GOLANG

1. Variables:

Declaration

var <var-name> int = 42
(int → type)

i= 42

* While initializing, we can omit the above things by

i := 42

Note: fmt.printf is a format

%v → value
%T → type.

Types

- int
- float 32

] → Later

Multiple declaration

```
var (
    x int = 1
    y int = 2
)
```

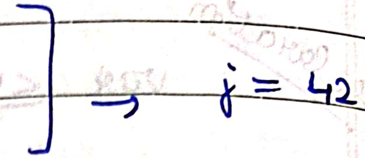
Note: Shadowing works the same as C/C++.

Note: Unused variable is an error here.
You can bypass this by using variable names as '_'.

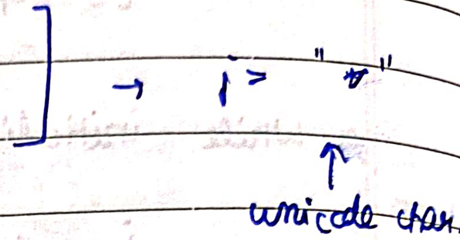
Note: Lower case variable name → Scope is package level
Upper case variable name → Global level scope

Type conversion

```
var j float32
j = int(i)
```




```
var j string
j = string(i)
```



* String conversion in golang is done by:

```
import (
        "strconv"
)
strconv.Itoa(i)
```

2. Primitives:

Note: All primitives have 0 as the default value.

→ Boolean

```
var n bool = true
```

→ Numeric

int → default int

int8	uint8	float32
int16	uint16	float64
int32	uint32	
int64		

byte = uint8

* var n float32

n := 3.14
↑
float64

→ Complex nos. < complex128 / complex64

var n complex64 = 1+2i

Functions
real (n)
imag (n)

var n complex128 = complex (5,12)

→ Strings (Text)

I. var s string = "this is"

s[2]
↑
uint8 | byte

~~s[2] = "u"~~] strings are immutable. but can be concatenated.

Note: we can convert a string into collection of bytes.

b := []bytes (s)

II. Runes
Alias for int32
It is for UTF-32 encoded strings

3. Constants :
const constVar int = 42] Need to be initialized

* Constants can also be shadowed.

Enumerated Constants

```
const (a = iota
       b
       c
)

const (
    x = iota
    y
z)
```

Note: Newline is required in golang or semicolon

☆
```
const (
    _ = iota
    KB = 1 << (10 * iota)
    MB
    GB
)
```

A smart example of using iota.

4. Arrays & Slices

```
grades := [n] int {97, 85, 93}    or
   or                                  grades := [...] int {1, 2}
var grades [n] int
grades [0] = 1
```

★ len(grades) gives length.

Note: There are pointers in array. (Golang)

```
a := [...]int {1, 2}
b := &a
```

Now, array's size need to known at compile time, so we use slice.

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

Note: slices are like objects (one change changes another reference)

```
b := a[1:2]
```

(slicing) can be done on both arrays & slices.

Slice Operations

→ `a = append (a, 1)`

5. Maps & Structs

```
population := map[string] int {
    "California" : 24,
    " Texas "    : 12,
}
```

→ `population ["x"] = 120`

→ `delete (population, "x")`

```
type Doctor struct {
        number int
        name string
        patients []string
}
```

```
aDoctor := Doctor {
        number: 3,
        name: "SK",
        patients: []string {"Bro", "Go"}
}
```

→ aDoctor.name

☆ Golang doesn't support OOPS but can support composition by embedding structs.

(6.) If & Switch Statements

Note: No single block statement in Golang

```
if true {

}
```

operators: > == >= < <= != | && ||

```
switch value/variable {
    case 1:
    case 2:
    default:
}
```

No break required

if we do want to continue the case, we can use "fallthrough" keyword.

Also, you can use break to short-circuit

Note: There is no "comma" operator in golang.

(7.) Looping

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

}
```

Multiple variables

```
for i, j := 0, 0 ; i<5 ; i, j = i+1, j+2 {

}
```

for-range loop

```
for k, v := range arr {
    fmt.printf (k, v)
}
```

→ same can be done for strings, maps.

Infinite loops

```
for {

}
```

(8.) Defer, Panic, and Recover

defer → run the statement at the end before func^n returns.

Note: It does evaluate the statement but executes it at the end. They are run in "LIFO" order.

```
a := "start"
defer fmt.Println(a)
a = "end"
```

→ start.

Date
Page

Panic

↓ usually done when program can't continue at all.

panic (some-string)

* It basically halts the program.

recover

- Used to recover from panics
- Only useful in deferred funcⁿs.

(9) Pointers

var b *int = &a

*b → value of a

Note: Pointer Arithmetic is not available in GoLang.

We can have pointers to array, structs, maps, . .

* "nil" is synonym to "NULL" in C/C++.

(10) Functions

```
                                (ret1, ret2)
func  &lt;func_name&gt;  ret-type( param1 type, param2 type) {

}
```

everything else is similar to C/C++

Note: There are anonymous functions in golang but looks useless.

11. **Interfaces** (Describe behaviours unlike structs which describe variables)

```
type <interface-name> Interface {
    Write ([]byte) (int, error)
}
```

↑ Method Defn.

```
type ConsoleWriter struct {}

func (cw ConsoleWriter) Write (data []byte) (int, error) {
    n, err := fmt.Println (string (data))
    return n, err
}
```

Method declaration

Embedding Interfaces

```
type WriterCloser interface {
    Writer   <- Interface1
    Closer   <- Interface2
}
```

12. **Go-routines** (Abstraction of Thread)

```
go func_routine();
```

☆ "Go has a scheduler that runs these "small threads" called as go-routines.

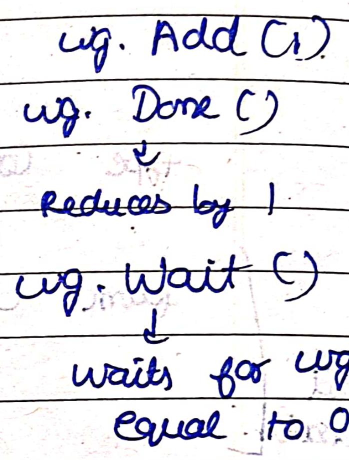
Go is better because threads are lighter.

☆ Main function is another go-routine.

Note: We can create sync between different go-routines using "mutex".

```go
import (
    "sync"
)

var wg = sync.WaitGroup{}
var m = sync.RWMutex{}
```



Note: runtime. GOMAX PROCS (-1)
↓
Returns threads available.

runtime. GOMAXPROCS (10)
↓
sets no. of threads

* Thrashing is also possible here.

Best Practices.

- Don't create go-routines in libraries.
- Know how a goroutine will end.
- Check for race conditions at compile time.

go run -race src/main.go.

- It's better to have no go-routines when we need to create hard synchronization.

buffer + lock
↑

(13.) Channels → ways to share data between go-routines while avoiding race conditions.

```
var wg = sync.WaitGroup{}

func main() {
    ch := make(chan int)
    wg.Add(2)

    go func() {
        i := <- ch          → recieve data from channel
        fmt.Println(i)      //42
        wg.Done()
    }()

    go func() {
        ch <- 42            → send data to channel.

        wg.Done()
    }()
    wg.Wait()
}
```

* Each channel can only have 1 data at a time. So, any other go-routine waits for channel to receive data.

* Channels are bi-directional but can be casted to uni-directional.

unless we use buffer channel

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

⇒ sync.cond (condition variable)

If
a) Some threads working on a shared data
b) Some property waiting for that condⁿ to become true

```
var mu = sync.Mutex
cond := sync.NewCond(&mu)

                go func() {
                    mu.Lock()
                    defer mu.Unlock()
                    cond.Broadcast()
                }()

Main
 ↓

mu.Lock()
for (condⁿ = false) {
    cond.Wait()
}
mu.Unlock()
```