**Go Language Study**

**Why Go Lang**

Fast & efficient. This is the only language which takes the advantage of multi core processors

**Go Lang Common Source code Structure**

bin

pkg

src

github.com

<username>

Folder/repo

**Important Environment Variables**

GOROOT - Installation path of golang

GOPATH – Workspace Path

**IDE**

* Visual Studio Code – Search for Plugin “go lang”
* Debugger – use delve <https://github.com/derekparker/delve/blob/master/Documentation/installation/linux/install.md>
* **Recommended Books**

Ardan Labs

**Packages in go**

Folder name and the package name has to be same (like java). One folder can contain multiple files in it

**Example**

import ( “fmt”,”github.com”)

Third party libraries should have a fully qualified domain name

**Capital Case vs. Small Case**

Function which is present inside a package is only visible when the first letter of the function name is capitalized, otherwise it is not visible

**Go Commands**

* go run <hello.go>
* go build

This creates the specific exe name

* go clean - Cleans up all the executable
* go install - Executable is built and put into workspace

Packages folder 🡪 archive file is created

* go help <command name>
* golint – Clean up all the code and provides error messages
* go fmt – Formats the code, alligns

**Syntax Spec**

* Semi colon is not required
* Unused variables or functions are called as Blank identifiers and are not allowed in Go Lang

**Side note**

* {} This is also called moustache in Web framework

**Basic Types of Variables**

* int { 8, 16, 32, 64 }
* uint { 8, 16, 32, 64 }
* byte 🡪 Equivalent uint8
* rune 🡪 Equivalent to uint32
* int 🡪 size of at least 32 but it can be more as well
* uint 🡪 size of at least 32 but it can be more as well
* float {32,64}
* bool
* var

golang is a statically typed language. However, when declaring and initialization – It is fine to skip the type

import (fmt)

func main() {

var i int = 1

fmt.Println(i)

}

Here is not declared with any specific type

func main() {

var i

fmt.Println(i)

}

This program will error out because the variable I is not explicitly declared

**Short hand notation of initialization and declaring variables in go lang**

**Short Hand**

func main() {

a:=10

b:=”sunil”

c:=4.17

d:=true

}

**Another Way**

func main() {

var a = 10

var b = “sunil”

var c = 4.17

var d = true

}

**Regular Way**

func main() {

var a int = 10

var b string = “sunil”

var c float = 4.17

var d bool = true

}

**Data Structures**

* Arrays
* Slice
* Map
* Struct

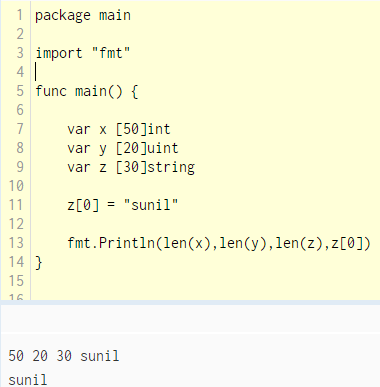
**Arrays 🡪 Not Dynamic**

var x [50]int

var y[50]uint

var z[50] string

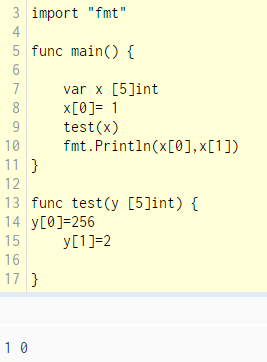
var m[50] byte



**Arrays not reference type**

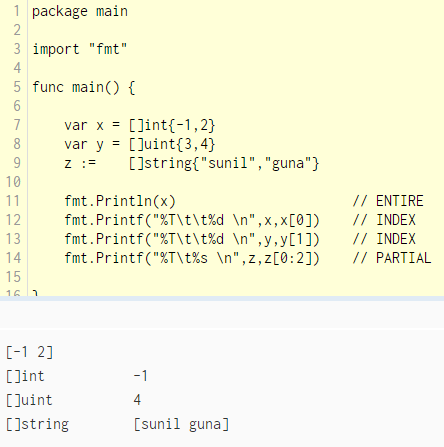
**Example**

* Values are not changed in the below program

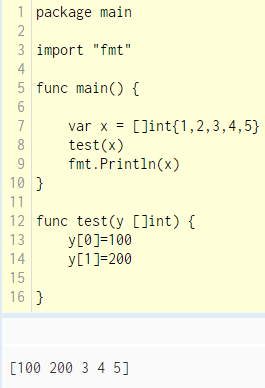


**Slices – Reference Type**

* Like array but no need to explicitly mention the size
* Slice can be fetched based on indexes, entire , partial like the sample below



**Reference Example**



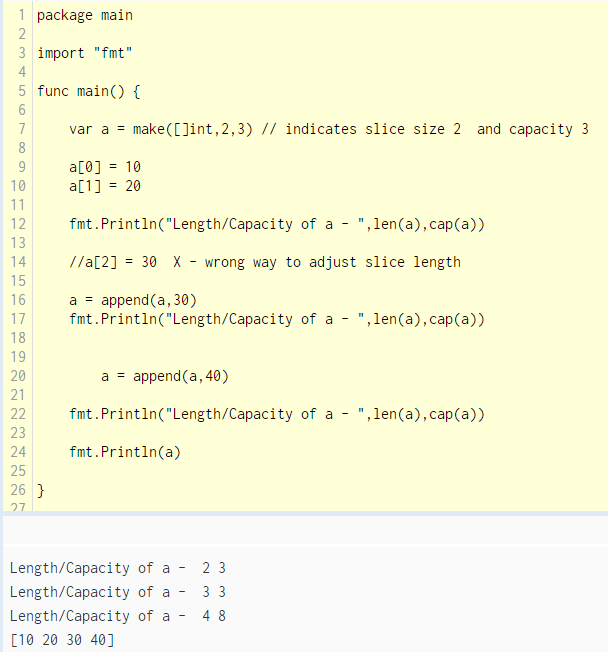
* Internally, Slice points to an underlying Array
* Slice can vary by size but if the underlying array size gets increased then whole reshuffling has to happen

**Ex: Underlying Array size : 100**

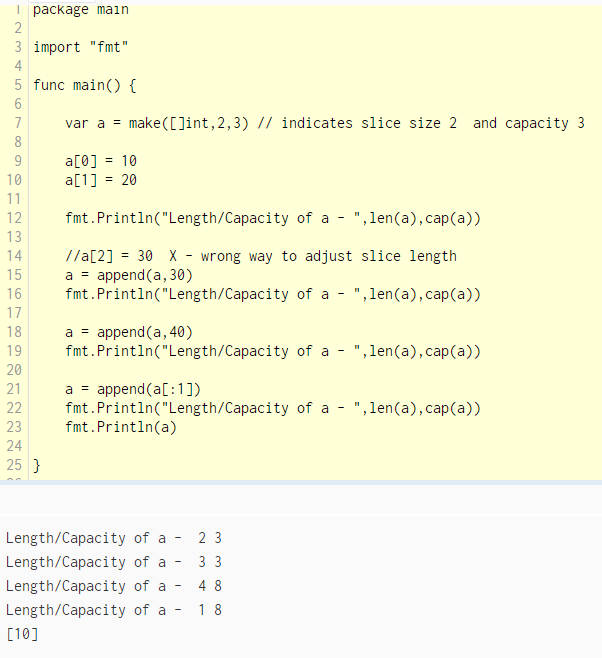
* Slice size 10 ,20 .. 100 🡪 no problem
* Slice – 101 🡪 Issue – Go automatically creates array size of 200 and copies the existing array to the new one and then issues slice 101 . This process is costly
* make command is used to set the slice length and underlying array size

make([]T, length, capacity)

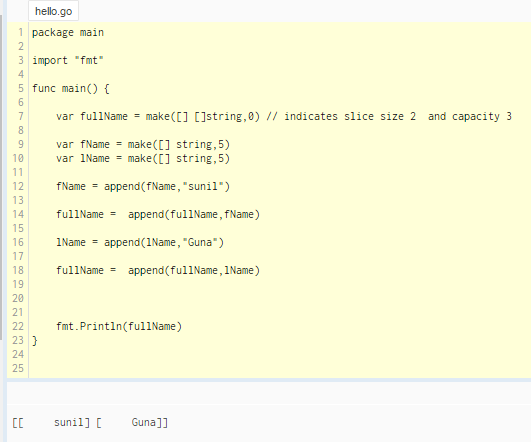
make([]int,50,100) or new ([100]int) [50:100]



**Sizing & Resizing Slice**

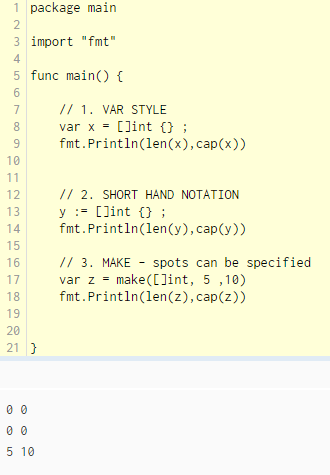


**Two Dimensional Slice**



**Creating a Slice**

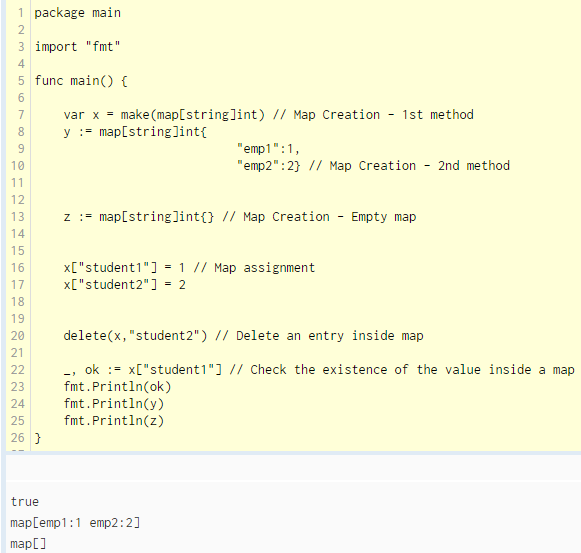
* Make is the preferred option



**Maps – Reference Type**

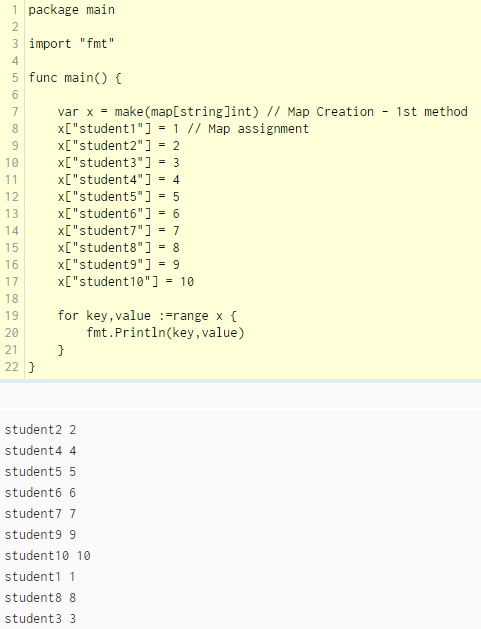
* Unordered ( Because of hash map)
* Maps underlying data structure is Hash tables
* Quick look up

**Map creation, adding elements, deletion, existence check**

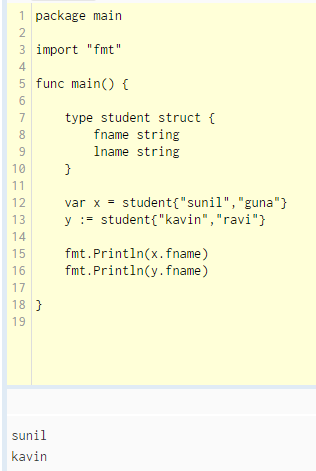


**Range loop – Map**

* Result is unordered



**Struct**



**Channels – Reference Type**

<Look at the end>

**OOPS in Go Lang**

**Encapsulation**

* is supported using struct
* export/unexport or visible/invisible is equivalent to Public/Private

**Inheritance**

* using Embedded Types

**Polymorphism**

* interfaces

**Scope**

* Block Level Scope - Usually like any other language
* Package Level Scope – Can be accessed across all the files inside the package

import “fmt”

var x int = 42

func main() {

}

x variable can be accessed by any of the files inside the package

**Functions**

func add(x int, y int) int {

return x + y

}

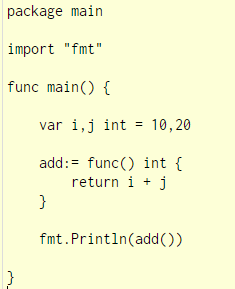
func main {

a:=add(4,5)

}

**Anonymous Functions**

* Function inside a function is possible only using anonymous function capability



**Constant Declaration**

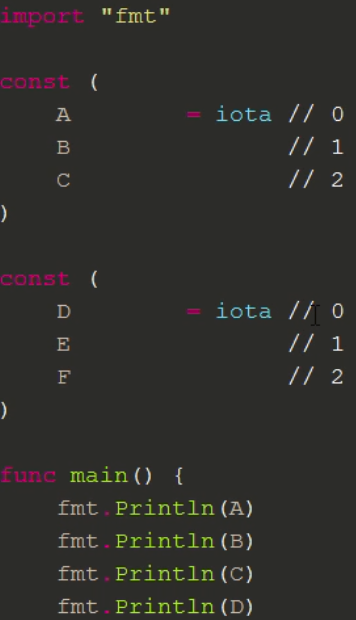
* Golang – It is possible to declare multiple variables at a time



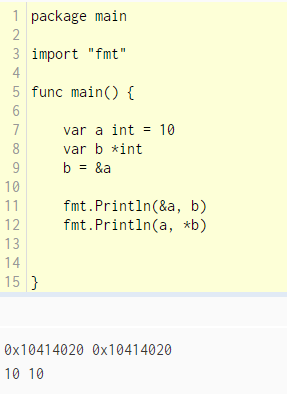
* TYPED : const name string = “hello” 🡪 This indicates TYPE String
* UNTYPED : const name = “hello” 🡪 TYPE is not decided until compilation is done

**IOTA Declaration – IOTA means the smallest amount**

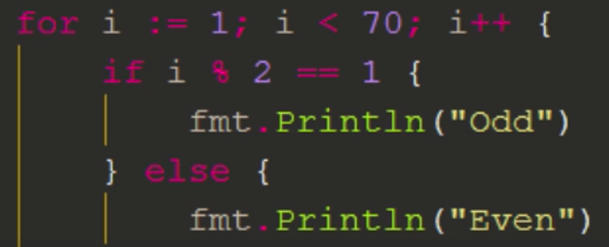
* Golang – It is possible to declare many iotas. Can be used only with const keyword

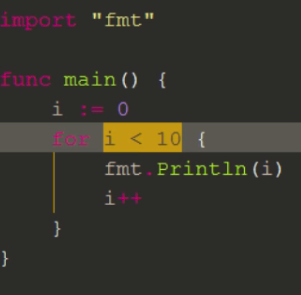


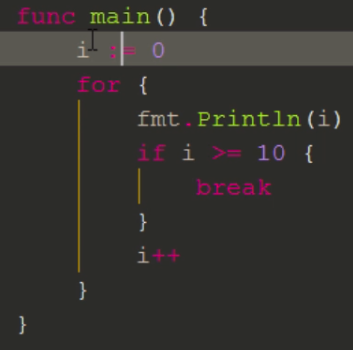
**Pointers**



**For loops/if condition doesn’t have open/close braces**





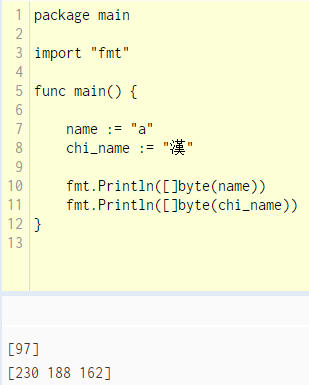




* There is no do loop or do while loop in Go Lang

**Rune**

* rune is equivalent to uint32
* UTF-8 is can be represented using rune as UTF-8 scheme is 4 byte



**You can notice,**

* character a, only 1 byte is used
* Chinese character, 3 bytes are utilized

Declaring rune

var chi\_name rune = '𠜎'

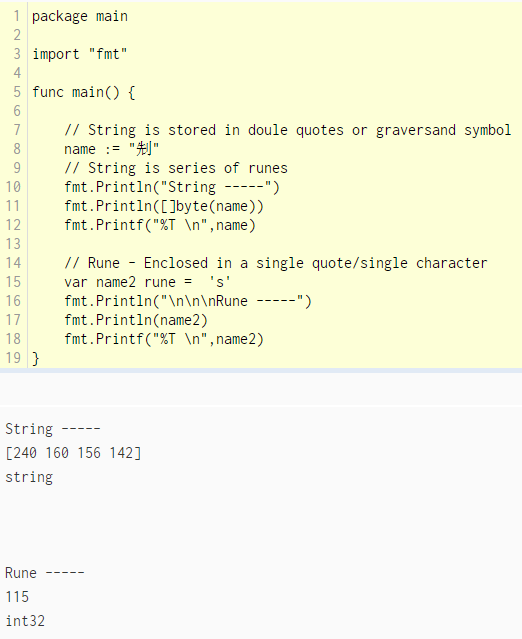
Single quote can only be used to assign to rune

**Strings**

var a = “hello”

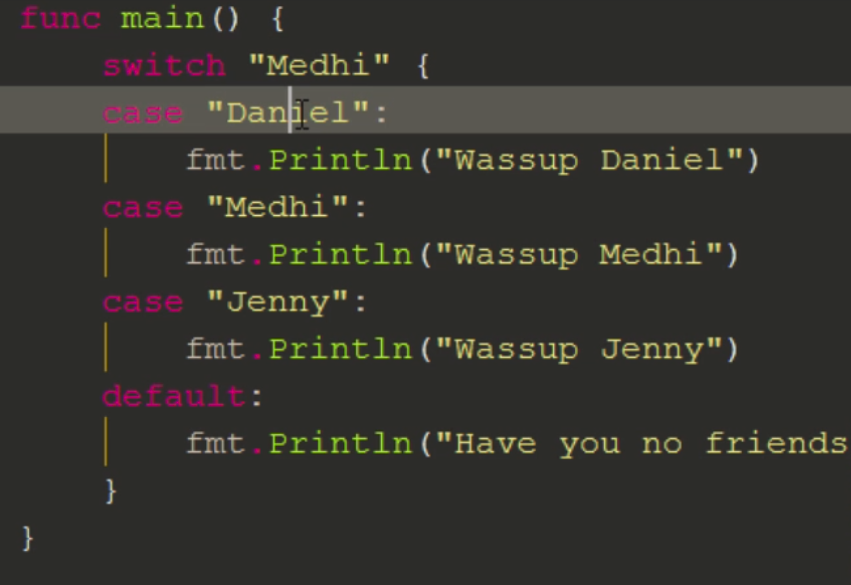
If there are double quotes inside a string , use the graversand symbol

Var x = `“hello`

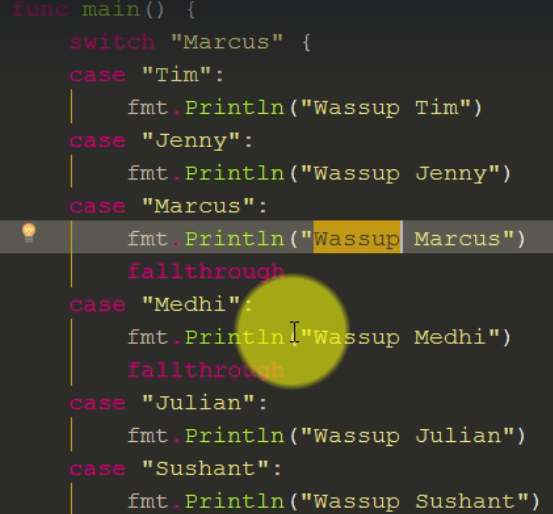


**Switch case**

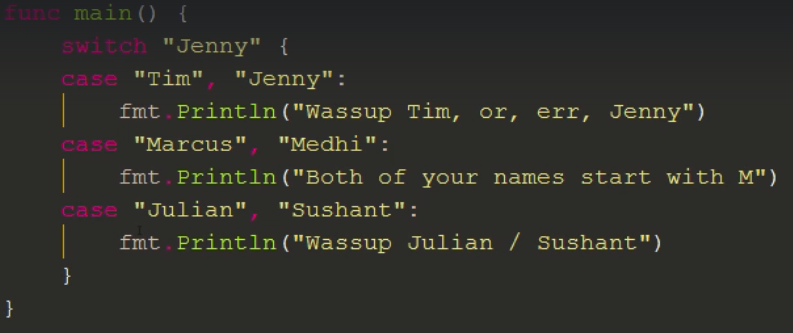
break statement is not required

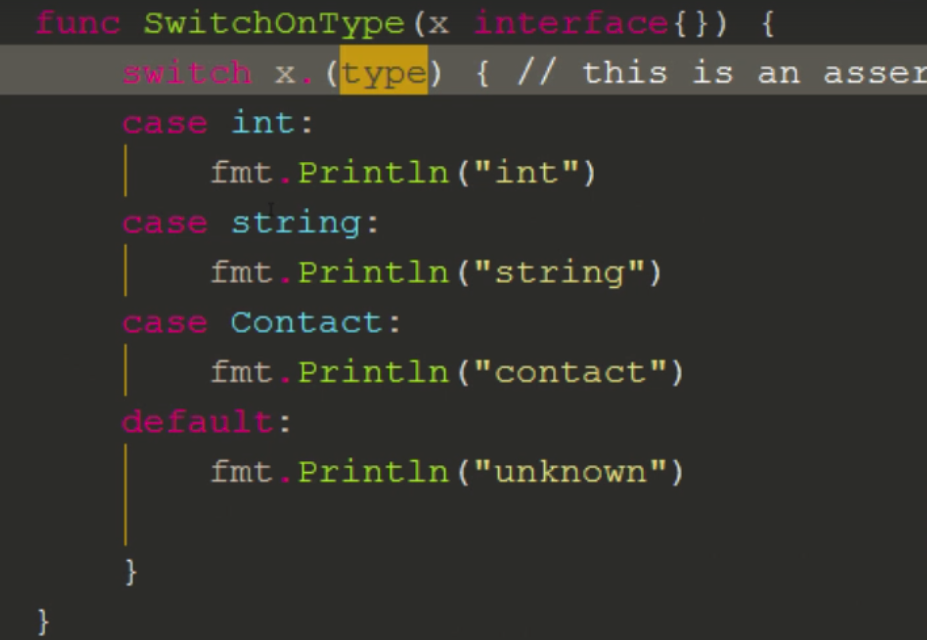


fallthrough – After matching the condition, if the fallthrough keyword exists – it executes the next case as well



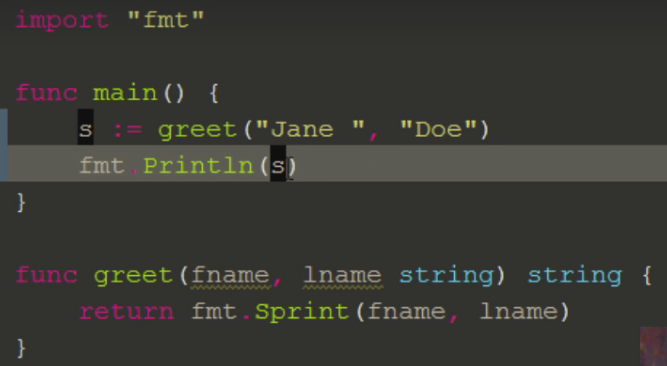
**Multiple conditions**



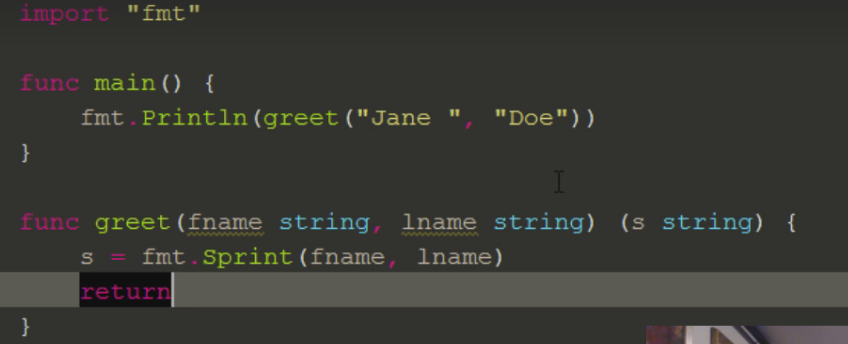


# Functions

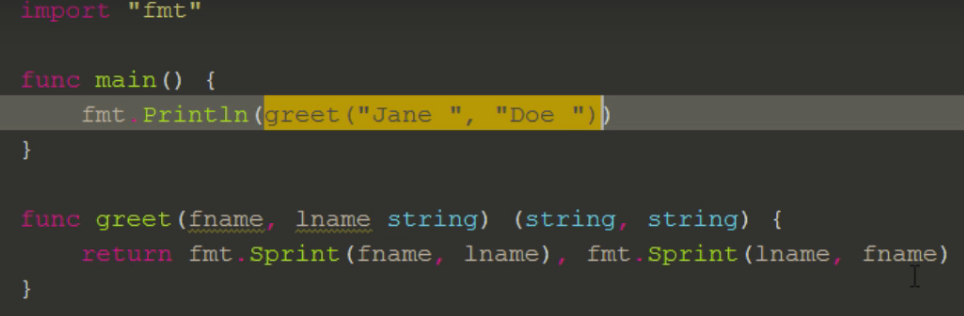
* Simple Function



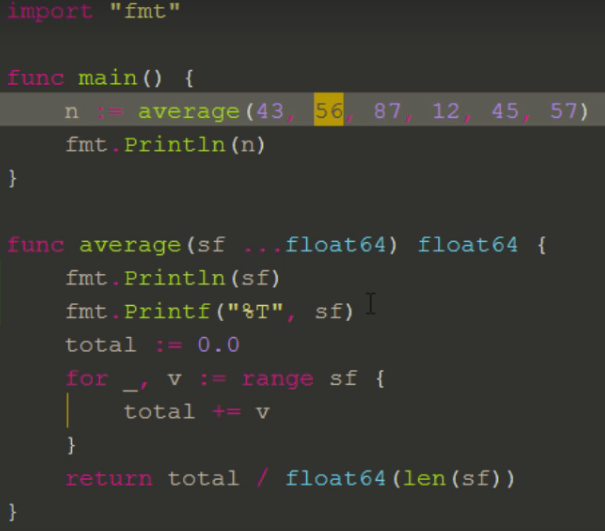
* Solidity style return



* Multiple return types



**Variadic Params**



Here

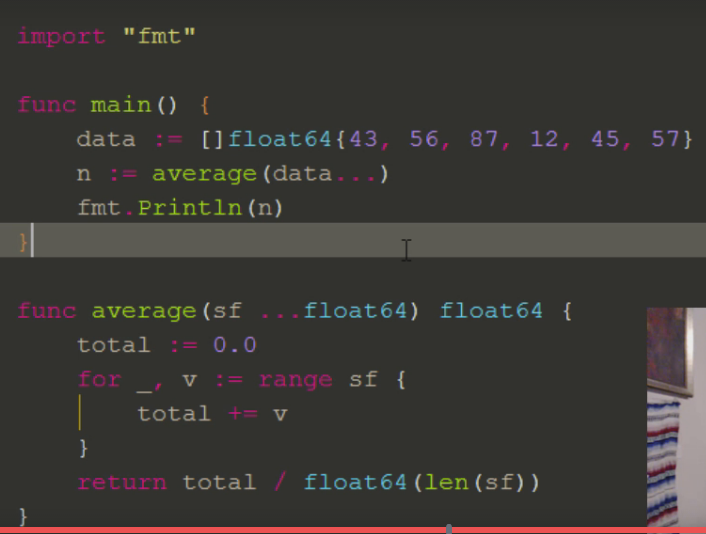
* The values 43, 56, 87, 12, 45 and 57 🡪 is kind variable type
* sf … takes variable list of argument. This as per declaration creates a float64 type

[ 43 56 87 12 45 57] 🡪 creates a slice

For loop

Takes blank identifier and loops through the range ( sf slice)

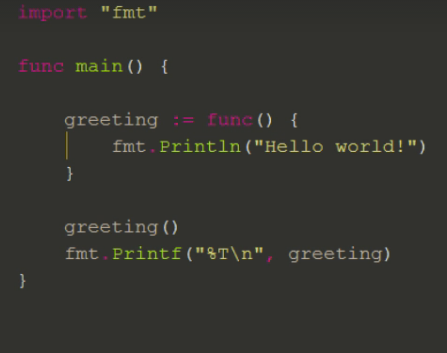
**Variadic Arguments**



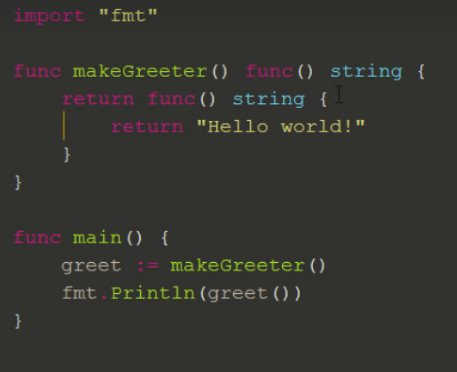
**Here**

* Passing varying number of arguments
* Slice float array is passed to variable float64

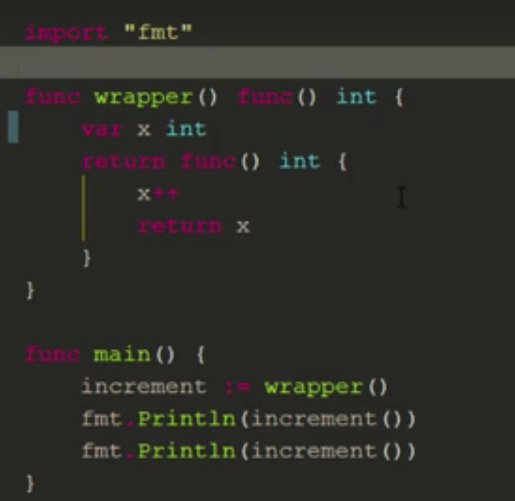
**Function Expressions**

* Function inside a function is possible only using anonymous function
* 

**Closures**

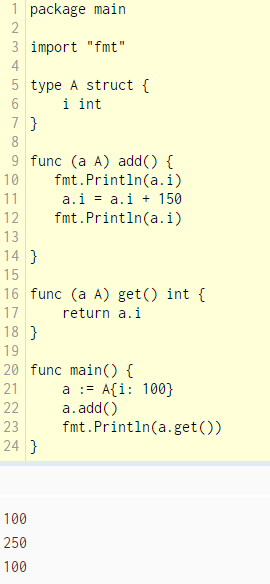


* Scope is very important aspect of closure
* Even though wrapper function ends, scope of x remains when increment() happening



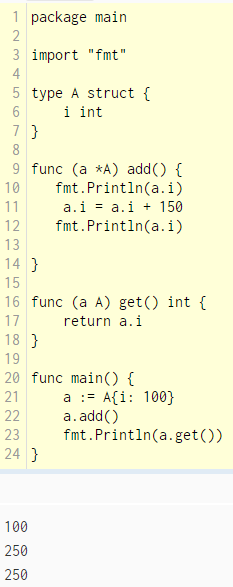
**Function Receivers**

* add() function has a receiver ( a A )
* This is useful when we need to call like object based syntax. However it is only the copy which is getting passed. In this case, copy of struct is passed to add() function
* Result is still 100 ( because it is just a copy)

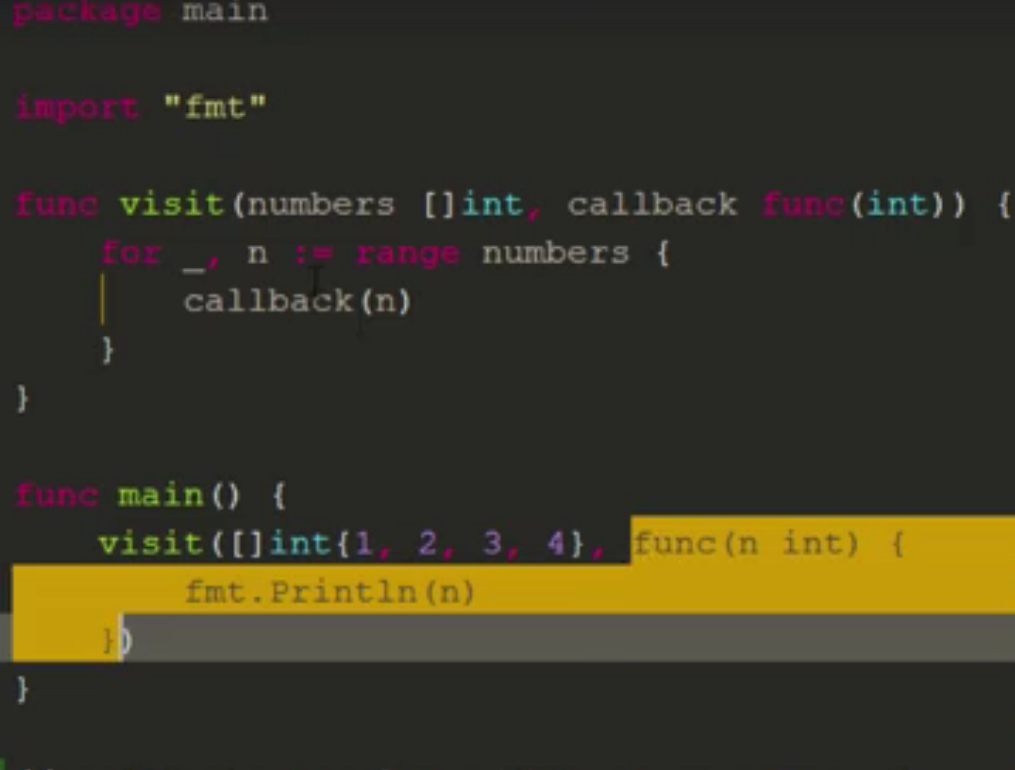


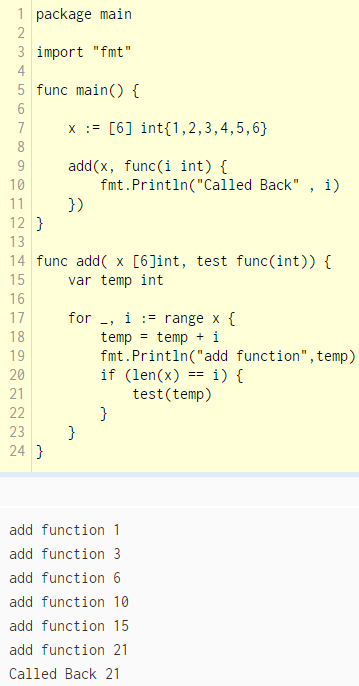
**Function receiver using a pointer reference**

* add() function has a receiver ( a \*A )
* Reference is passed. Hence we are operating with a single copy
* 250 is the result



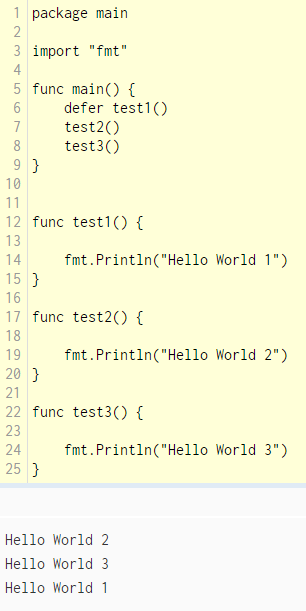
**Callbacks**



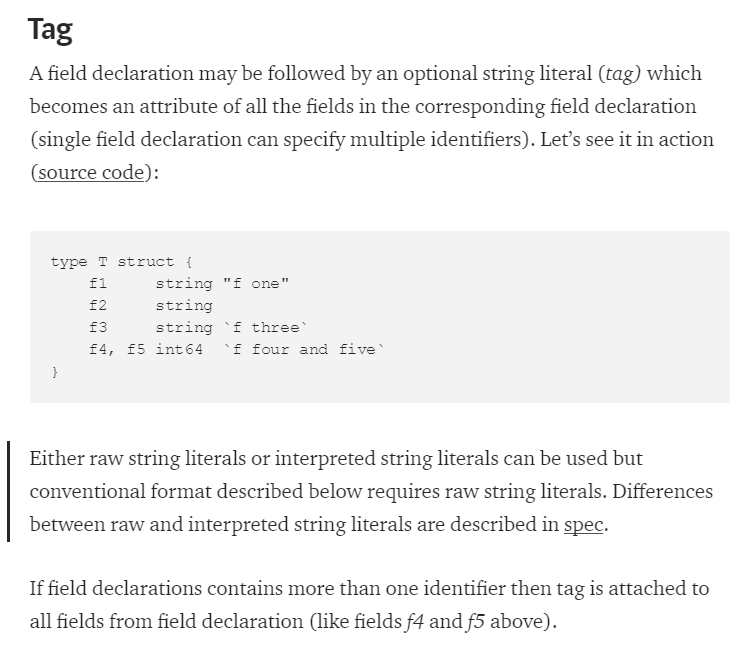


**Defer**

* Function execution will be deferred temporarily and will be executed at last. Typically can be used for closure activities like File close, DB close etc.

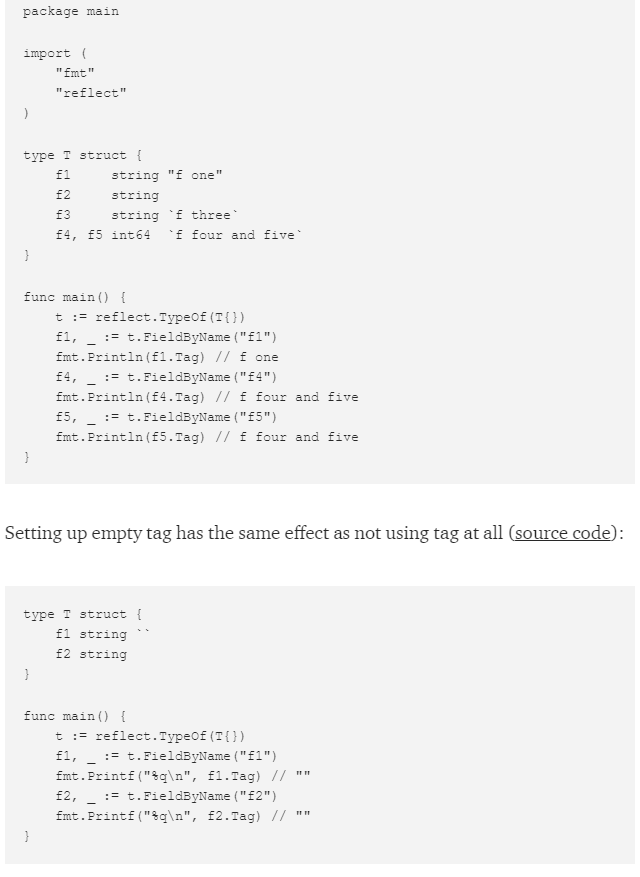


**Tags**

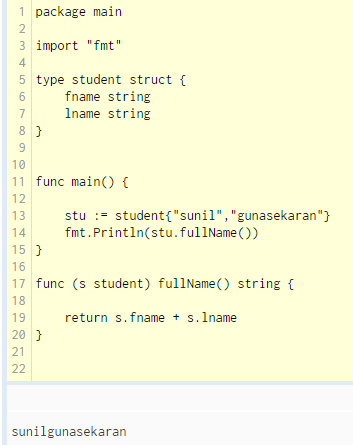


**Reflection**

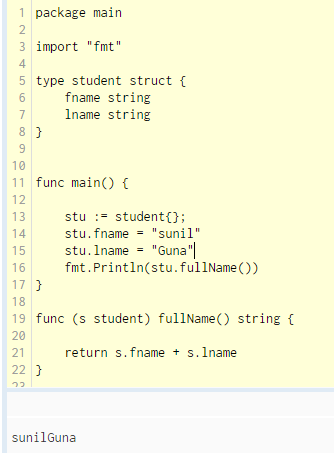
* Tags are accessible through *reflect* package which allows run-time [reflection](https://en.wikipedia.org/wiki/Reflection_%28computer_programming%29)



**Function receiver usage as Object**

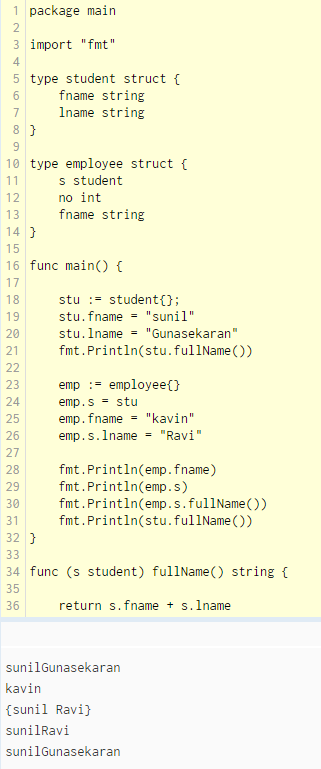


**Another approach**



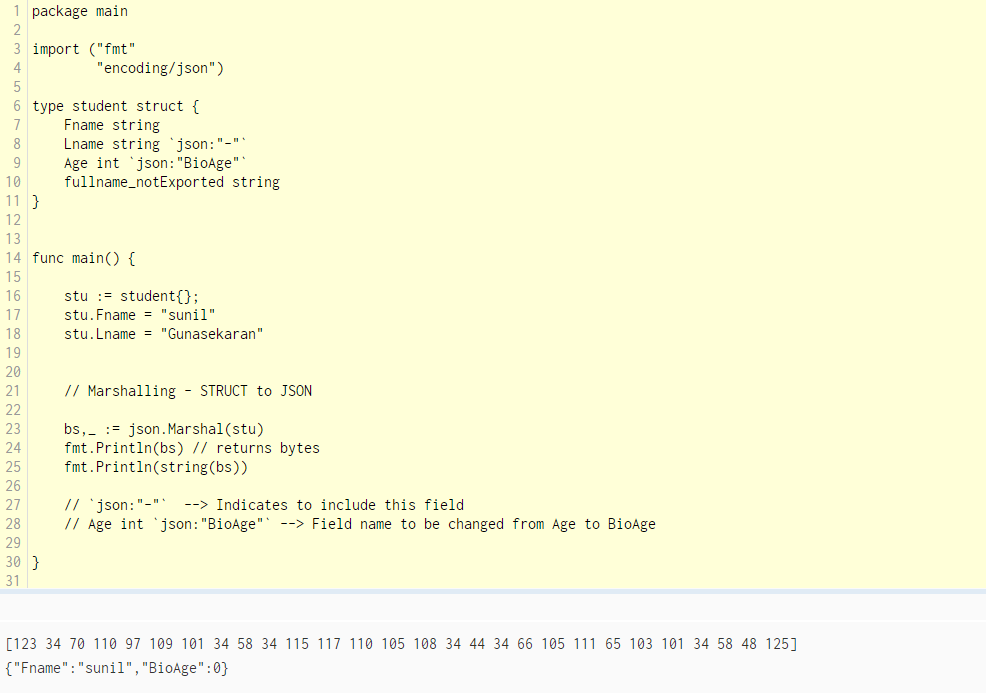
**Overriding Example**

* Using type student inside employee type ( Inheritance)



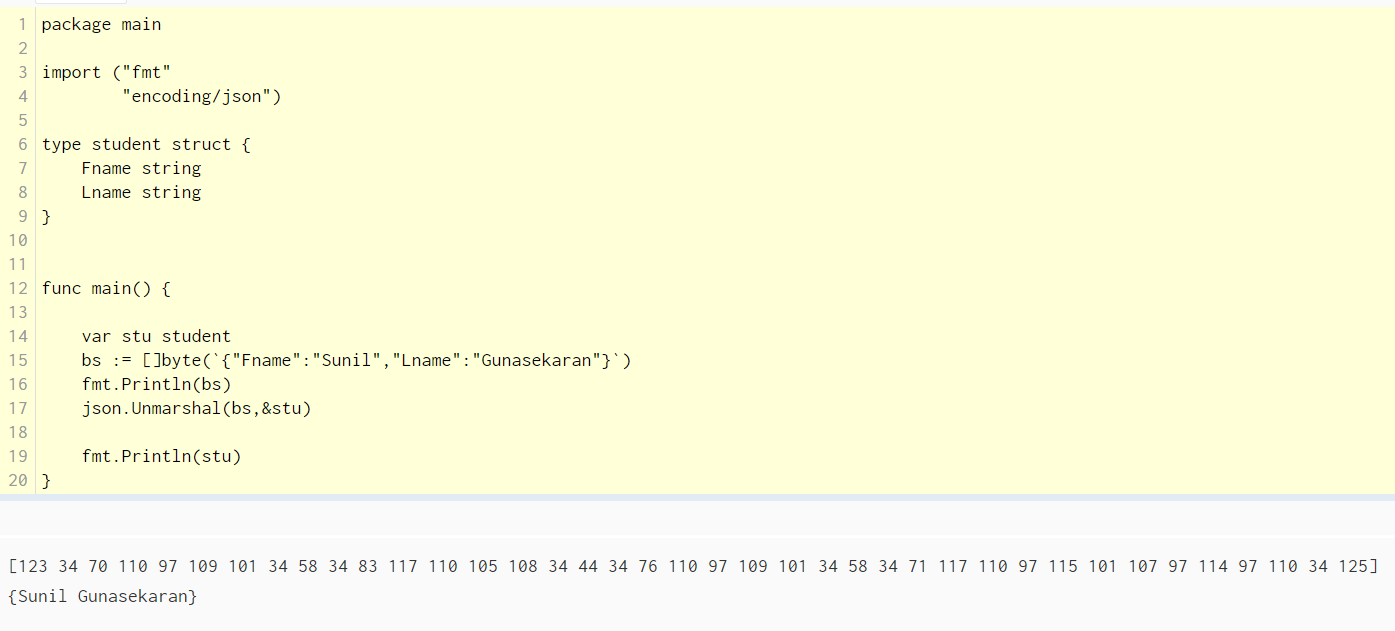
**JSON Marshal**

* Deals with bytes



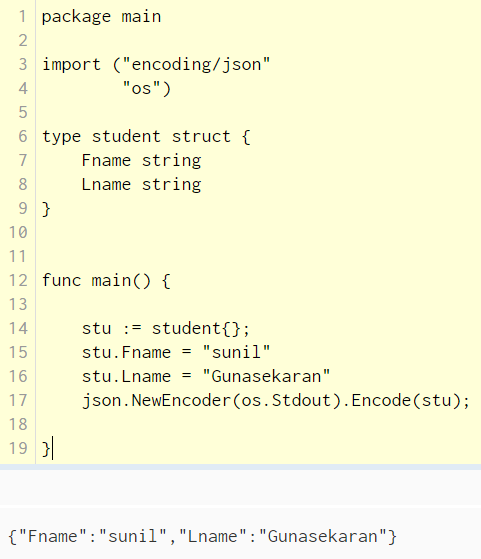
**JSON UnMarshal**

* Deals with bytes

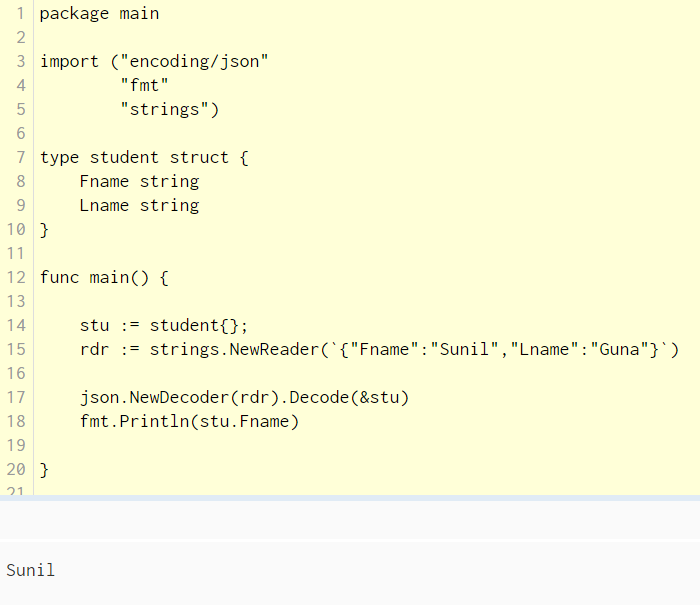


**JSON Encode**

* Deals with Streams

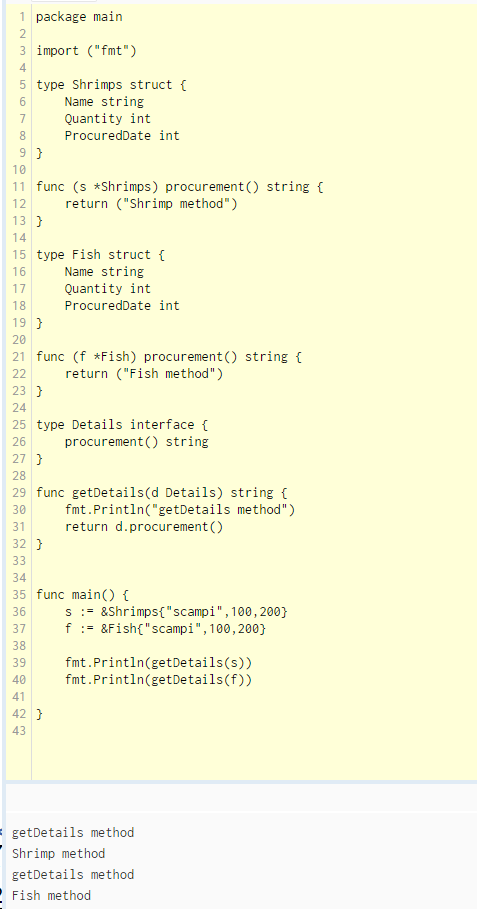


**JSON Decode**



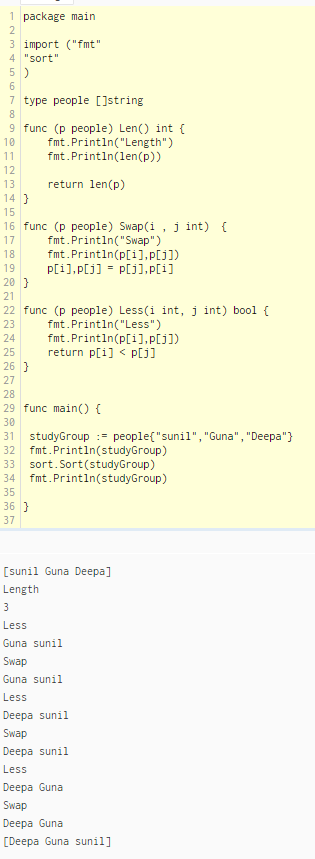
**Interfaces**

* Create an interface type and create respective methods. Here it is procurement
* & to be used if the receiver is declared as pointer to struct
* Define a method based on struct ( Ex: procurement method is associated with struct Shrimps)
* Define one more method based on struct ( Ex: procurement method is associated with struct Fish) 🡪 Not mandatory
* Create a custom method for the interface 🡪 Ex : getDetails()

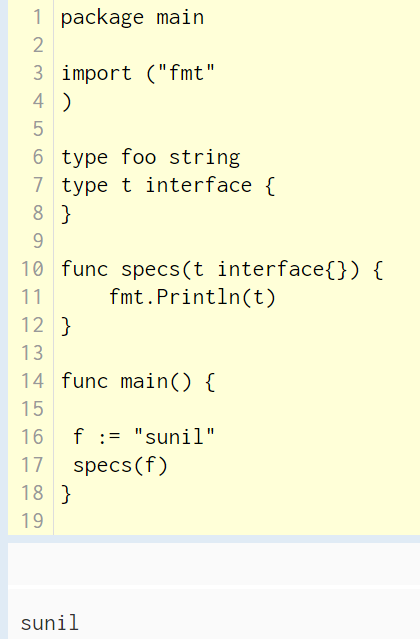


**Important Packages**

**Sort Package**



**Empty Interface**



**Method Sets**

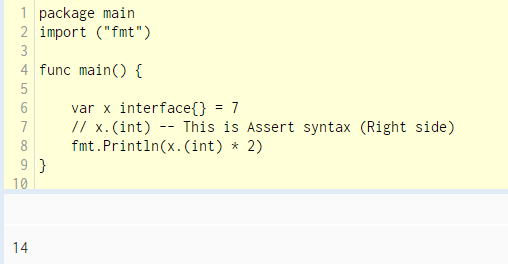
* Method attached to types are called method sets
* Value receiver 🡪 Either value or address can be passed
* Pointer receiver 🡪 address can only be passed



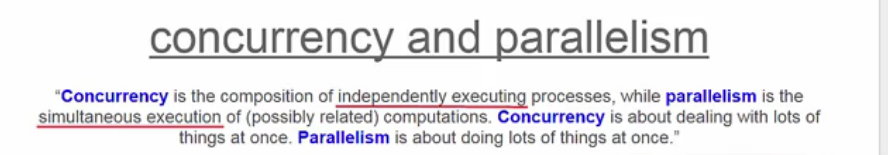
**Conversion**

* strconv
* parseInt
* Type casting are all available

**Assertion – Only used in Interfaces**



**Concurrency Vs. Parallelism**



* After go version 1.5, system automatically chooses the number of cores. Previously number of cores needs to specified explicitly

**Init Function**

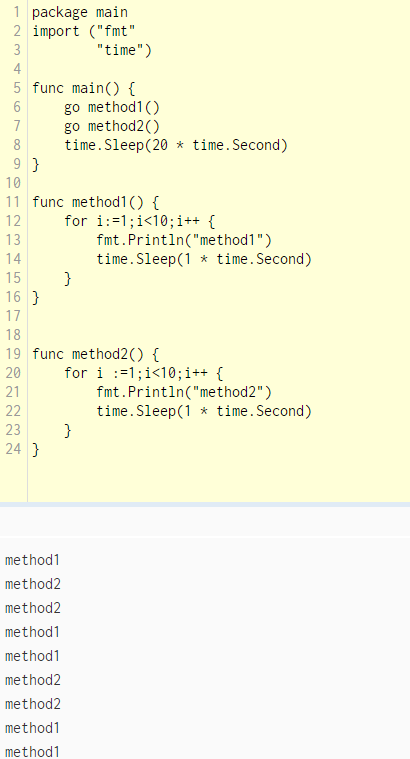


**Channels**

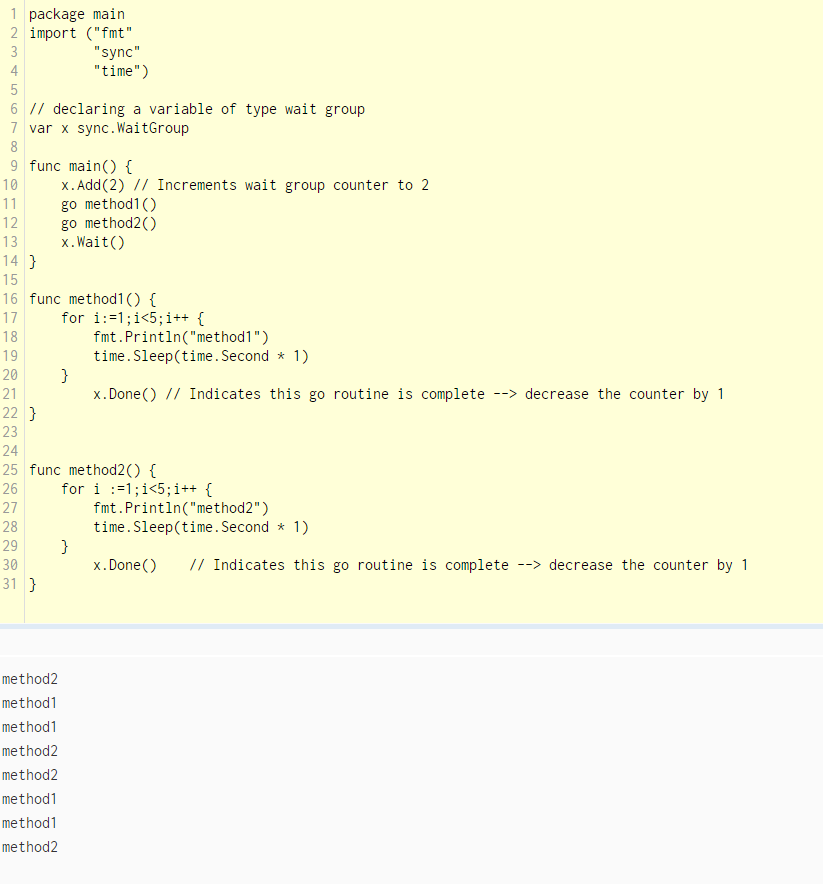
* Channels are used to communicate between go routines
* It is more of relay race. One go routine sends data, other routine waits for the data and executes
* When go routines needs to communicate, it doesn’t require like mutex etc. to lock the variable so that the other routines won’t access. Instead pass on the variables to other go routines instead of sharing (which typically happens in C++)

**Sample go routine**

* Using Sleep so that we can visibly notice interleaving



**Go routine using wait group**

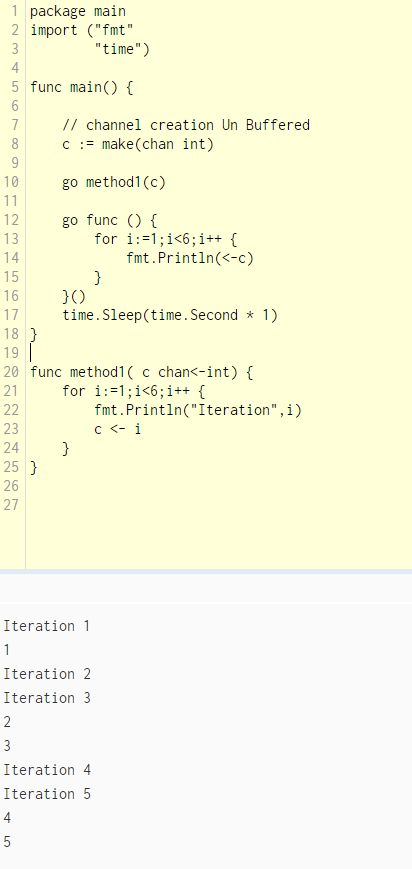


**Channel Creation**

* Buffered Channel 🡪 c := make( chan int,20)
* Un-Buffered Channel 🡪 c := make( chan int)

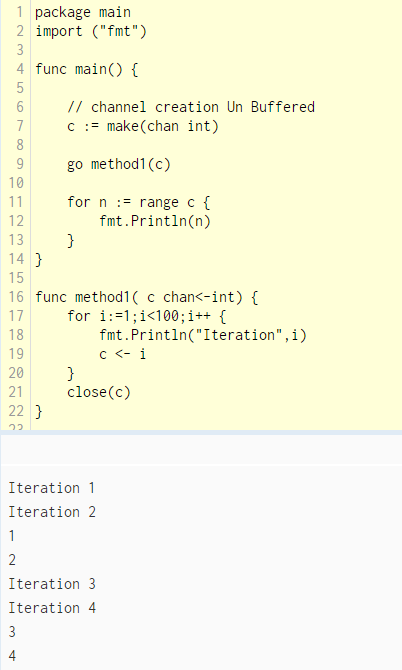
Here channel ‘c’ is set with values of ‘i’. Until ‘c’ is read , other go routine cannot update the value of c and it won’t proceed.

Code method1() writes into c and stops and waits for func() to read 🡪 BLOCKING NATURE



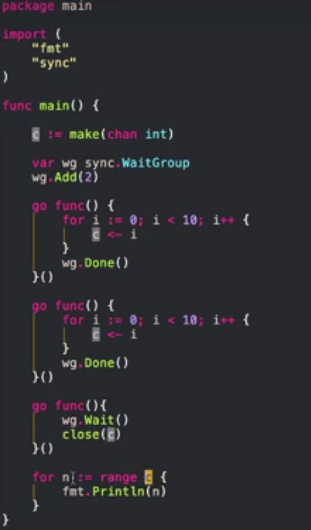
**Using Range in Go Routines**

* Range can work for all the values of channel ‘c’ (auto blocking)
* Waits for close() . Once closed, channel cannot take any more values



**Many go routines writing into channel but one is receiving ( Many to 1 )**

* Used wait group in go routines to make sure they execute completely before exiting



**Many to 1 without waiting group( Semaphore approach)**

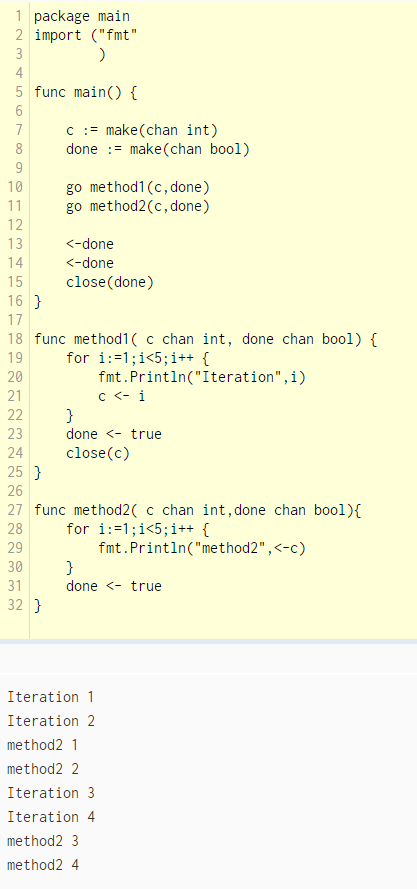
🡨 done waits on true



**1 Channel – Received by Many**



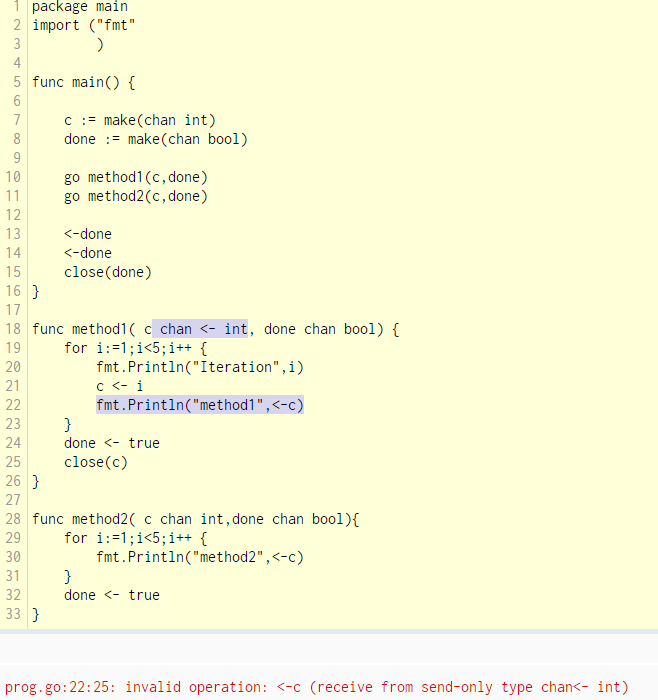
**Channel Passing and Returning arguments**

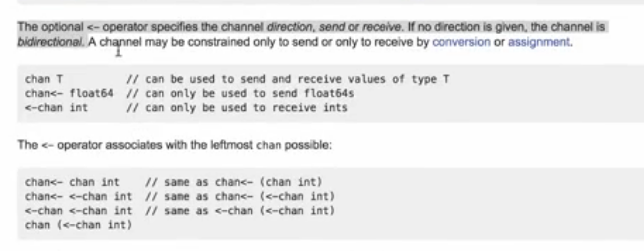




**Channel Directions**

* c := make(chan int)
* <- int – This indicates it can only write values of type int
* <- This indicates it can only receive values
* If nothing is mentioned, it can both send and receive values





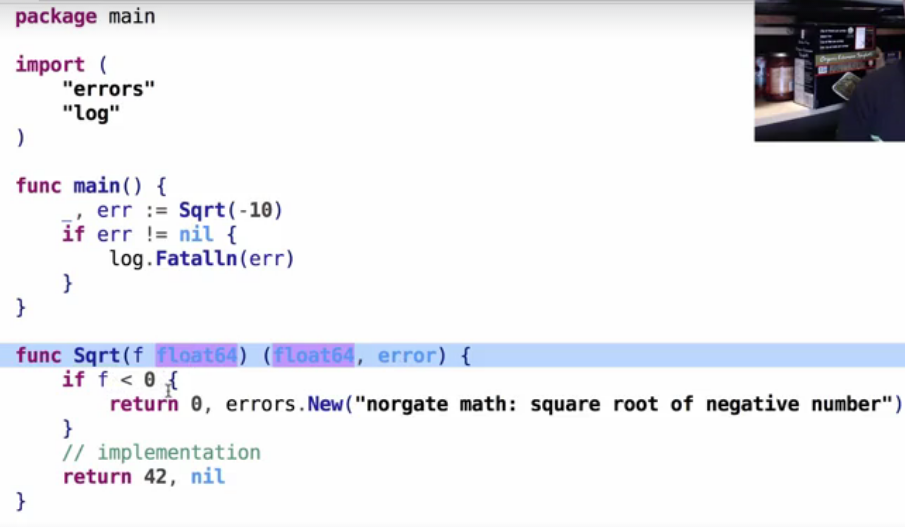
**Error Handling**

* Go has no exceptions

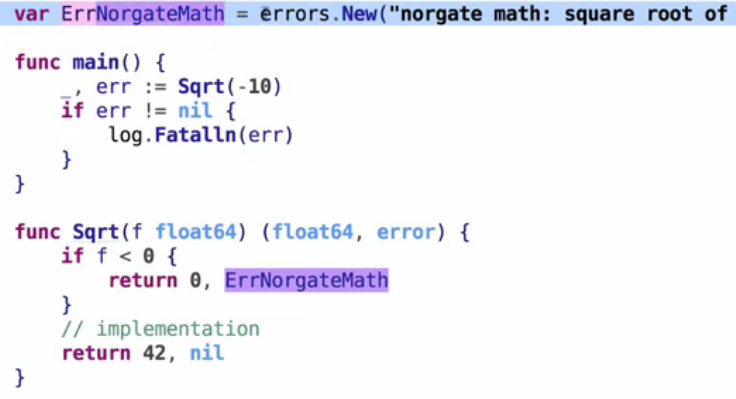
**Different ways**

* log.Println(err)
* fmt.Println(err)
* log.Fatalln(err) 🡪 Writes the error message into log and then invokes OS system.exit return code 1
* panic(err) 🡪 Exit code 2 . Also you get an additional stack trace information [This is builtin package]

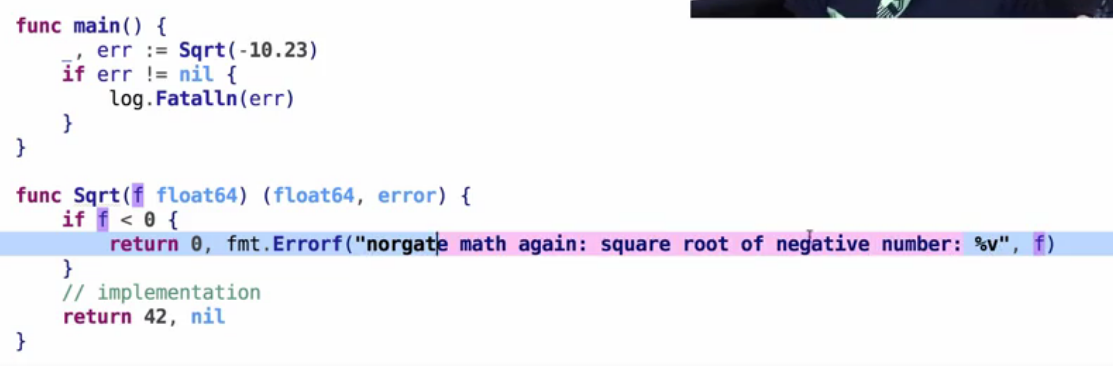
**Creating error – Return error**



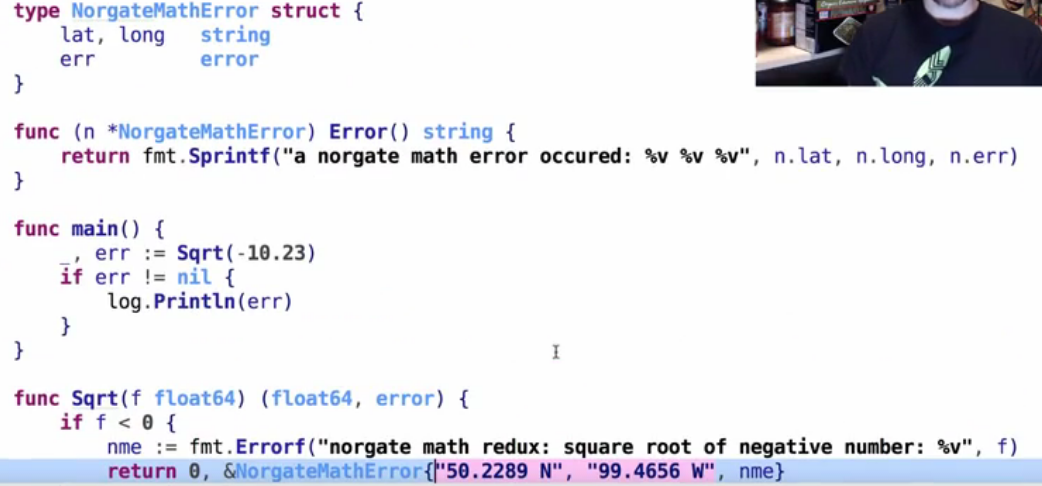
**Typical way of having all error messages in a single file**



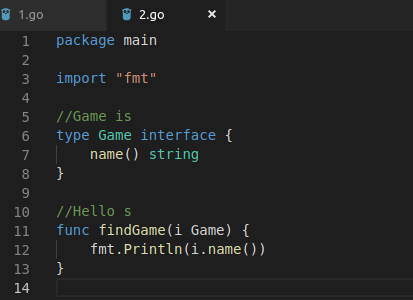
**Log messages with arguments**

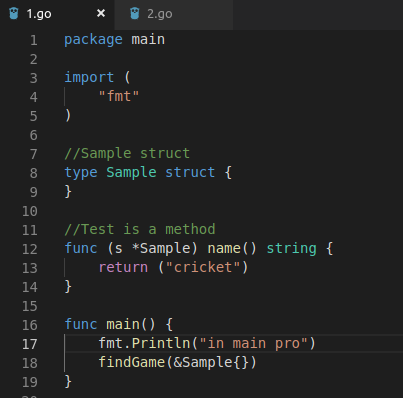


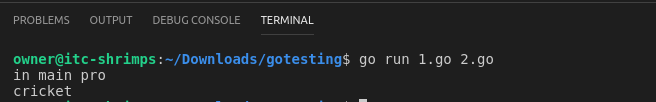
**Customize Error Message using Struct**



**Interface implementation**







**Utilities**

