GO

1. Arrays:

* Fixed-size sequence of elements of the same type.
* Syntax: ***var arrayName [size]elementType***
* *arrayName: = [size]elementType {element1, element2, ..., elementN}*
* *numbers := [3]int{1, 2, 3}*

\*\* Go doesn't support pointer arithmetic directly like in c++ like prt++.

1. Slices:

* A slice is a dynamic-size sequence built on top of arrays.
* Unlike arrays, slices are flexible and can be resized.
* *shortSlice := []int{7, 8, 9}*

1. Maps:

* used to represent collections of key-value pairs.
* They are unordered, and each key in the map must be unique.
* *studentGrades := map[string]int{*

*"Alice": 85,*

*"Bob": 92,*

*"Charlie": 78,*

*}*

\*\* Makes maps of the key value of any type.

*map[string]interface{}{*

*"ok": true,*

*"balance": prevBalance + req.Amount,*

*}*

1. Functions

func functionName(parameters) returnType {

*// Function body*

*return returnValue*

*}*

1. Structs

* A struct is a composite data type that groups together variables, known as fields, under a single name.
* Syntax: *type StructName struct {*

*/\* Fields \*/*

*}*

\*\* The syntax ***func add(a, b int)***is a shorthand way of declaring multiple parameters of the same type

1. Methods

* *goCopy code func (receiverType) methodName() {*

*// Method implementation*

*}*

* A method must have a receiver.
* The receiver is a special parameter that specifies the type on which the method operates.
* Method Receivers:

1. Value Receiver

* *func (t Type) method Name() {*

*// Value receiver method implementation*

*}*

* Takes a copy of the instance; suitable when modifications aren't required.

1. Pointer Receiver

* *func (s \*Student) UpdateGrade(newGrade string) {*

*s.Grade = newGrade*

*}*

* Takes a reference to the instance; used for modifying the original instance or avoiding unnecessary copying.

1. Interfaces:

* An interfaces provide a way to define a set of method signatures that a type can implement.
* This allows you to define behaviours without specifying how they are implemented.
* *type Shape interface {*

*Area() int*

*}*

1. Error Handling:

* error type for handling errors.
* *if err! = nil* commonly used for error checking.

1. Defer

* Used to ensure that a function call is performed later in a program's execution, usually for purposes like cleanup.
* Deferred calls are executed in Last In, First Out (LIFO) order.
* *defer fmt.Println("This will be executed last.")*

1. Panic

* Panic stops the normal execution of a function and begins panicking.
* *panic("Something went wrong!"*

1. Recover:

* Recover is used to regain control of a panicking goroutine.
* ***recover()****:* This function is called to check if a panic occurred.
* *if r := recover(); r != nil {*

*fmt.Println("Recovered from panic:", r)*

*}*

1. **File Handling:**

* import “os”
* create file 🡪 *newFile, err = os.Create("test.txt")*
* get file info 🡪 *fileInfo, err = os.Stat("test.txt")*
* rename and move 🡪 *err := os.Rename(test.txt, test2.txt)*
* delete 🡪 *err := os.Remove("test.txt")*
* open 🡪 *file, err := os.Open("test.txt")*
* compress file *🡪 gzipReader, err := gzip.NewReader(gzipFile)*

\*\*When reading data from a file or extracting information from the web, it is retrieved in byte format rather than in string format.

1. **Web Data**

* *url :=* [*https://www.example.com*](https://www.example.com)

*response, err := http.Get(url) //,making http request*

1. **Goroutines:**

* Lightweight threads, concurrent execution
* **Syntax :** *go functionName()*

1. **Channels:**

* Communication between goroutines.
* Think of them as pipes through which you can connect with different concurrent goroutines. The communication is bidirectional by default, meaning that you can send and receive values from the same channel.
* Channels are a typed conduit through which you can send and receive values with the channel operator , <-.

\*\*In Go, you define methods separately from the struct declaration

*type Rectangle struct {*

*Length float64*

*Width float64*

*// You can embed a type to achieve a similar effect*

*ShapeMethods*

*}*

*type ShapeMethods struct {*

*}*

Fields() 🡪 The Fields() function in Go is used to split a string into substrings based on whitespace and return a slice of the substrings. It removes any leading or trailing whitespace and treats consecutive whitespace characters as a single separator.

json.Marshal 🡪 function in Go is used to encode data into a JSON representation. It takes an input data structure (typically a struct, map, slice, or primitive type) and returns a byte slice containing the JSON encoding of the input.

\*\*For Marshal if you use lower case in case of json it won’t show up.

Q. What happens when we run hello world in go?

Ans.

1. Allocate the memory for the program
2. Start a main goroutine for our program
3. Run the program on a thread
4. Execute the code
5. Shut down and clean up one the program completes
6. Clean up in programming means releasing acquired resources and memory after a program completes its task. This ensures a clean system state, preventing issues like memory leaks. It's akin to tidying up after a successful performance, maintaining order and efficiency.

CONCURRENCY

Q1. What is Concurrency?

Ans.

Concurrency entails the appearance of multiple tasks operating simultaneously, although, in reality, they are interchanging their active states or activities.

Q2. How does Concurrency differ from parallelism?

Ans.

Concurrency involves tasks appearing to run simultaneously by interleaving their execution, while parallelism involves tasks actually executing simultaneously using multiple processors.

Q3. What is a Goroutine in Go?

Ans.

A Goroutine is a lightweight thread managed by the Go runtime. It enables concurrent execution of functions, allowing for efficient concurrency in Go programs.

\*\* Go keyword starts a new go routine

Q4. How are Goroutines created and managed?

Ans.

Goroutines are created using the **go** keyword followed by a function call. The Go scheduler manages their execution, and they run concurrently, utilizing available CPU cores.

Q5. What is the purpose of Channels in Go concurrency?

Ans.

Channels facilitate communication and synchronization between Goroutines in Go. They provide a safe way for Goroutines to exchange data and coordinate their execution.

Q6. How does the **select** statement work in Go concurrency?

Ans.

The **select** statement allows Goroutines to wait on multiple communication operations. It enables non-blocking communication and helps manage multiple channels effectively.

Q7. Can you explain the concept of Mutex in Go?

Ans.

A Mutex (Mutual Exclusion) in Go is used to synchronize access to shared resources to avoid data races. It ensures that only one Goroutine can access the shared resource at a time.

*var mu sync.Mutex*

*mu.Lock() // Locks the mutex*

*// Critical section (shared data access)*

*mu.Unlock() // Unlocks the mutex*

\*\* To prevent race condition of the goroutines we use sync mutex

Q8. What is the purpose of the **sync** package in Go?

Ans.

The **sync** package in Go provides synchronization primitives, including Mutex, WaitGroup, and others, to support coordination between Goroutines and manage shared resources safely.

Q9. How does the **WaitGroup** work in Go concurrency?

Ans.

**WaitGroup** is used to wait for a collection of Goroutines to finish executing. It ensures that the main Goroutine waits for all other Goroutines to complete before proceeding.

*var wg sync.WaitGroup*

*wg.Add(1) // Increment the WaitGroup counter*

*go func() {* \*\*anonymous function

*defer wg.Done() // Decrement the WaitGroup counter when the Goroutine completes*

*// Goroutine logic*

*}()*

*wg.Wait() // Wait for all Goroutines to finish*

Q10. Explain the concept of Race Conditions in concurrent programming.

Ans.

Race conditions occur when multiple Goroutines access shared data concurrently, leading to unpredictable outcomes. Proper synchronization mechanisms, such as Mutexes, are necessary to avoid race conditions in Go programs.

\*\*Executing go run -race file\_name.go enables the race detector, which can identify and highlight race conditions in the code.

Q11. What is the purpose of the **sync.Map** in Go?

Ans.

The **sync.Map** in Go is a concurrency-safe map provided by the **sync** package. It allows for concurrent read and write operations on a map without requiring external synchronization.

\*\*If a race condition is detected, one potential resolution is to address it using the sync.Map.

Q12. How do you store key-value pairs in a **sync.Map**?

Ans.

* Key-value pairs are stored in a **sync.Map** using the **Store** method.

*var m sync.Map*

*m.Store(key, value)*

* retrieve the value for a given key 🡪 v*al, ok := m.Load(key)*
* delete a key-value pair 🡪 *m.Delete(key)*
* iterate over all key-value pairs 🡪

*m.Range(func(key, value interface{}) bool {*

*// Process key-value pair*

*return true // Return false to stop iteration*

*})*

Channels

Q1. What is the purpose of channels in Go?

Ans.

Channels in Go facilitate communication and synchronization between Goroutines, allowing safe data exchange.

Q2. How do you close a channel, and why is it important?

Ans.

Closing a channel is done using the **close** function. It signals that no more data will be sent on the channel. It's crucial for receivers to detect when no more data is coming to avoid deadlocks.

*ch := make(chan int)*

*close(ch)*

Q3. How do you check if a channel is closed when receiving data?

Ans.

You can use the second return value when receiving data from a channel. If it's **false**, the channel is closed.

*data, ok := <-ch*

*if !ok {*

*// Channel is closed*

*}*

Q4. What is the purpose of ranging over a channel?

Ans.

Ranging over a channel allows you to iterate over the values sent until the channel is closed, making it a convenient way to receive all values.

*ch := make(chan int)*

*go func() {*

*defer close(ch)*

*for i := 1; i <= 3; i++ {*

*ch <- i*

*}*

*}()*

*for val := range ch {*

*fmt.Println(val)*

*}*

Q5. What is the **select** statement in Go channels?

Ans.

The **select** statement allows Goroutines to wait on multiple communication operations, making it possible to handle multiple channels simultaneously without blocking.

Q6. How does the **select** statement work?

Ans.

The **select** statement randomly picks a case (communication operation) that is ready to proceed. If multiple cases are ready, one is chosen randomly. If none are ready, it blocks until at least one is ready.

*ch1 := make(chan int)*

*ch2 := make(chan string)*

*go func() {*

*ch1 <- 42*

*}()*

*go func() {*

*ch2 <- "Hello"*

*}()*

*select {*

*case value := <-ch1:*

*fmt.Println("Received from ch1:", value)*

*case message := <-ch2:*

*fmt.Println("Received from ch2:", message)*

*}*

Concurrency Patterns

Q1. What are concurrency patterns, and why are they important in Go? Ans.

Concurrency patterns in Go refer to common strategies for managing concurrent tasks efficiently. They are crucial for achieving parallelism and responsiveness in programs with Goroutines, ensuring proper synchronization and communication.

Q2. Explain the concept of worker pools in concurrent programming. Ans.

Worker pools involve a group of worker Goroutines that perform tasks concurrently. Tasks are submitted to a shared pool, and workers pick up and execute these tasks. This pattern helps manage resources, control concurrency, and improve performance.

Q3. How does the worker pool pattern contribute to efficient concurrency management?

Ans.

The worker pool pattern optimizes concurrency by reusing a fixed number of workers to execute tasks concurrently. It prevents the creation of excessive Goroutines, thereby avoiding resource exhaustion and improving overall system performance.

Q4. What is the significance of using contexts and cancellation in concurrent programming?

Ans.

Contexts in Go provide a way to carry deadlines, cancelation signals, and other request-scoped values across API boundaries and between processes. Cancellation allows for the graceful termination of Goroutines, preventing resource leaks and enhancing system responsiveness.

Q5. How can you implement context and cancellation in Go?

Ans.

The **context** package in Go facilitates the implementation of contexts and cancellation. By creating a context and associating it with Goroutines, you can gracefully handle cancelation requests, manage deadlines, and pass values across the Goroutine hierarchy.

Q6. Why is context and cancellation essential when dealing with long-running concurrent tasks?

Ans.

Context and cancellation are crucial for handling long-running tasks in a controlled manner. They allow for the timely termination of Goroutines when needed, preventing unnecessary resource consumption and ensuring the overall stability of the concurrent system.

GO STANDARD LIBRARY

1. **Base Packages:**

* **fmt**: Printf, Println, Sprintf, Errorf, Scan, Scanf, Scanln, Fscan, Fscanf, Fscanln, etc.
* **os**: Create, Open, Remove, Mkdir, Chdir, Getenv, Exit, etc.
* **time**: Now, Since, Sleep, Parse, Format, After, etc.
* **math**: Abs, Sqrt, Sin, Cos, Tan, Ceil, Floor, Max, Min, etc.
* **sync**: Mutex, WaitGroup, Once, Cond, etc.

1. **String Manipulation Functions:**

* **strings**: Contains, Join, Split, Trim, Replace, ToLower, ToUpper, etc.

1. **Trigonometry Functions:**

* **math**: Sin, Cos, Tan, Asin, Acos, Atan, etc.

1. **Random Number Generation:**

* **math/rand**: Intn, Int31, Int63, Float32, Float64, Perm, Seed, etc.

1. **File Operations:**

* **os**: Create, Open, Remove, Mkdir, Chdir, Stat, etc.

1. **Working with URLs:**

* **net/url**: Parse, ParseRequestURI, QueryEscape, QueryUnescape, etc.

1. **Networking HTTP Get, Post:**

* **net/http**: Get, Post, NewRequest, DefaultClient, Serve, FileServer, etc.

1. **JSON Encoding and Decoding:**

* **encoding/json**: Marshal, Unmarshal, NewDecoder, NewEncoder, etc.

1. **XML Coding and Decoding:**

* **encoding/xml**: Marshal, Unmarshal, NewDecoder, NewEncoder, etc.