**1) Explain what is GO?**

GO is an open source programming language which makes it easy to build simple, reliable and efficient(hiệu quả )software. Programs are constructed from packages, whose properties allow efficient management of dependencies.

**4) Explain packages in Go program?**

Every GO program is made up of packages.  The program starts running in package **main.**This program is using the packages with import paths **“fmt”** and **“math/rand”.**

**5) Explain workspace in GO?**

Inside a workspace GO code must be kept.  A workspace is a directory hierarchy with three directories at its root.

* src contains GO source files organized into packages
* pkg contains package objects and
* bin contains executable commands

**7) Explain what is GOPATH environment variable?**

* The GOPATH environment variable determines the location of the workspace. It is the only environment variable that you have to set when developing Go code.

**10) What are the advantages of GO?**

* GO compiles very quickly
* Go supports concurrency at the language level
* Functions are first class objects in GO
* GO has garbage collection
* Strings and Maps are built into the language

**12) Explain what is go routine in GO? How you can stop go routine?**

* A goroutine is a function which is capable of running concurrently with other functions
* To stop goroutine, you pass the goroutine  a signal channel, that signal channel is used to push a value into when you want the goroutine to stop.  The goroutine polls that channel regularly as soon as it detects a signal, it quits.( Để dừng goroutine, bạn truyền cho goroutine một kênh tín hiệu, kênh tín hiệu đó được sử dụng để đẩy một giá trị vào khi bạn muốn goroutine dừng lại. Các cuộc thăm dò goroutine kênh thường xuyên ngay khi phát hiện tín hiệu, nó thoát ra.)

**Explain how pointer is represented in GO?**

In GO a pointer is represented by using the \* (asterisk) character followed by the type of the stored value.

Allow you to pass a reference to an object, rather than the object itself. Sometimes called “passing by reference”

**18) Explain GO Interfaces ?**

In GO, interfaces is a way to specify the behaviour of an object.  An interface is created by using the **“type”**word, followed by a name and the keyword interface.  An interface is specified as two things.

* A set of methods
* Also it is referred as type

**Question 47. How To Create A Map In Go?**

**Answer :**

You must use make function to create a map.

/\* declare a variable, by default map will be nil\*/  
var map\_variable map[key\_data\_type]value\_data\_type  
/\* define the map as nil map can not be assigned any value\*/  
map\_variable = make(map[key\_data\_type]value\_data\_type)

**Question 46. What Are Maps In Go?**

**Answer :**

Go provides another important data type map which maps unique keys to values. A key is an object that you use to retrieve a value at a later date. Given a key and a value, you can strore the value in a Map object. After value is stored, you can retrieve it by using its key.

**Question 45. What Is Range In Go?**

**Answer :**

The range keyword is used in for loop to iterate over items of an array, slice, channel or map. With array and slices, it returns the index of the item as integer. With maps, it returns the key of the next key-value pair.

**Question 43. What Is The Difference Between Len() And Cap() Functions Of Slice In Go?**

**Answer :**

len() function returns the elements presents in the slice where cap() function returns the capacity of slice as how many elements it can be accomodate.

**Question 41. How To Define A Slice In Go?**

**Answer :**

To define a slice, you can declare it as an array without specifying size or use make function to create the one.

var numbers []int /\* a slice of unspecified size \*/  
/\* numbers == []int{0,0,0,0,0}\*/  
numbers = make([]int,5,5) /\* a slice of length 5 and capacity 5\*/

**Question 40. What Is Slice In Go?**

**Answer :**

Go Slice is an abstraction over Go Array. As Go Array allows you to define type of variables that can hold several data items of the same kind but it do not provide any inbuilt method to increase size of it dynamically or get a sub-array of its own. Slices covers this limitation. It provides many utility functions required on Array and is widely used in Go programming.

**Question 39. How To Define A Structure In Go?**

**Answer :**

To define a structure, you must use type and struct statements. The struct statement defibes a new data type, with nore than one member for your program. Type statement binds a name with the type which is struct in our case

**Question 38. What Is Structure In Go?**

**Answer :**

Struct is a user-defined type that contain a collection of name fiel/properties. It used to group related data together to from a single unit

**Question 37. What Is A Pointer On Pointer?**

**Answer :**

It's a pointer variable which can hold the address of another pointer variable. It de-refers twice to point to the data held by the designated pointer variable.

**Question 35. What Is An Array?**

**Answer :**

Array is collection of similar data items under a common name.

**Question 36. What Is A Nil Pointers In Go?**

**Answer :**

Go compiler assign a Nil value to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned nil is called a nil pointer. The nil pointer is a constant with a value of zero defined in several standard libraries.

**Question 23. What Are The Function Closures?**

**Answer :**

Functions closure are anonymous functions and can be used in dynamic programming.

**Question 24. What Are Methods In Go?**

**Answer :**

Go programming language supports special types of functions called methods. In method declaration syntax, a "receiver" is present to represent the container of the function. This receiver can be used to call function using "." operator.

**Question 17. Explain The Syntax For 'for' Loop?**

**Answer :**

The syntax of a for loop in Go programming language is −

for [condition |  ( init; condition; increment ) | Range]  
{  
   statement(s);  
}

Here is the flow of control in a for loop −

1. if condition is available, then for loop executes as long as condition is true.
2. if for clause that is ( init; condition; increment ) is present then  
     
   The init step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.  
     
   Next, the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.  
     
   After the body of the for loop executes, the flow of control jumps back up to the increment statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.  
     
   The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.
3. if range is available, then for loop executes for each item in the range.

**2. What are goroutines and how are they different from threads?**

For this to answer you should have a good knowledge of goroutines. In short, we can say *Goroutines are a way of doing tasks concurrently in golang.*For more about goroutines and their difference with threads please please read [this](https://medium.com/@riteeksrivastava/a-complete-journey-with-goroutines-8472630c7f5c). After the reading the mentioned article you will have enough content to answer the questions related to goroutines

# **Data structure:** data structure is a data organization, management, and storage format that enables efficient access and modification. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data

#### Reflection goes from interface value to reflection object.

At the basic level, reflection is just a mechanism to examine the type and value pair stored inside an interface variable

pointer, map, slice, channel, or array, you can find out the contained type by using varType.Elem()

struct: reflect.StructField

**Making a New Instance**

If you want to be able to use reflection to modify the value, you have to get a pointer to the variable with refPtrVal := reflect.ValueOf(&var); if you don’t, you can read the value using reflection, but you can’t modify it.

You use refPtrVal.Elem().Set(newRefVal) to make the change, and the value passed into Set() has to be a reflect.Value too.

### **Making Functions**

Reflection doesn’t just let you make new places to store data. You can use reflection to make new functions using the reflect.MakeFunc function. This function expects the reflect.Type for the function that we want to make and a closure whose input parameters are of type []reflect.Value and whose output parameters are also of type []reflect.Value

### **I Want a New Struct**

There’s one more thing that you can make using reflection in Go. You can make brand-new structs at runtime by passing a slice of reflect.StructField instances to the reflect.StructOf function. This one is a bit weird; we are making a new type, but we don’t have a name for it, so you can’t really turn it back into a “normal” variable. You can create a new instance and use Interface() to put the value into a variable of type interface{}, but if you want to set any values on it, you need to use reflection

#### There’s one big limitation on reflection. While you can use reflection to create new functions, there’s no way to create new methods at runtime. This means you cannot use reflection to implement an interface at runtime. It also means that using reflection to make a new struct can break in strange ways. When you create a new struct out of a slice of struct fields, there are some problematic interactions with one of the my favorite features in Go — delegation via anonymous struct fields.

**OOP IN GOLANG**

-Is a p[ramming paradigm](https://en.wikipedia.org/wiki/Programming_paradigm) based on the concept of “[objects](https://en.wikipedia.org/wiki/Object_%28computer_science%29)”, which may contain [data](https://en.wikipedia.org/wiki/Data), in the form of [fields](https://en.wikipedia.org/wiki/Field_%28computer_science%29), often known as attributes; and code, in the form of procedures, often known as [methods](https://en.wikipedia.org/wiki/Method_%28computer_science%29)

### What are the channels?

roc := make(<-chan int)((only recive)  
soc := make(chan<- int)

A **channel** is a communication object using which goroutines can communicate with each other. Technically, a channel is a data transfer pipe where data can be **passed into** or **read from**. Hence one goroutine can send data into a channel, while other goroutines can read that data from the same channel.

### Deadlock

*If you are trying to read data from a channel but channel does not have a value available with it, it blocks the current goroutine and unblocks other in a hope that some goroutine will push a value to the channel. Hence,****this read operation will be blocking****. Similarly, if you are to send data to a channel, it will block current goroutine and unblock others until some goroutine reads the data from it. Hence,****this send operation will be blocking****.*

When channel is closed, value read by the goroutine is zero value of the data type of the channe

#### Buffer size or channel capacity

a channel buffer size is 0 also called as unbuffered channel. Whatever written to the channel is immediately available to read.

#### Unidirectional channels

roc := make(<-chan int)((only recive)  
soc := make(chan<- int)

Using unidirectional channels **increases the type-safety** of a program. Hence the program is less prone to error.)

#### Select

select is just like switch without any input argument but it only used for channel operations. The **select** statement is used to perform an operation on only one channel out of many, conditionally selected by case block.

If all case statements (channel operations) are blocking then select statement will wait until one of the case statement (its channel operation) unblocks and that case will be executed. If some or all of the channel operations are non-blocking, then one of the non-blocking cases will be chosen randomly and executed immediately.

#### default case

Like switch statement, select statement also has default case. **A default case is non-blocking**. But that’s not all, **default case makes selectstatement always non-blocking**. That means, send and receive operation on any channel (buffered or unbuffered) is always non-blocking.

#### WaitGroup

**WaitGroup** is a struct with a counter value which tracks how many goroutines were spawned and how many have completed their job. This counter when reaches zero, means all goroutines have done their job.

#### Mutex

**func** worker(wg \*sync.WaitGroup, m \*sync.Mutex) {  
 m.Lock()  
 i = i + 1  
 m.Unlock()  
 wg.Done()  
}  
  
**func** main() {  
 **var** wg sync.WaitGroup  
 **var** m sync.Mutex  
  
 **for** i := 0; i < 1000; i++ {  
 wg.Add(1)  
 **go** worker(&wg, &m)  
 }

OOP

### 1.enCAPSulation

*“*[*Encapsulation is the mechanism of hiding of data implementation by restricting access to public methods.*](http://crackingjavainterviews.blogspot.com/2013/04/what-are-four-principles-of-oop.html)

### 2. Polymorphism

Polymorphism describes a pattern in object oriented programming in which classes have different functionality while sharing a common interface.

### 3. Composition

[*When we embed a type, the methods of that type become methods of the outer type, but when they are invoked the receiver of the method is the inner type, not the outer one.*](https://golang.org/doc/effective_go.html#embedding)

### Abstraction

-Slice (right removed, lef take):

Unmarshaling:

Types Unmarshaling an object refers to converting JSON into a typed data structure―the reverse of marshaling. Go handles unmarshaling by asking you to create a blank object of a type, and then attempts to apply the JSON string to that object.

Session :

A session can be used to store any information we choose. We need to know how long to keep a session active for, and the name of the cookie that we store it in,

### Why do we need enums?

* Grouping and expecting only some related values
* Sharing common behavior
* Avoids using invalid values
* To increase the code readability and the maintainability

**Buffer**

To avoid the overhead of many small write operations Golang is shipped with bufio.Writer. Data, instead of going straight to destination (implementing io.Writer interface) are first accumulated inside the buffer and send out when buffer is full:

producer buffer destination (io.Writer)  
   
 a -----> a  
 b -----> ab  
 c -----> abc  
 d -----> abcd  
 e -----> e ------> abcd  
 f -----> ef abcd  
 g -----> efg abcd  
 h -----> efgh abcd  
 i -----> i ------> abcdefgh

#### Reset

Buffer which is the core part of bufio.Writer can be re-used for different destination writer with Reset method. It saves memory allocation and extra work for garbage collector

#### Buffer free space

To check how much space left inside the buffer we can use Available method

We’ll be using the hashing algorithm BCrypt.

Goroutin

The other point to note, and this is important, is that goroutines allow you to easily scale your code across all the CPU cores that are available.

Go maintains a philosophy of “Don’t communicate by sharing memory, share memory by communicating.”

Channels are a way to send and receive data between goroutines, and provide a way to block your code until you receive information on it.

myChan := make(chan int)

A select statement with two channels is a great way to handle timeouts in your code.