Persistent :

1. What is difference between Module & package in golang?

Key Differences Between Packages and Modules

Feature	Package	Module
Definition	A collection of Go files in the same directory.	A collection of related Go packages with versioning.
Purpose	Organize code into reusable components.	Manage dependencies and versioning for Go projects.
File	No specific file defines a package.	Defined by a go.mod file.
Scope	Local to a directory.	Can include multiple packages and dependencies.
Visibility	Uses uppercase/lowercase for exported/unexported identifiers.	No visibility concept; manages dependencies.
Versioning	No versioning.	Supports semantic versioning.
Dependency Management	Not applicable.	Manages dependencies explicitly in go.mod.

2.

=>

3. Write program below :
   Input -> aabbbcccc O/p –> a:2, b:3, c:4
   Input ->Hello O/p -> H:1, e:1,l:2,o:1

```
package main

import (
	"fmt"
)

func countCharacters(input string) map[rune]int {
	// Create a map to store character counts
	charCount := make(map[rune]int)

	// Iterate over each character in the input string
	for _, char := range input {
		// Increment the count for the current character
		charCount[char]++
	}

	return charCount
}
```

```
func main() {
	// Test cases
	input1 := "aabbbcccc"
	input2 := "Hello"

	// Count characters for input1
	result1 := countCharacters(input1)
	fmt.Print("Input -> ", input1, " O/p -> ")
	first := true
	for char, count := range result1 {
		if !first {
			fmt.Print(", ")
		}
		fmt.Printf("%c:%d", char, count)
		first = false
	}
	fmt.Println()

	// Count characters for input2
	result2 := countCharacters(input2)
	fmt.Print("Input -> ", input2, " O/p -> ")
	first = true
	for char, count := range result2 {
		if !first {
			fmt.Print(", ")
		}
		fmt.Printf("%c:%d", char, count)
		first = false
	}
	fmt.Println()
}
```

4.

trace the output & please explain why

```
func foo(ch chan int, n int) {
	for i := 0; i < n; i++ {
		ch <- (i * 2)
	}
}

func main() {
	ch := make(chan int, 10)

	go foo(ch, cap(ch))
```

```
        for i := range ch {

                fmt.Print(" ", i)
        }
}
```

O/P => 0 2 4 6 8 10 12 14 16 18 fatal error, deadlock!

Why => does foo close the channel after sending all values? No, it doesn't. That's a problem. Because the range loop in main will keep waiting for more values, but the channel is never closed. So the main function might block forever, causing a deadlock..

After 10 (0 to 9)channel is empty and the sender is gone, so the receiver blocks forever.

```
func main(){
  ch1 := make(chan int)
  ch1 <- 5
  Go foo(ch)
  fmt.Println(<- ch1)

}
```

O/P => error , blocking call unbuffer channel

Solution:

```
ch1 := make(chan int,1) // make buffer channel
```

==========

```
a := [...]int{0, 1, 2, 3}
        x := a[:1]          //0
        y := a[2:]          //2,3
        x = append(x, y...)  //0,2,3
        x = append(x, y…)//0,2,3,2,3
        fmt.Println(a, x)//0,1,2,3,    0,2,3,2,3
```

Backing array,
How go lang work in micro service than other programming language.
How rest API worked ?
Write interface in Golang with example, take different function to call interface function

What is o/p of below code?

```
package main

import (
      "fmt"
)

func Test(c chan string) {
      fmt.Println(<-c) // Attempt to receive from the channel
```

```
}

func main() {
      ch := make(chan string)
      ch <- "This main" // Send to the channel
      go Test(ch)       // Launch goroutine
}
```

O/P => deadlock, as channel is send data before launch go routine

Solution => always send data to channel after launching go routine.

What is O/p Why ? How to fixed below code:

```
package main

 import ("fmt""time")

func receive(c chan int)

 {  fmt.Println("received data:", <-c) // Receive data from the channel and print it

}

func main() {

ch := make(chan int, 10) // Create a buffered channel with a capacity of 10

go receive(ch)           // Launch a goroutine to call the 'receive' function

for i := 0; i < 10; i++ {   ch <- i // Send data (0 to 9) to the channel

l}

close(ch)              // Close the channel (no more data will be sent)

time.Sleep(2 * time.Second) // Sleep for 2 seconds to ensure the goroutine can receive}
```

O/P= >received data 0

Why=> receive function only receives and processes **one value** from the channel, even though there are 10 values sent. it exits after processing the first value.

Solution=>for value := range c { // Loop until the channel is closed

```


func receive(c chan int) {

for value := range c { // Loop until the channel is closed
```

```
 fmt.Println("received data:", value)
}
```

What is difference between concurrency & parallel programming?

=> Parallel meaning multiple task at same time like eating & watching TV

But in case Concurrency meaning schedule task into timeslice to execute it some time interval. Means it switched from one thread to another for CPU execution but they not executing together at same time.

In Parallel programming task is running in multiple CPU(Processor,Cores) but concurrency not given to multiple CORE(CPU), it run within single core.

Does GO have exception?

=> NO, we used if statement

What is pointer?

=> it hold address of variable.

Go routine	Thread
It application level	It is OS level
Required less memory 2KB	Required less memory 2KB
It manage by Go run time	It manage OS.

What is difference between Buffered & unbuffered channel?

=> In Un-buffered : sender is block until received the received data from channel.
vice versa mean receiver is also block until sender send data.

Buffered	Unbuffered
Sender is block when Channel is full. Receiver is block when Channel is empty.	Sender block until receiver received data Receiver block until sender send.

What is default value of boolean?
=>false

what is Type assertion?
=> it extract value of underline interface.(empty interface)
ex. Foo(param interfacef{})
switch val :=param.(type){

```
default:
        fmt.Printf("\n type:%T", test)
case int8:


}
```

Ex2,
value, ok := param.(Type)

What is Empty interface value?
=> NIL

what happen other goroutine, when “main” function terminated ?
=>When “main” function terminated all other goroutine is terminated.

silce := 1,2,3,4,5,6 => remove at index 3
o/p => 1, 2, 3, 5, 6
=>

```
// Remove element at index 3 (value 3)
 index := 3
 silce = append(silce [:index], silce [index+1:]...)
```

What is differnce between grpc & REST API
=> rest API used http layered ,grpc used transport layer.

what difference between new & make?
=>Both allocate memory but

New	Make
It return pointer with uninitialized pointer	returns an initialized value of the given type, not a pointer.
It used allocate memory of basic data type like int, float	

What is difference between below statement?
a := make([]int, 10)
b := make([]int, 0, 10)
c := make([]int, 10, 10)
=>

convert int to string
=>

How microservice communicate ?
=>1. REST API
2. gRPC
3. Message queues

What is context ,why it used?
=>It used for run background, timeout (cancellation),Request-Scoped Values:

Request-Scoped Values: it useful scenario where web server required to pass request id, authenticate token or tracking information(request information between goroutine Or function call.

What is generic programming in golang ?

=>

Ideas

1. Todoapp (in cli)
2. Web api (stateless calculator)
3. Web scraper (dead link finder)
4. URL shortener (html in go)

5. Currency converter (TUI + api)

1. Basic Level Questions

What is Go, and why is it popular?

Go (Golang) is an open-source, statically typed, compiled programming language designed by Google in 2009. It's known for simplicity, performance, and concurrency support. Its popularity stems from:

- **Simplicity**: Minimal syntax, no exceptions or inheritance.
- **Concurrency**: Built-in goroutines and channels make parallelism easy.
- **Performance**: Compiled to machine code, rivaling C/C++.
- **Ecosystem**: Strong standard library and tools (e.g., go fmt, go test).

  It's widely used for cloud systems (Docker, Kubernetes), microservices, and CLI tools.

How does Go manage memory without a garbage collector?

Trick question! Go *does* have a garbage collector (GC). It uses a mark-and-sweep GC to automatically manage memory, freeing developers from manual allocation/deallocation (unlike C). You can influence memory with pointers, but the GC handles cleanup.

Explain the difference between var, :=, and const in Go.

- var: Explicit variable declaration with type (e.g., var x int = 5). Can be used at package or function scope. Allows zero values without initialization (e.g., var x int is 0).
- :=: Short-hand declaration and assignment (e.g., x := 5). Infers type, only usable inside functions.
- const: Defines constants (e.g., const pi = 3.14). Immutable, evaluated at compile time, no reassignment.

What are slices, and how do they differ from arrays?

- **Arrays**: Fixed-size, contiguous memory blocks (e.g., [3]int{1, 2, 3}). Rarely used directly.
- **Slices**: Dynamic views over arrays (e.g., []int{1, 2, 3}). Backed by an array with length and capacity. More flexible—can grow/shrink via append().

  Key difference: Arrays are static; slices are dynamic and reference-based.

How do you create and use maps in Go?

Maps are key-value stores (like hash tables). Creation:

go

CollapseWrapCopy

```
m := make(map[string]int) // Empty map
```

```
m := map[string]int{"a": 1, "b": 2} // Initialized map
```

Usage:

go

CollapseWrapCopy

```
m["c"] = 3          // Add/update
```

```
val, ok := m["a"]  // Retrieve (ok is true if key exists)
delete(m, "b")     // Remove
```

What is the purpose of the init() function?

init() runs automatically before main() or when a package is imported. Used for setup (e.g., initializing variables, configs). Multiple init()s per package are allowed and run in declaration order.

How do defer, panic, and recover work in Go?

- defer: Schedules a function to run after the surrounding function completes (e.g., closing files). Executes in LIFO order.
- panic: Stops normal execution, triggers stack unwinding (like an exception).
- recover: Catches a panic, resumes execution. Must be called in a deferred function.

Example:

go

CollapseWrapCopy

```
func main() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered:", r)
        }       }()      panic("Oops!")
}
```

Explain the difference between pointers in Go vs other languages.

Go pointers (e.g., *int) are similar to C: they hold memory addresses. Key differences:

- No pointer arithmetic (unlike C/C++).
- Pass-by-value by default; pointers enable pass-by-reference.
- GC manages memory, so no manual free() (unlike C).

Syntax: p := &x (address), *p (dereference).

How does Go handle concurrency with goroutines?

Goroutines are lightweight threads managed by the Go runtime (not OS threads). Start with go func(). Cheap (2KB stack) and multiplexed onto OS threads. Channels coordinate communication. Example:

go

CollapseWrapCopy

```
go func() { fmt.Println("Running!") }()
```

What is the difference between buffered and unbuffered channels?

- **Unbuffered**: Blocks sender until receiver is ready (synchronous, make(chan int)).
- **Buffered**: Allows sending up to capacity without blocking (asynchronous, make(chan int, 5)). Blocks only when full.

Example:

go

CollapseWrapCopy

```
ch := make(chan int, 1) // Buffered
ch <- 1                 // Non-blocking until full
```

2. Medium Level Questions

What is the purpose of the context package in Go?

The context package manages request-scoped data, deadlines, and cancellation signals. Common uses: timeouts, cancellation in goroutines, passing metadata. Example:

go

CollapseWrapCopy

```
ctx, cancel := context.WithTimeout(context.Background(), time.Second)
defer cancel()
```

How does Go implement interfaces differently from other languages?

Go uses *implicit* interfaces: a type satisfies an interface if it implements its methods—no explicit declaration (unlike Java's implements). Duck typing: "If it walks like a duck..." Example:

go

CollapseWrapCopy

```
type Writer interface {
    Write([]byte) (int, error)
}
```

What are struct embedding and composition in Go?

- **Embedding**: Include a struct in another to inherit its fields/methods (e.g., type B struct { A }). Promotes composition over inheritance.
- **Composition**: Build structs from smaller structs explicitly. No classical inheritance in Go.

  Example:

go

CollapseWrapCopy

```
type A struct { x int }
type B struct { A }
b := B{A{5}} // b.x is 5
```

Explain how mutexes and wait groups work in Go.

- sync.Mutex: Locks shared resources for thread safety (Lock(), Unlock()).
- sync.WaitGroup: Waits for goroutines to finish (Add(), Done(), Wait()).

  Example:

go

CollapseWrapCopy

```
var mu sync.Mutex
var wg sync.WaitGroup
i := 0
wg.Add(2)
```

```
go func() { mu.Lock(); i++; mu.Unlock(); wg.Done() }()
wg.Wait()
```

How do you handle timeouts in Go using context?

Use context.WithTimeout:

go

CollapseWrapCopy

```
ctx, cancel := context.WithTimeout(context.Background(), 2*time.Second)
defer cancel()
select {
case <-time.After(1 * time.Second): // Work
case <-ctx.Done(): // Timeout
    fmt.Println("Timed out")
}
```

What are race conditions, and how do you detect them in Go?

Race conditions occur when multiple goroutines access shared data concurrently, causing unpredictable results. Detect with go run -race (race detector) or tools like sync.Mutex.

How do you implement worker pools using goroutines?

Spawn a fixed number of goroutines reading from a job channel:

go

CollapseWrapCopy

```
jobs := make(chan int, 100)
for w := 1; w <= 3; w++ {
    go func() {
        for j := range jobs {
            fmt.Println("Worker", w, "processed", j)
          }       }() }
```

Explain how dependency injection is done in Go.

Pass dependencies (e.g., interfaces) via constructor or parameters, avoiding global state. Example:

go

CollapseWrapCopy

```
type Service struct {
     db DB } func NewService(db DB) *Service {
    return &Service{db}
}
```

How do you write and optimize table-driven tests in Go?

Use a slice of test cases:

go

CollapseWrapCopy

```
tests := []struct {
    input  int
    want   int
}{     {1, 2},
    {2, 4},
} for _, t := range tests {
     got := someFunc(t.input)     if got != t.want {
        t.Errorf("got %d, want %d", got, t.want)
     } }
```

Optimize by parallelizing: t.Parallel().

What is the purpose of the sync.Once and sync.Map packages?

- sync.Once: Ensures a function runs exactly once (e.g., initialization).
- sync.Map: Thread-safe map for concurrent access (unlike regular maps).
  Example:

go
CollapseWrapCopy

```
var once sync.Once
once.Do(func() { fmt.Println("Init") })
```

3. Advanced Level Questions

How does Go achieve high performance compared to other languages?

- Compiled to machine code (no VM like Java).
- Efficient GC with low latency.
- Lightweight goroutines (vs OS threads).
- Simple type system, minimal runtime overhead.

Explain memory management in Go and how the garbage collector works.

Go uses a concurrent, tri-color mark-and-sweep GC:

- **Mark**: Identifies live objects (runs concurrently with program).
- **Sweep**: Frees unmarked objects.
  Tunable via GOGC (default 100, controls GC frequency). Stack allocation for small objects reduces heap pressure.

What are Go interfaces, and how do they enable polymorphism?

Interfaces define behavior (methods). Types implement them implicitly, enabling runtime polymorphism without inheritance. Example:

go
CollapseWrapCopy

```
type Animal interface { Speak() string }
var a Animal = Dog{} // Dog implements Speak
```

How does Go handle parallelism vs concurrency?

- **Concurrency**: Managing multiple tasks (goroutines, channels).

- **Parallelism**: Executing tasks simultaneously (multi-core via GOMAXPROCS).
  Go prioritizes concurrency; parallelism depends on hardware and runtime.

How would you design a distributed system using Go?

Use goroutines for concurrency, gRPC/HTTP for communication, and libraries like Raft for consensus. Example:

- Microservices with REST/gRPC.
- Channels for internal coordination.
- Redis/Kafka for messaging.

What is reflection in Go, and when should you use it?

Reflection (reflect package) inspects types/values at runtime. Use sparingly for serialization, ORMs, or dynamic behavior. Example:

go

CollapseWrapCopy

```
v := reflect.ValueOf(5)
fmt.Println(v.Type()) // int
```

Avoid when performance is critical—slow and complex.

How do you optimize Go applications for high throughput?

- Minimize allocations (reuse buffers).
- Use goroutines judiciously (avoid over-spawning).
- Leverage sync.Pool for object reuse.
- Profile with pprof to find bottlenecks.

Explain how Go's HTTP package handles concurrent requests.

The net/http package spawns a goroutine per request via http.Server. Scales well due to lightweight goroutines and event-driven I/O.

How does Go handle real-time event-driven architectures?

Use channels for event passing, goroutines for handlers, and libraries like gorilla/websocket for real-time comms. Example: pub/sub with channels.

What are the best practices for writing scalable Go applications?

- Keep code simple and modular.
- Use interfaces for decoupling.
- Handle errors explicitly.
- Optimize concurrency with channels/mutexes.
- Profile and benchmark regularly.

Let me know if you'd like deeper dives or code examples for any of these!

```
// You can edit this code!
// Click here and start typing.
package main
```

```
import "fmt"

type student struct {
	rollNumber int
}

func foo(c chan string) {
	for val := range c {
		fmt.Printf("\n%v", val)
	}
}


func main() {



	ch := make(chan string, 10)

	//var count int

	//count = 1

	for i := 0; i < 3; i++ {

		m := map[student]string{

			student{1}: "a",

			student{3}: "c",

			student{2}: "b",

		}




		for _, v := range m {

			//fmt.Printf("%v - %v\n", k.rollNumber, v)

			//rollno := string(k.rollnumber) //string.itoa(k.rollnumber)

			//var rollno string

			//_ := sprintf(rollno, "%s", k.rollnumer)


			go foo(ch)
```

```
                ch <- v
            }
            //defer close(ch)
            fmt.Printf("\n")
        } //for outer
        defer close(ch)
        time.Sleep(2*time.second)
}
```