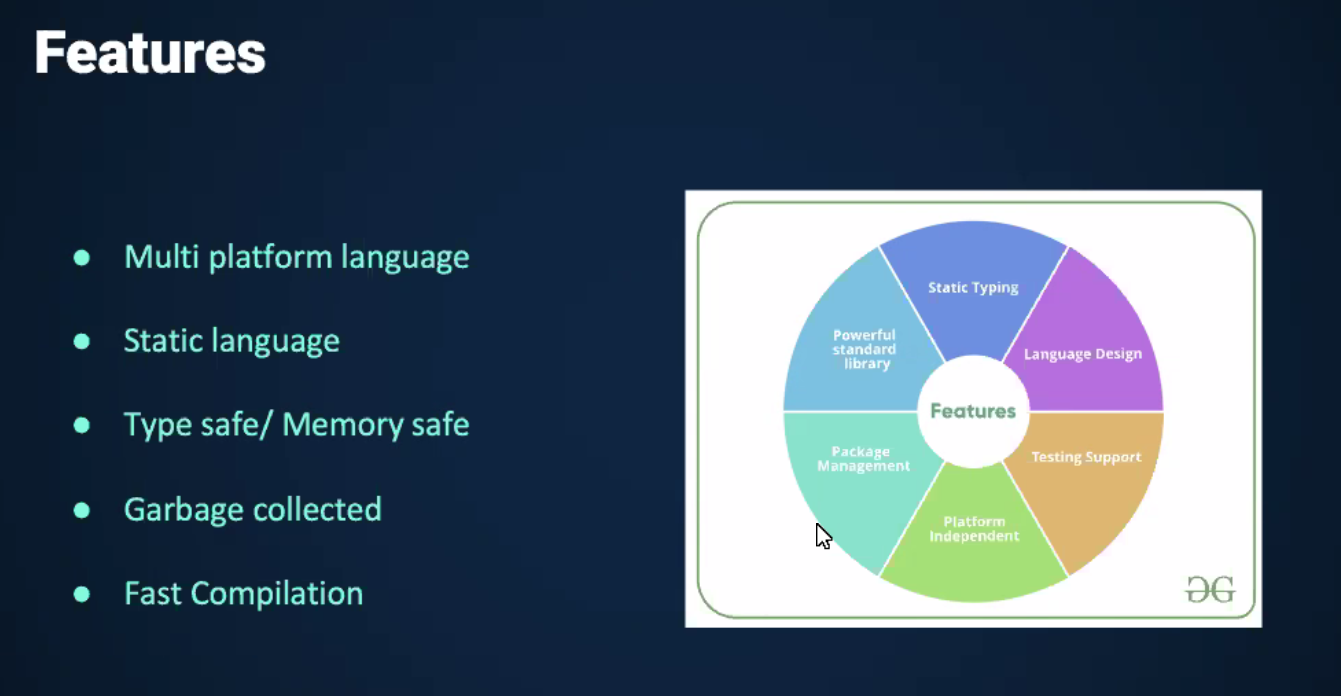
**Go lang training notes**

Go is explicitly made to be faster and not simpler

Go is best in concurrency and microservices.

Go is compiled language (not interpreted language)



* Go is cross platform, without worrying about the platform (windows/linux etc)
* It is statically typed: variables will have strict types
* Fast Compilation
* Go supports object oriented programming paradigm.
* Should the package name match with directory name: NO (not mandatory)
* When I import a package, its source code is made available to our package.
* Garbage collection:
  + What is [Garbage collection (computer science) - Wikipedia](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)) (example: [Garbage Collection in Java - GeeksforGeeks](https://www.geeksforgeeks.org/garbage-collection-java/))
  + Go has automatic garbage collection. It will automatically handle orphaned processes.

==============================================================

**Starting go lang with hello world program**

1. Create a folder (folder name say gorepo)
2. go mod init (module\_name) : this mod file will be used to handle library dependencies
3. Open this folder in vscode, start coding.

package main

import "fmt"

func main(){

fmt.Println("hello world")

}

To run the program:

>>go run hello.go

To build executable:

>>go build hello.go

**Go tools:**

****

**Cross compiling features golang:**

We can compile golang code for different platform while being working on a different platform:

For this we use runtime package

| package main |  |
| --- | --- |
|  |  |
|  | import ( |
|  | "fmt" |
|  | "runtime" |
|  | ) |
|  |  |
|  | func main(){ |
|  | fmt.Printf("OS: %s\nArchitecture: %s\n", runtime.GOOS, runtime.GOARCH) |
|  | } |

## Variable Declaration:

Var keyword for declaring variables

>> var a int

>> var b float

Var can be used to declare multiple variables together

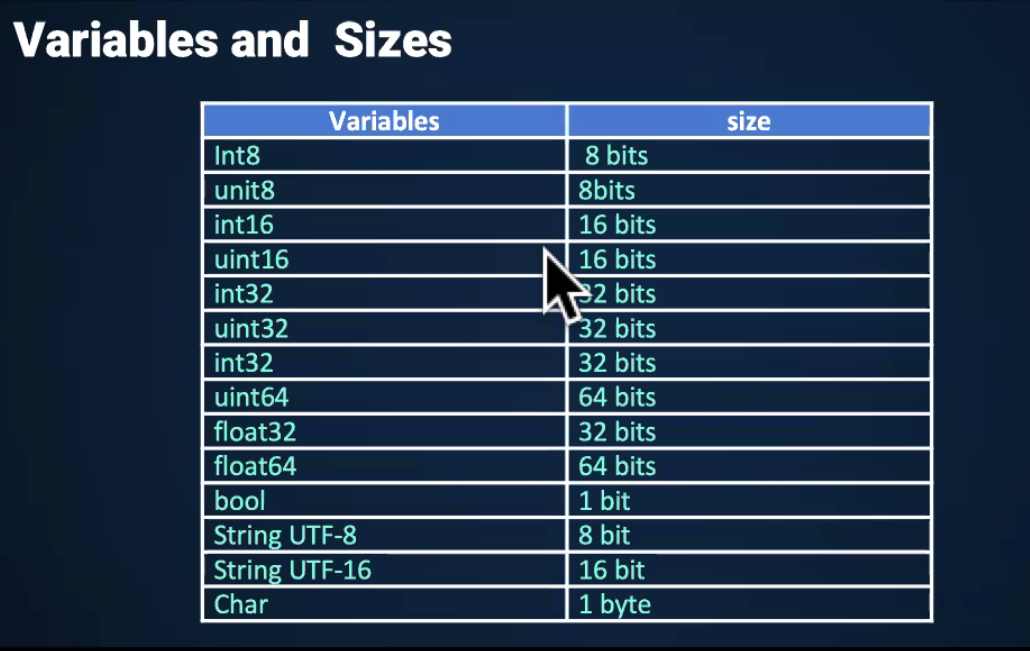
>> var c,d int

Multiple variables initialized

>> var d,e int = 1,2

Without var declaration+initialization

>>f:=123



Int8 ⇒ range(-2^7 to 2^7)

Uint8 ==> range(0 to 2^8)

**To find out**: How float32 is stored: → how many bits before floating point and how many after floating point.

## Enumeration:

\*Iota can be used to make the sequence in enumeration

Example

package main

import (

"fmt"

)

func main() {

const(

a=iota

b

c

)

fmt.Println(a)

fmt.Println(b)

fmt.Println(c)

}

**Output**:

0

1

2

## 

## Fmt:

Sprintf:

The fmt.Sprintf() function in Go language formats according to a format specifier and returns the resulting string.

[fmt.Sprintf() Function in Golang With Examples - GeeksforGeeks](https://www.geeksforgeeks.org/fmt-sprintf-function-in-golang-with-examples/)

fmt.Printf(‘%T’,variable) ⇒ prints type of variable

## 

## Aliases:

>>import s “string” // this imports string package as “s”

>>var p = fmt.Println // Now p can be used to print

p(“hello”)

Output: hello

## String package:

<https://www.geeksforgeeks.org/string-package-in-golang>/

## Struct:

package main

import (

"fmt"

)

type point struct{

x,y int //notice that var keyword is not needed here.

}

func main(){

p:=point{1,2}

fmt.Printf("%v\n",p) //if the value is struct , %v will print all values

}

Output:

{1 2}

Anonymous structures

When a structure is a member of another structure.

Example:

package main

import "fmt"

type Kitchen struct {

numOfPlates int

}

type House struct {

Kitchen //anonymous field

numOfRooms int

}

func main() {

h := House{Kitchen{10}, 3} //to initialize you have to use composed type name.

fmt.Println("House h has this many rooms:", h.numOfRooms) //numOfRooms is a field of House

fmt.Println("House h has this many plates:", h.numOfPlates) //numOfPlates is a field of anonymous field Kitchen, so it can be referred to like a field of House

fmt.Println("The Kitchen contents of this house are:", h.Kitchen) //we can refer to the embedded struct in its entirety by referring to the name of the struct type

}

Output:

House h has this many rooms: 3

House h has this many plates: 10

The Kitchen contents of this house are: {10}

REFLECTION IN GOLANG

<https://www.geeksforgeeks.org/reflection-in-golang/>

Methods in structs can only be created through receivables

In golang, we cannot declare/define methods inside structure.

Receivables are of two types: received by pointer , received by value

Received by pointer allows to edit contents of structure.

//Go supports methods defined on struct types.

package main

import "fmt"

type rect struct {

width, height int

}

//This area method has a receiver type of \*rect.

func (r \*rect) area() int {

return r.width \* r.height

}

//Methods can be defined for either pointer or value receiver types. Here’s an example of a value receiver.

func (r rect) perim() int {

return 2\*r.width + 2\*r.height

}

func main() {

r := rect{width: 10, height: 5}

//Here we call the 2 methods defined for our struct.

fmt.Println("area: ", r.area())

fmt.Println("perim:", r.perim())

//Go automatically handles conversion between values and pointers for method calls. You may want to use a pointer receiver type to avoid copying on method calls or to allow the method to mutate the receiving struct.

rp := &r

fmt.Println("area: ", rp.area())

fmt.Println("perim:", rp.perim())

}

Output:

area: 50

perim: 30

area: 50

perim: 30

## 

## Array vs variables:

Default values are 0 for integer array.

Arrays are created in heap memory, variables are created in stack memory.

package main

import (

"fmt"

)

type point struct{

x,y int

}

func main(){

x := [5]int{1,2,3,4,5}

fmt.Println(x)

fmt.Printf("%v\n",x))

}

Output:

[1 2 3 4 5]

[1 2 3 4 5]

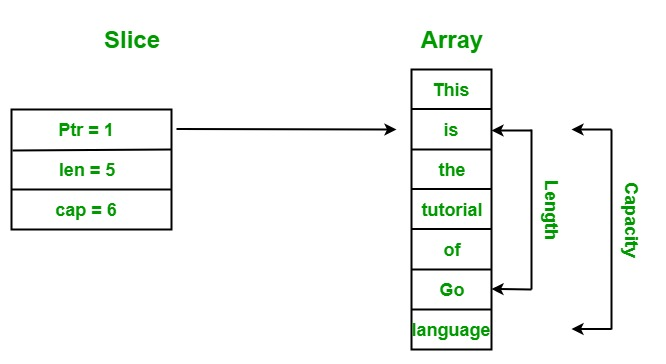
**To read**: golang unsafe package (allows pointer manipulation and playing with memory addresses)

Q1: Can we have a dynamic array in go?

Read: <https://medium.com/gojekengineering/grab-a-slice-on-the-go-c606344186c1>

## Arrays vs Slices

Arrays need size and type at the time of declaration.



A slice contains three components:

**Pointer**: The pointer is used to points to the first element of the array that is accessible through the slice. Here, it is not necessary that the pointed element is the first element of the array.

**Length**: The length is the total number of elements present in the array.

**Capacity**: The capacity represents the maximum size upto which it can expand.

[Slices in Golang - GeeksforGeeks](https://www.geeksforgeeks.org/slices-in-golang/)

[Go Slices: usage and internals - go.dev](https://go.dev/blog/slices-intro)

Copying one slice into another slice:

1. Copy keyword [How to copy one slice into another slice in Golang? - GeeksforGeeks](https://www.geeksforgeeks.org/how-to-copy-one-slice-into-another-slice-in-golang/)
2. assignment

## Changes in capacity and length of slice with addition of more elements to slice

package main

import (

"fmt"

)

func main() {

a:=[6]int{1,2,3,4,5,6}

s1:=a[1:5]

s2:=a[:4]

s1 = append(s1, 2)

fmt.Println("slice1:",s1)

fmt.Println("")

fmt.Println("length of slice1:",len(s1))

fmt.Println("")

fmt.Println("capacity of slice1:",cap(s1))

fmt.Println("")

s2 = append(s1,s2...) // 3 dot notation explodes the slice and puts all elements as arguments

fmt.Println("slice2:",s2)

fmt.Println("")

fmt.Println("length of slice2:",len(s2))

fmt.Println("")

fmt.Println("capacity of slice2:",cap(s2))

}

<https://play.golang.org/p/88olIDnkrub>

Output:

slice1: [2 3 4 5 2]

length of slice1: 5

capacity of slice1: 5

slice2: [2 3 4 5 2 1 2 3 4]

length of slice2: 9

capacity of slice2: 10

## 

## String Byte Slice

[How to Convert Golang String to Byte Array (appdividend.com)](https://appdividend.com/2020/03/02/golang-how-to-convert-string-to-byte-array-example/)

package main

import (

"fmt"

)

func main() {

str:="MBB$"

data:=[]byte(str)

fmt.Println(data)

}

Output:

[77 66 66 36]

## Maps

[Golang Maps - GeeksforGeeks](https://www.geeksforgeeks.org/golang-maps/)

## Range function

Range function can be used to iterate through maps and slices.

When range function is used with slices, it returns two output params:

“index” and “value at index”

When range function is used with maps, it returns two output params:

“Key” and “value”

## Switch statements

[Switch Statement in Go - GeeksforGeeks](https://www.geeksforgeeks.org/switch-statement-in-go/)

## Errors in golang

Error is a return type in golang: we can define functions to return more than one value, one of them can be of type error

## Defer statements:

[Defer Keyword in Golang - GeeksforGeeks](https://www.geeksforgeeks.org/defer-keyword-in-golang/)

Defer statements are used to change order of execution of functions. Function called with defer keyword will be called after the code below it is executed.

package main

import "fmt"

func add(a1,a2 int)int{

res:=a1+a2

fmt.Println("result:",res)

return res

}

func main() {

defer fmt.Println("End")

add(34,56)

}

Output:

result: 90

End

------------------------------------------------------------------------

## Multiple defer statements:

(executes in LIFO order)

package main

import "fmt"

func add(a1,a2 int)int{

res:=a1+a2

fmt.Println("result:",res)

return res

}

func main() {

defer fmt.Println("End")

defer fmt.Println("about to end")

add(34,56)

}

Output:

result: 90

about to end

End

* We cannot defer a panic. A panic will always be run immediately.

## 

## 

## 

## 

## Recover function:

[Recover in Golang - GeeksforGeeks](https://www.geeksforgeeks.org/recover-in-golang/)

Panic statement will immediately stop the function. However, if we want our complete program to run irrespective of panic condition, we use recover function.

package main

import (

"fmt"

)

func recoverFullName() {

if r := recover(); r != nil {

fmt.Println("recovered from ", r)

}

}

func fullName(firstName \*string, lastName \*string) {

defer recoverFullName()

if firstName == nil {

panic("runtime error: first name cannot be nil")

}

if lastName == nil {

panic("runtime error: last name cannot be nil")

}

fmt.Printf("%s %s\n", \*firstName, \*lastName)

fmt.Println("returned normally from fullName")

}

func main() {

defer fmt.Println("deferred call in main")

firstName := "Elon"

fullName(&firstName, nil)

fmt.Println("returned normally from main")

}

## Functions

Private and public functions:

First letter: Capital (public/exported)

First letter: small (private)

Functions are started using func keyword.

package main

import (

"fmt"

)

func vals() (int,int){

return 3,7

}

func main() {

a,b:=vals()

\_,c:=vals()

fmt.Println(a,b,c)

}

Output:

3 7 7

package main

import (

"fmt"

)

func plus(a,b,c int,float32,float32) float32{ //not allowed

return b+c

}

func main() {

fmt.Println(plus(1,2,3))

}

Output:

syntax error: mixed named and unnamed parameters

===================================================

package main

import (

"fmt"

)

func plus(a,b,c int) int{

return a+b+c

}

func main() {

fmt.Println(plus(1,2,3))

}

Output:

6

==========================================

* Functions can take arbitrary number of values as input parameters
  + These are called variatic functions

Example:

>>

package main

import (

"fmt"

)

func sum(nums ...int) {

fmt.Println(nums, " ")

total := 0

for \_, num := range nums {

total += num

}

fmt.Println(total)

}

func main() {

sum(1, 2)

sum(1, 2, 3)

nums := []int{1, 2, 3, 4}

sum(nums...)

}

Output:

[1 2]

3

[1 2 3]

6

[1 2 3 4]

10

========================================

package main

import (

"fmt"

)

func sum(nums ...int) {

fmt.Println(nums, " ")

total := 0

for \_, num := range nums {

total += num

}

fmt.Println(total)

}

func main() {

sum(1, 2)

sum(1, 2, 3)

nums := []int{1, 2, 3, 4}

fmt.Println(nums)

sum(nums...)

}

<https://play.golang.org/p/0CqgWIg_Oi1>

Three dotted notation in golang: [How to use Ellipsis (...) in Golang? - GeeksforGeeks](https://www.geeksforgeeks.org/how-to-use-ellipsis-in-golang/)

**Closures in functions**

package main

import "fmt"

//This function intSeq returns another function, which we define anonymously in the body of intSeq. The returned function closes over the variable i to form a closure.

func intSeq() func() int {

i := 0

return func() int {

i++

return i

}

}

func main() {

//We call intSeq, assigning the result (a function) to nextInt. This function value captures its own i value, which will be updated each time we call nextInt.

nextInt := intSeq()

//See the effect of the closure by calling nextInt a few times.

fmt.Println(nextInt())

fmt.Println(nextInt())

fmt.Println(nextInt())

newInts := intSeq()

fmt.Println(newInts())

}

//To confirm that the state is unique to that particular function, create and test a new one.

**Closures** maintain the stack frame of the anonymous function until it is garbage collected by go.

<https://play.golang.org/p/XcyrRjSO9Ai>

**Methods in Go**

Methods in go is basically a function with some type(can be struct or ant type)

| //This area method has a receiver type of \*rect. |  |
| --- | --- |
|  | func (r \*rect) area() int { |
|  | return r.width \* r.height |
|  | } |
|  |  |
|  | //Methods can be defined for either pointer or value receiver types. Here’s an example of a value receiver. |
|  | func (r rect) perim() int { |
|  | return 2\*r.width + 2\*r.height |
|  | } |

Go automatically handles conversion between values and pointers for method calls. You may want to use a pointer receiver type to avoid copying on method calls or to allow the method to mutate the receiving struct.

Pointers are always reduced to object.

**Interfaces**

Complete abstract classes in which only function declarations are present.

//Here’s a basic interface for geometric shapes.

type geometry interface {

area() float64

perim() float64

}

To implement an interface in Go, we just need to implement all the methods in the interface.

If a variable has an interface type, then we can call methods that are in the named interface.

Example:

//Interfaces are named collections of method signatures.

package main

import "fmt"

import "math"

//Here’s a basic interface for geometric shapes.

type geometry interface {

area() float64

perim() float64

}

//For our example we’ll implement this interface on rect and circle types.

type rect struct {

width, height float64

}

type circle struct {

radius float64

}

//To implement an interface in Go, we just need to implement all the methods in the interface. Here we implement geometry on rects.

func (r rect) area() float64 {

return r.width \* r.height

}

func (r rect) perim() float64 {

return 2\*r.width + 2\*r.height

}

//The implementation for circles.

func (c circle) area() float64 {

return math.Pi \* c.radius \* c.radius

}

func (c circle) perim() float64 {

return 2 \* math.Pi \* c.radius

}

//If a variable has an interface type, then we can call methods that are in the named interface. Here’s a generic measure function taking advantage of this to work on any geometry.

func measure(g geometry) {

fmt.Println(g)

fmt.Println(g.area())

fmt.Println(g.perim())

}

func main() {

r := rect{width: 3, height: 4}

c := circle{radius: 5}

//The circle and rect struct types both implement the geometry interface so we can use instances of these structs as arguments to measure.

measure(r)

measure(c)

}

Output:

{3 4}

12

14

{5}

78.53981633974483

31.41592653589793

The basic difference between structs and interface is that structs are basically collection of data types where as interfaces are collection of methods signature

(declaration of function).

Composition in interfaces(can be supported by structs too)

There can be interfaces containing interfaces.

Interface that does not have any method-Marker Interface

Type-Switching

Empty Interface can takes any kind of value like int,string ,float etc.

All primitive type implements empty interface so that we can use it more efficiently

## Advanced data structures libraries in golang:

Stack implementation in Golang

https://stackoverflow.com/questions/28541609/looking-for-reasonable-stack-implementation-in-golang

Queue Implementation in Golang

https://stackoverflow.com/questions/3042329/how-to-implement-a-queue-in-go

Binary Tree Implementation in Golang

https://www.golangprograms.com/golang-program-to-implement-binary-tree.html

## Memory management and why strings are immutable?

Since String is the most widely used data structure, improving the performance of String have a considerable effect on improving the performance of the whole application in general.

Being immutable automatically makes the String thread safe since they won't be changed when accessed from multiple threads.

Hence immutable objects, in general, can be shared across multiple threads running simultaneously. They're also thread-safe because if a thread changes the value, then instead of modifying the same, a new String would be created in the String pool. Hence, Strings are safe for multi-threading

1. Performance

The String is the most widely used data structure. Caching the String literals and reusing them saves a lot of heap space because different String variables refer to the same object in the String pool. String intern pool serves exactly this purpose.

Because of the presence of the String pool, two different variables may point to same String object from the pool, thus saving crucial memory resource.

String pool exists because Strings are immutable. In turn, it enhances the performance by saving heap memory and faster access of hash implementations when operated with Strings.

2. Hashcode Caching

Since String objects are abundantly used as a data structure, they are also widely used in hash implementations like HashMap, HashTable, HashSet, etc. When operating upon these hash implementations, hashCode() method is called quite frequently for bucketing.

The immutability guarantees Strings that their value won’t change. So the hashCode() method is overridden in String class to facilitate caching, such that the hash is calculated and cached during the first hashCode() call and the same value is returned ever since.

This, in turn, improves the performance of collections that uses hash implementations when operated with String objects.

On the other hand, mutable Strings would produce two different hashcodes at the time of insertion and retrieval if contents of String was modified after the operation, potentially losing the value object in the Map.

3. Security

The String is widely used in Golang applications to store sensitive pieces of information like usernames, passwords, connection URLs, network connections, etc.

Hence securing String class is crucial regarding the security of the whole application in general.

We're doing all necessary security checks initially to check if the String is only alphanumeric, followed by some more operations.

If Strings were mutable, then by the time we execute the update, we can't be sure that the String we received, even after performing security checks, would be safe. The untrustworthy caller method still has the reference and can change the String between integrity checks. Thus making our query prone to SQL injections in this case. So mutable Strings could lead to degradation of security over time.

In general, immutability comes to our rescue in this case because it's easier to operate with sensitive code when values don't change because there are fewer interleavings of operations that might affect the result.

Pointer arithmetic is not supported by golang naturally. Golang unsafe library can be used for this .

Assignment:

Implement queue,stack and tree in golang