Basic Code

```
Package main

import (
    "fmt"
)

func main () {
    fmt.Println ("Hello")
}
```

---

1 -> How do we run the code in project

2 -> What does 'Package main' mean?

3 -> What does 'import "fmt"' mean?

4 -> What's that 'func' thing?

5 -> How is the main.go file organized?

(1) Go CLI

go build → Compile a bunch of go source codes file

go run → Compile & execute one or two files

go fmt → Format all the code in each file in the current directory

go Install → Compile & "Install" a Package

go get → Downloads the raw source code of someone else's package

go test → Runs any tests associated with the current projects

(2)

Package == Project == Workspace

Package: It is a collection of common source code called files

Package Main

main.go

```
Package main

import "fmt"

func main() {
    fmt.Println("Hi there!")
}
```

support.go

```
Package main

func support() {
    fmt.Println("Hi")
}
```

helpa.go

```
Package main

func help() {
    fmt.Println("I love ?")
}
```

Types of Packages

Executable

↓

Generate a file that we can run

Reusable

↓

Code used as 'helpers'. Good place to put reusable logic

Executable Package

↓

Package main

(main is spel)

Defines a package that can be compiled & then *executed. Must have a func called 'main'

Reusable Package

Package calculator

↓

Defines a package that can be used as a dependency (helper code)

Package Uploader

↓

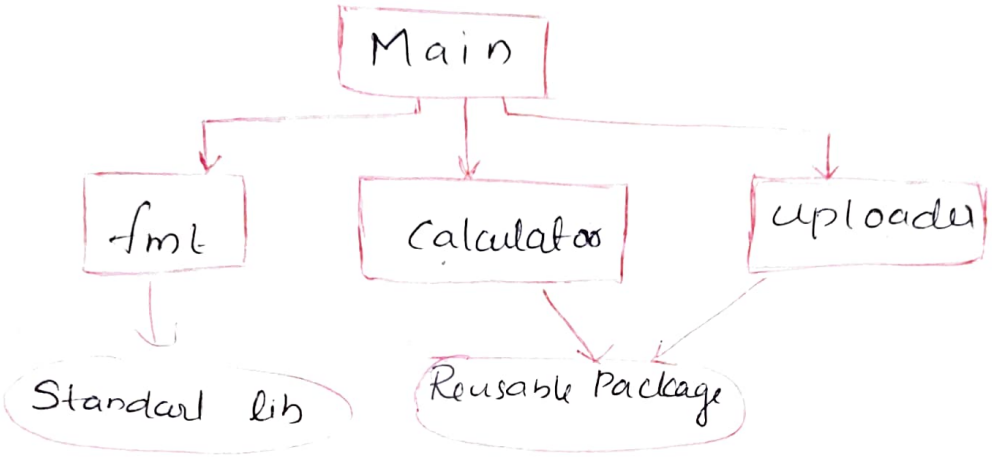
Define a package that can be used as a dependency

(3) import "fmt"

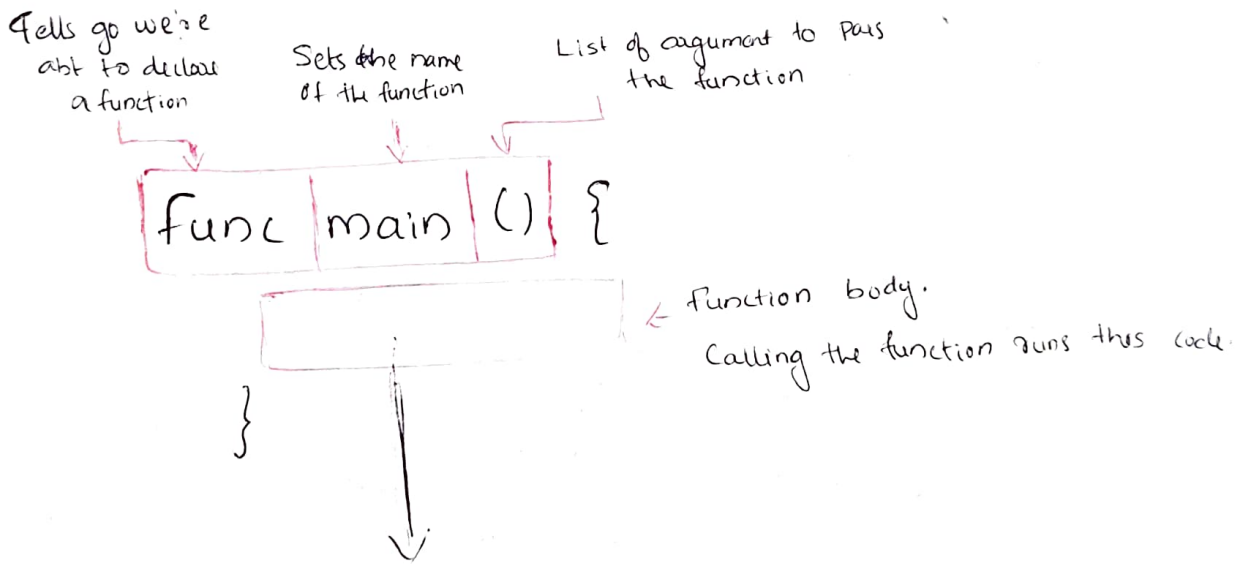
Used to give package main access to all the code & all the functionality that is contained inside the other Package called "fmt"

* fmt - format.

fmt library is used to print out lot of library informations



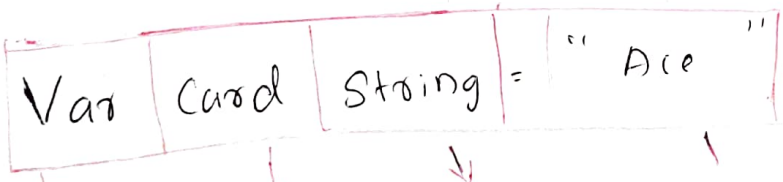
(4)



fmt . Println ("hi")

Cards

NewDeck	Creates a list of Playing Cards. Essentially an array of strings
Print	Log out the contents of a deck of cards
Shuffle	Shuffles all the card in a deck
deal	create a 'hand' of cards
SaveToFile	Save a list of cards to a file on the local machine
NewDeckFromFile	Load a list of cards from the local machine



We're abt to create a new variable

The name of variable will be 'card'

Only a "string" will ever be assigned to this variable

Assign the value "Ace" to this variable

Go - Statically typed language

Basic go types - Boolean, String, int, float

eg.

```
Package main

import "fmt"

func main() {

    //var card string = "Ace"

        card := "Ace"                    //new card

    fmt.Println(card)

}

func newCard() string {

    return "five"

}
```

String can be represented by :=

→ **Array** - Fixed length list of things

→ **Slice** - An array that can grow & shrink

eg: `card := []string{"Ace", newCard()}`

`cards = append(cards, "Six")`

Index of this element in array

current card we're iterating over

Take the slice of 'cards' & loop over it

```
for index, card := range cards {
    fmt.Println(card)
}
```

Run this one time for each card in the slice

Base Go Types

string | integer | float | array | map

We want to "extend" a base type & add some extra functionality to it

```
type deck []string
```

Tell Go we wanna create an array of string & add a bunch of functions specifically made to work with it

Function with 'deck' as a 'receiver'

A function with a receiver is like a "method" - a function that belongs to an "instance"

'Cards' folder

main.go → code to create & manipulate a deck

deck.go → code that describe what a deck is & how it works

deck_test.go → code to automatically test the deck

Map is a collection of key value pairs

String to Integer
"x" -> 90

Integer to Integer
1 -> 40

Integer - Units means - unsigned Integer

Int means - Signed Integer

Float - float 32 — 32 bit or 4 bytes

float 64 — 64 bit or 8 bytes

String = "abc" "90%"
10 bytes

Printing - Print, Println, Printf
↓
-format specifier

format specifier - %v = value

%v = default format
%T = type of the value
%d = integer
%c = character
%q = quoted character string
%s = Plain string

%t = true or false

%f = floating number

%.2f = floating number upto 2 decimal places

Scope - Inner blocks can access variables declared within outer blocks

- Outer blocks cannot access variables declared within inner blocks.

Zero Values : Bool - false

Int - 0

float64 - 0.0

String - ""

Pointers, Function, Interface, maps } nill

User Input - fmt.Scanf("%<format specifier>(s)", Object_arguments)

count : Number of arguments that the function writes to

err : Any error thrown during the execution of the function

Types of variable

%T format specifier

reflect.TypeOf function from the reflect package

★ Type Casting - Process of converting one data type to another

eg: var i int = 90
var f float64 = float64(i)
fmt.Printf("%.2f\n", f)

ans: 90.00

→ Itoa() : convert integer to string
→ Atoi() : convert string to integer

Operater - Symbols that help us to perform specific mathematical & logical operations on operands

Types of Operators - Compaison Operators

(==, !=, <, >, >=)

- Arithmetic Operator

(+, -, *, /, %, ++, --)

- Assignment Operators

(=, +=, -=, *=, /=)

- Logical Operator

(&&, ||, !)

- Bitwise Operators

(&, |, <<, >>, ^)

Loop

declaring, initializing, assigning

for i, card := range cards {

fmt.Println(i, card)}

Arrays

Collection of similar data elements stored at contiguous memory location

-> Have fixed length

-> Elements should be of the same data type

eg: Cards := [] string {"Hai"}

Slice

It is a flexible & dynamic data structure that Provides more ways to work with the sequence of elements compared to traditional array

* Variable typed
* More flexible

Components of slice — Pointer
Length
capacity

Pointers - Variables that store memory address

Struct / Class - User defined

Define custom data type by grouping together variables with different data types

Eg:

```
Package main

type Person Struct {
    firstName string
    lastName string
}

func main () {
    alex := Person {firstName: "Alex",
                    lastName: "Andrew"}
    fmt.Println(alex)
}
```

→ Method :-

Go allow you to define methods associated with structs, which can be a struct or any other type

→ Interfaces :-

Go support Interfaces, which define a set of method signature.

→ Encapsulation :-

Go promote encapsulation by using Struct field & method with visibility modifier

- lowercase - Private (accessible within the same package)
- Uppercase - Public (accessible outside the package)

→ Go does support certain object-oriented concept & provides mechanism to achieve encapsulation, data abstraction & code Organisation.

Advantages

-> Strong & Statically typed languages

-> Simplicity & readability

-> Concurrency & Goroutine

-> fast compilation & execution

-> Garbage collection

-> Standard libraries

-> Cross-platform support

-> Tooling & ecosystem

function

A function is basically a collection of statement that perform some specific task & return the result to the caller.

```
Func function_name (Parameter list) (Return type) {
    // function body
}
```

```
Func add (a, b int) Int {
    Sum := a + b
    return sum
}
```

→ function - call by value
call by reference

* Call by value :

•) Parameter passed to a function are called actual parameter

•) The parameter recieved by a function is called formal parameter

* function call by reference :

Both the actual & formal parameter refer to the same location So any changes made inside the function are actually reffered in actual parameter of the caller

Golang

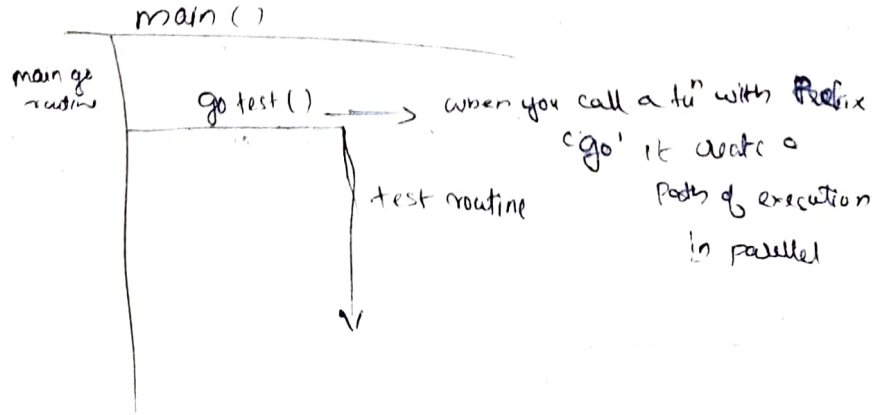
Goroutines _ Concurrency in golang

Go routine is a function or method which execute independently & simultaneously in connection with anyother goroutine present in your program.

→ Go routines are light-weight thread managed by the go runtime.

when go routine start, it branches out from the current path of execution & start a new path

→ When main go routine exits, all other routine will get terminated irrespective of its completion.



Go channels

Channels are passage in which we can send signals to synchronise go routines.

- Channels could be created using 'make' & Send & receive can be done through '<-' (channel) operator.

Buffered & Unbuffered channels

→ Unbuffered :-

- Send & recieve happen @ same time
  ∴ It guarantees successful delivery of signal.
- By default

→ Buffered :-

- It doesn't guarantee delivery & block only when buffer is full.
- It store a certain number of value before blocking the send operation

- To create a buffered channel we need to specify the buffer size at the time of declaration.