt.Println ("XYZ")

06 case i <20 :
fmt.Println ("ABC")

07 default :
fmt.Println ("AXZ")

}

Scanned with CamScanner

* Type Switch

```
var i interface{} = 1
switch i.(type){
    case int :
        :
    case float64:
        :
    case string:
        :
    default:
        :
}
```


Scanned with CamScanner

For loop →

(1)

```
for i:=0 ; i<=5 ; i++ {
    :
}
```

(2)

```
for i, j := 0, 0 ; i<5 ; i, j = i+1, j+1 {
    :
}
```

label

Loop:

```
for i:=0 ; i<=2; i++ {
    for j:=0; j<=2; j++ {
        if i*j > 2 {
            break loop
        }
    }
}
```


Scanned with CamScanner

for loops + slices

s := [] int {1, 2, 5}

for k, v := range s {
_, v := range s
fmt.Println(k, v)
}

O/P →

k	v
0	1
1	2
2	3

Defer → used to close resources

panic → to throw error messages / crashes

recover() → returns nil if app is not panicking

func main()
{
defer say
defer xyz
}

xyz
say

func say Hello()
fmt.Plan("Hello")

Scanned with CamScanner

Pointers →

a := 42

var a int = 42

var b* int = &a

Same stuff is sent

fmt. Println (a, b)

42 memory location of a

functions in Go →

values are passed by values

(1)

```
func <name> () {

    fmt. Println ("Hellow, playground")
}
```

(2)

```
func <name> (<param_name> <param DT>) {
    :
}
```


Scanned with CamScanner

fun sum (a, b int)

variatic parameters

sum (1, 2, 3, 4, 5, 6)

(3)

```
func sum (values ... int){
    fmt.Println (values)

    result := 0
    for _, v := range values {
        result += v
    }
    fmt. Println(meg, result)
}
```

(4)

```
func <name> ( <params>) (<return type>){

    .
    .
    return <return>
}
```


Scanned with CamScanner

Go has implicit interface implementation.

Go Routines → Concurrency & parallelism.

go run -race programfolder.go

Channels
↓
designed to sync transmission between go routines

channels are created using make function

ex: ch := make(chan int)

sending data into channel

ch <- data

pulling data

i := <- ch

Scanned with CamScanner

Blocking a channel

receive only channel

```
go func( ch <- chan int) {
        i := <- ch
        fmt.Println(i)
        wg.Done()
}(ch)

go func( ch chan <- int) {
        ch <- 42
        wg.Done()
}(ch)
wg.Wait()
```

Buffered Channel

```
ch := make(chan int, 50)
```


Scanned with CamScanner

for range in buffed channel.

```
ch := make(chan int, 50)        wg.Add(2)
go func( ch <- chan int ){
    for i := range ch {
        fmt.Println(i)
    }
    wg.Done()
}(ch)

go func ( ch chan <- int) {
    ch <- 42
    ch <- 27
    close(ch)
    wg.Done()
}(ch)

wg.Wait()
```

Scanned with CamScanner

```
for {
        if i, ok := <- ch ; ok {
                fmt.Println(i)
        } else {
                break
        }
}
```

Select Statement

```
donech := make (chan struct{})
for {
    select {

        case entry := <- chan :
            :

        case <- doneCh :
            break
    }
}
```

Note: Do not give default

Scanned with CamScanner