1. **What is a blank identifier?**

* blank identifier is represented by the underscore (\_) character
* avoiding unused import errors
* Used to ignore values in various context
  + ignore loop variables
  + ignore function variables

1. **What's the difference between declare, assign, and initialize?**

Declare: var a int

Assignment: a = 10

Initialization: var a int = 10

1. **Explain allocation and initialization**

Allocation:

* refers to reserving a block of memory for storing data
* ensures that memory is reserved for variables or data structures

Initialization:

* refers to assigning an initial value to a variable at time of declaration
* ensures that a variable starts with a known, defined value rather than random data, avoiding undefined behaviour

1. **What is the difference between make([]int, 10), make([]int, 0, 10), make([]int, 10, 10)**

make([]int,10)

* Length 10
* Capacity 10
* All the elements are initialized to 0 value for int

make([]int,0,10)

* Length 0
* Capacity 10
* No element is present initially but the underlying array has space for 10 integers

make([]int,10,10)

* Same as make([]int,10)

1. **What types can a map use as a key in the Go programming language?**

* Types that support **==** and **!=** operations
* Basic types(int, bool, string, uint, etc)
* Pointers
* Channel
* Interface [ underlying value in interface must be comparable]
* Struct [if all the fields are comparable]

CANNOT USE

* Slice
* Struct [any field inside is not comparable]
* Map
* Function

1. **What does it mean to write idiomatic Go code?**

means adhering to the conventions, practices, and style guidelines that the Go community has established

1. **Which do you choose: performance or readability?**

Readability first and then go for performance where it can be achieved.

And also depends on the projects need

1. **Why was Go created, and who created it?**

* team at Google primarily consisting of Robert Griesemer, Rob Pike, and Ken Thompson
* Reason for creating
  + to bridge the gap between high-performance, low-level system programming and high-productivity, high-level application programming
  + Concurrency [simplified concurrent programming in go]
  + Performance [c like performance with high level language]
  + Fast Compilation
  + Simplicity
  + Scalability
  + Garbage Collection [auto memory management and reduces bugs]

1. **Can you explain what is meant by 'Go is strongly typed?**

Type Safety

* Compile time checks
* No implicit conversions

Type Inference

* Supports type inference through `: =`
* Once type is set it can’t be changed

Consistent and Predictable Behaviour

1. **What is the var keyword used for and when do you use it?**

* Used to declare variables
* Explicit type declaration
* Zero value initialization
* Global or package level variables
* Multiple variable declaration var (a int b string)

1. **When dealing with computer architecture, what does the 'word size' mean?**

* Standard unit of data that the CPU can handle and process in one operation
* Common size
  + 8-bit, 16-bit, ….32-bit, 64-bit
* Larger word size allows more data to be processed at once

1. **What is a compiler?**

* Software program that translates high level code (human readable) into machine code (binary executable)
* Go complier `gc` or `gccgo`
* **Compilation Process**: Involves lexing, parsing, semantic analysis, optimization, code generation, and linking
* Go Toolchain Commands: go build, go install, and go run for compiling and managing Go code.

1. **How does a computer work?**

* **Power On:** PSU powers up the system.
* **POST:** Hardware diagnostics and checks.
* **BIOS/UEFI Initialization:** Hardware setup and bootloader location.
* **Bootloader Execution:** Loading the OS kernel.
* **Loading the OS:** Kernel and essential services initialization.
* **System Initialization:** Starting system services and hardware drivers.
* **User Login:** User authentication and desktop access.

1. **What is a garbage collector?**

* the garbage collector (GC) is a built-in feature of the Go runtime that automatically handles the allocation and deallocation of memory
* primary role is to identify and reclaim memory that is no longer in use
* which prevents memory leaks and making memory management simpler
* Features
  + Automatic Memory Management
  + Concurrency friendly
  + Generational Collection
  + Background Collection
  + Tunable
* Works
  + Mark and sweep p

1. **Explain how package management works in Go?**

* It resolves around modules
* When you run go build, go test, or go mod tidy, Go will automatically add this dependency to your go.mod file and download it.
* Also talk about
  + go mod tidy
  + go.sum & go.mod
  + go get
  + import

1. **What is the difference between make and new?**

* Built in function used for memory allocation

|  |  |  |
| --- | --- | --- |
|  | **Make** | **new** |
| Return type |  | pointer |
|  |  | Zeroed allocation |
|  | Slice, map and channel | Any type |
|  | Initializes the underlying variables also |  |

1. **Tell us about bytes, code points, and characters in relation to strings and UTF-8?**

* **Byte:**
  + unit of digital information that consists of 8 bits
  + bytes are minimal unit of storage in context of string
  + UTF-8 is a variable-width character encoding
    - uses one to four bytes to represent each Unicode character
* **Code Point**
  + code point is a value that maps to a particular character in the Unicode standard
  + Higher level concept in the hierarchy text encoding
* **Characters**
  + text element that represents a letter, digit, punctuation, symbol, or other sign
  + can be of multiple code point
  + can be of multiple bytes
* UTF-8 strings are composed of bytes. The number of bytes used to represent a character varies.
* Code points are logical units that represent characters in Unicode
  + The UTF-8 encoding determines how these code points are mapped to bytes
* Characters are what we see and work with in text

1. **What is embedding a struct and inner-type promotion?**

**Embedding:**

* means including one struct type within another struct type without explicitly defining a field name
* promotes code reuse and composition over inheritance
* Advantage:
  + **Code Reuse**
  + **Simplicity:** simplifies access to the embedded struct's fields and methods

**Inner-Type Promotion:**

* the visibility and accessibility of the embedded struct's fields and methods in the embedding struct

**Key Points**

* If a method or field is defined in both the outer and inner struct, the outer struct's definition takes precedence.
* All fields and methods of the embedded struct are promoted to the embedding struct's scope, provided there are no name conflicts.
* In case of a conflict (e.g., both structs having a field or method with the same name), you need to access the inner struct's field or method explicitly

1. **What is the comma ok idiom?**

* pattern in Go used to handle certain operations that can fail or produce multiple results, with one result indicating success or failure
* used in scenarios involving map lookups, type assertions, and channel operations.
* allows for a clean and expressive way to handle potential errors or the absence of expected results
* age, ok := MyInterface.(int)

1. **Is Go an Object Oriented language?**

* Not completely
* Go blends procedural and functional programming paradigms with object-oriented techniques
* Supports
  + Encapsulation:
    - exported and un-exported identifiers(variable, function, type, etc)
    - uppercase – exported(public) -accessed from other packages
    - lowercase-unexported(private)- only within package
  + Polymorphism
    - Through interface
  + Composition over Inheritance
    - Through embedding
* **No Constructors or Destructors:** Go does not have constructors or destructors. Instead, it uses factory functions(NewCar(){..}) to initialize structs.

1. **Somebody says that 'a string is a two word data structure' - what does this mean?**

* Refers to the internal implementation of the string
* Two word representation [internal representation]
  + Pointer to Data : points to the byte slice that holds the sequence
  + Length
* Why two word
  + Efficiency
    - Storing the pointer and the length separately allows for constant-time access to the length of the string and efficient memory allocation.
  + Safety
    - Strings in Go are immutable
  + **Interoperability**
    - two-word representation is straightforward enough to easily interface with underlying C libraries or system calls

1. **Somebody says that 'a slice is a three word data structure' - what does this mean?**

* Referring to the internal implementation of the slice
* Three word representation[underlying type]
  + Pointer to array
  + Length
  + Capacity
* Why
  + pointer provides direct access to the array's data,
    - enabling efficient read and write operations.
    - Share underlying data with other slices
    - Avoiding unnecessary data copying
  + Length
    - indicating how many elements the slice includes
    - number of elements that the slice can safely access
    - crucial for operations like iteration, indexing, and slicing, ensuring that bounds are not exceeded
  + Capacity
    - maximum number of elements the slice can hold without reallocating the underlying array
    - manage memory more efficiently
    - helps in scenarios where the slice needs to grow

1. **Tell me the difference between a nil slice and an empty slice?**

nil slice:

* The length of the slice is 0
* The capacity of the slice is 0
* The underlying array of the slice is nil

Empty slice:

* The length of the slice is 0
* The capacity of the slice is 0
* The underlying array of the slice is [] (empty array)

1. **What are typed and untyped constants, which do you prefer and why, and how do they relate to numeric literals?**

* Constants are immutable values that are known at compile time
* Type
  + Typed
    - Constants which are given types explicitly
    - Provide type safety [ensures that the value conforms to specific data type, explicit type checking at compile time]
    - Disadvantage: less flexibility
  + Untyped
    - Constants which are not explicitly bound to a type
    - Can be used more flexibly [without explicit type conversion]
    - Disadvantage: potential for type related issue
* Numeric Literals
  + Can be inherently treated as untyped constants.
  + Can be used with different numeric types without explicit conversion

1. **How does Go handle error handling compared to other languages?**

* Go treats errors as value
* Errors are handled through explicit checks rather than exceptions
* Go’s **errors** are built-in interface with a single **Error()** method
* Difference
  + Performance
    - Go: error handling is lightweight, with no performance overhead
    - Others: Often incur performance costs due to stack unwinding and exception objects.
  + Safety
    - **Go:** Explicit error returns ensure that errors are considered at every function call.
* Disadvantage
  + Readability
    - Error handling code is mixed with the main logic,
    - which some argue can lead to “error handling clutter

1. **Conversion or casting, and why?**

* They both refer to transforming one data type to another
* Go’s strict type system requires explicit conversions
* Why choose conversion
  + Prevents unintentional type changes
  + Compile time check ensures that conversions are only performed between the compatible type
  + Reduces errors

1. **What is nil?**

* Nil represents 0 value for pointers, interfaces, maps, slices, channels, function types
* It indicates absence of value or non-existence of the object
* Go allows comparison of these values to nil
* Nil in interface
  + An interface holding a nil pointer is not itself nil.

type MyInterface interface {

    Method()

}

type MyStruct struct {

    val string

}

func (ms \*MyStruct) Method() {

    fmt.Println("Hell")

}

func main() {

    var ms \*MyStruct // this is nil

    fmt.Println(ms)

    var iface MyInterface = ms // iface is not nil, but holds a nil pointer

    fmt.Println(iface == nil)  // false

    if iface != nil {

        // Trying to invoke a method on a nil pointer through an interface

        iface.Method() // This will not panic

        fmt.Println("Reached here")

    }

}

1. **How do we do benchmarking in Go?**

* Create a benchmarking function

func BenchmarkMyFunction(b \*testing.B) {

for i := 0; i < b.N; i++ {

MyFunction()

}

}

* Run benchmarks(. Can be replaced with specific benchmarking function)  
  go test -bench=.

1. **What is the difference between an int and a uint, and how does this relate to mechanical sympathy?**

* int and uint are both integer types
* int
  + signed integer
  + can store both +ve and -ve
  + range [depends on architecture]
    - **32-bit:** -2,147,483,648 to 2,147,483,647
    - **64-bit:** -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
* uint
  + unsigned integer type
  + only store +ve and 0
  + range [depends on architecture]
    - **32-bit:** 0 to 4,294,967,295
    - **64-bit:** 0 to 18,446,744,073,709,551,615
* Mechanical sympathy
  + refers to understanding how computer hardware and software interact in order to write more efficient code
  + using appropriate size can directly influence
    - cache usage
    - memory throughput
* Instruction Set Utilization
  + Certain CPU instructions are optimized for specific types of data
  + uint benefit from simpler and faster arithmetic
  + signed int(int) are more appropriate for general use cases
* Example
  + Using an **int** for array index makes no sense because it will never be negative
    - Using uint can reduce logical errors and might optimize performance

1. **Put these in the correct order: centisecond, second, nanosecond, millisecond, decisecond, microsecond**

* Nanosecond, Microsecond, Millisecond, Centisecond, Decisecond, Second

1. **What are pointers? Show us pointers at work in code.**

* Pointers are variable that store memory address of another variable
* Use **&** to get the address of a variable
* Use **\*** to dereference the address to the value that is stored at that address
* Go doesn’t allow pointer arithmetic

1. **Explain value semantics and pointer semantics. What are rules-of-thumb for using one versus the other?**

Value Semantics:

* Working with copies of data
* Each time we pass a value to a function a copy or assign it to another variable, a copy is made
* Behaviour:
  + Any change to the copy does not affect the original data

Pointer Semantics:

* You work with references(pointer) to data rather than the copies of data itself
* Behaviour:
  + Any changes made via pointer will reflect in the original data

Rule of Thumb:

* Use value semantics when
  + working with small simple data (int, float)
  + you want to avoid data modification
  + The cost of copying the data is low
* Use pointer semantics when
  + You need to modify the original data
  + Data is large and copying the data is inefficient
  + You are working with reference type (slice, map, channel)

1. **What is the stack, and what is the heap?**

* Both refer to a different region of memory with distinct characteristics
* Stack
  + Region of memory that operates on LIFO approach
  + Used to store
    - Function calls frames (or activation records)
      * Function call pushes a frame
      * Function return pops a frame
  + Typically stores
    - Local variables
    - Function parameters
    - Return addresses
    - Control information
  + Lifetime
    - Variable on a stack is limited to the duration of function call
    - Once the function exits, the memory for its local variables is automatically reclaimed.
  + Memory Management
    - Automatically by the compiler/runtime
    - Fast allocation and deallocation
  + Size limit
    - Limited and predefined
    - Stack overflow can occur if too much memory is used [deep recursion]
* Heap
  + Larger, more flexible region of memory used for dynamic memory allocation
  + Used for that data that persists beyond the scope of a function call
  + Content
    - Used for data that persists beyond the scope of the function
  + Lifetime:
    - Variable on heap stay allocated until they are explicitly deallocated or garbage collected(in go)
  + Memory management
    - By programmer or garbage collector
    - Garbage collector in go
  + Size limit
    - Larger and more flexible than the stack
    - Potentially fragmented
    - Slower due to the overhead of managing dynamic memory
* Thread Safety
  + Stack
    - Each thread has its own stack, so local variable management is inherently safe
  + Heap
    - Shared among threads, so care must be taken to manage concurrent access
* In Go
  + Stack: The Go runtime dynamically manages the stack size, allocating more space as needed.
  + Heap: Managed by Go's garbage collector, which automatically reclaims memory that is no longer in use.

1. **What is escape analysis?**

* Technique used by compilers to determine whether a variable should be allocated on the stack or the heap
* This analysis is crucial for optimizing memory allocation and enhancing performance
* Scenarios of escape
  + Function return:
    - If you return a pointer to variable the variable will be allocated to the heap
  + Assignment to global or long-lived variables
    - If a local variable’s address is stored in a global variable with a longer lifetime
  + Pass to another function
    - If local variable is passed by reference (via pointer) to another function
* Benefits
  + Performance:
    - stack allocation is faster compared to heap.
    - It avoids overhead association in heap management
  + Memory Management
    - Reduces the load on garbage collector by minimizing heap allocation
  + Safety
    - Help identify the variables that should have controlled lifetimes, reducing the risk of dangling pointers
* Command
  + **go build -gcflags="-m" main.go**

1. **What is an interface?**

* interfaces define behaviours that other types (structs or other types) must implement
* This allows a form of polymorphism
* Key features
  + Method set
    - Defines a set of method signatures but not the actual implementation
  + Implicit implementation
    - Types in GO implicitly satisfy an interface as long as they implement all the methods specified in the interface
  + Polymorphism
    - Can use any variable that implements an interface
* Empty interface can be used to store anything

1. **What are method sets, and how do you use them?**

* Method sets defines the collection of methods that a type possesses
* They define if a type implements an interface
* Types
  + Value Receiver
    - Methods implemented on a value receiver can only be accessed through the value type of the receiver
  + Pointer Receiver
    - Methods implemented using a pointer receiver can be accessed through both the value and pointer types
* Use
  + Used to implement an interface
* Why
  + Promotes Clean design
  + Reducing Coupling

1. **What is a type set?**

* Type set is a collection of types that a type parameter in a generic type or function can represent
* Why
  + Reusability
    - Generic code can be reused across various types without rewriting for each specific type.
  + Type Safety
    - Ensures that only compatible types are used with generic code, reducing runtime errors and bugs
  + Flexibility
    - Allows functions and types to be defined with broad applicability while still retaining type constraints
* Ex

type Number interface {

int | int32 | int64 | float32 | float64

}

* Usage:
  + Simple Generics
    - Use `any` to define function and types that accept any type
  + Constraints
    - Use interfaces or type unions to define more specific type sets

1. **What is concrete data?**

* Exact type that directly represents a set of values
* Ex: int, float64, bool, string
* Unlike interfaces, concrete types provide an actual implementation and storage of data.
* Benefits
  + Direct value representation
  + Performance efficiency
  + Type Safety

1. **Explain generics.**

* Generics in programming refer to the ability to write functions, data structures, and types that can operate on many different data types while providing type safety
* Limitation
  + Complexity
  + Performance

1. **What is the difference between concurrency and parallelism?**

* Describe concepts related to multitasking and performance
* Concurrency
  + Dividing tasks into smaller tasks and managing the execution such that they appear to run simultaneously
  + Managing multiple tasks at once but not necessarily simultaneously
  + Involves structuring a program in such a way that it can handle multiple operations that overlap in time
  + Can be achieved on single processor through techniques called context switching
  + Key Points
    - Interleaved Execution: Task are split into smaller sub-tasks and the processor switches between them
    - Task management
* Parallelism
  + Executing Multiple tasks simultaneously
  + Achieved through multiple processors or cores that can run different tasks or sub-tasks at the same time
  + Key Points
    - Simultaneous Execution
    - Performance: Is about increasing performance by utilizing multiple computing resources

1. **What is a Goroutine?**

* Is a lightweight thread managed by go runtime
* Goroutines are used to perform concurrent operations
* Key Features
  + Lightweight:
    - Much lighter compared to operating system threads
    - Smaller stack size (a few KB) and grow as needed
  + Managed by goroutine:
    - Go runtime schedules goroutine onto available processor threads
  + Easy to use:
    - Just type `go` before a function call
* Go provides channels for communication and synchronization
* Key Points to remember
  + Scheduling
    - Goroutine are scheduled by go runtime
    - Uses the structure known as GPM model (Goroutine, Processes, Machine threads)
  + Stack Management
    - Stack for goroutine increases and shrinks as needed unlike traditional thread which have fixed stack size
  + Blocking
    - If a goroutine performs a blocking operation only that goroutine is blocked, while the others continue to execute

1. **Describe a time when you used goroutines and channels?**
2. **Should you use buffered channels? Why or why not?**
3. **What is the difference between switch and select?**

* Both are control flow statement
* Switch
  + It evaluates an expression and executes the case that matches the result
* Select
  + Used for handling multiple channel operations
  + It blocks until one of the cases can run, then it executes the case
  + If multiple channels are ready, one of them is chosen at random
  + I works only with channels

1. **Explain internal, external, and data latencies in Go.**

* Time delays encountered in different stages of the program execution
* Internal Latency
  + Internal latency refers to the delays that occur within the program's internal mechanisms
    - Data latency, sometimes referred to as data access latency, is the delay experienced when accessing or transferring data goroutines
* External Latency
  + External latency refers to delays that occur due to external factors and dependencies outside the control of the Go runtime
    - Network Latency: Delays caused by network communication, ex: when making HTTP requests or communication with remote services
    - I/O Operations: Disk I/O, database queries, and other forms of data storage and retrieval that involve waiting for external system to respond
    - External Services: Latency introduced by dependencies on third-party services or APIs
* Data Latency
  + Data latency, sometimes referred to as data access latency, is the delay experienced when accessing or transferring data
    - Memory Access Patterns: In efficient memory access patterns, such as cache misses, can lead to delays
    - Data Structure Efficiency: The choice of performance characteristics of data structures (array, linked list)
    - Inter-Process Communication: Delay in transferring of data between processes or through channels