**Go Concurrency**

**Go routine:**

A Goroutine is a function or method which executes independently and simultaneously in connection with any other Goroutines present in your program. Or in other words, every concurrently executing activity in Go language is known as a Goroutines.

**Syntax:**

func name(){

// statements

}

go name()

**Program:**

package main

import (

"fmt"

"time"

)

func display(str string) {

for w := 0; w < 6; w++ {

time.Sleep(1 \* time.Second)

fmt.Println(str)

}

}

func main()

go display("Welcome")

display("GeeksforGeeks")

}

**Output:**

Welcome

GeeksforGeeks

GeeksforGeeks

Welcome

Welcome

GeeksforGeeks

GeeksforGeeks

Welcome

Welcome

GeeksforGeeks

GeeksforGeeks

**Anonymous Goroutine:**

you can create an anonymous Goroutine simply by using go keyword as a prefix of that function as shown in the below Syntax:

**Syntax:**

// Anonymous function call

go func (parameter\_list){

// statement

}(arguments)

**Syntax:**

// Anonymous function call

go func (parameter\_list){

// statement

}(arguments)

**Program:**

package main

import (

"fmt"

"time"

)

func main() {

fmt.Println("Welcome!! to Main function")

// Creating Anonymous Goroutine

go func() {

fmt.Println("Welcome!! to GeeksforGeeks")

}()

time.Sleep(1 \* time.Second)

fmt.Println("GoodBye!! to Main function")

}

**Output:**

Welcome!! to Main function

Welcome!! to GeeksforGeeks

GoodBye!! to Main function

**Multiple Goroutine:**

In Go language, you are allowed to create multiple goroutines in a single program. You can create a goroutine simply by using go keyword as a prefixing to the function

**Syntax:**

func name(){

// statements

}

go name()

**Program:**

package main

import (

"fmt"

"time"

)

func Aname() {

arr1 := [4]string{"Rohit", "Suman", "Aman", "Ria"}

for t1 := 0; t1 <= 3; t1++ {

time.Sleep(150 \* time.Millisecond)

fmt.Printf("%s\n", arr1[t1])

}

}

// For goroutine 2

func Aid() {

arr2 := [4]int{300, 301, 302, 303}

for t2 := 0; t2 <= 3; t2++ {

time.Sleep(500 \* time.Millisecond)

fmt.Printf("%d\n", arr2[t2])

}

}

func main() {

fmt.Println("!...Main Go-routine Start...!")

go Aname()

go Aid()

time.Sleep(3500 \* time.Millisecond)

fmt.Println("\n!...Main Go-routine End...!")

}

**Output:**

!...Main Go-routine Start...!

Rohit

Suman

Aman

300

Ria

301

302

303

!...Main Go-routine End...!

**Channel in Golang:**

A Channel is a technique which allows to let one goroutine to send data to another goroutine.

By default channel is bidirectional, means the goroutines can send or receive data through the same channel



**Creating a Channel:**

A Channel is created using chan keyword and it can only transfer data of the same type, different types of data are not allowed to transport from the same channel.

**Syntax:**

var Channel\_name chan Type

You can also create a channel using make() function using a shorthand declaration.

channel\_name:= make(chan Type)

**Program:**

// Go program to illustrate

// how to create a channel

package main

import "fmt"

func main() {

// Creating a channel

// Using var keyword

var mychannel chan int

fmt.Println("Value of the channel: ", mychannel)

fmt.Printf("Type of the channel: %T ", mychannel)

// Creating a channel using make() function

mychannel1 := make(chan int)

fmt.Println("\nValue of the channel1: ", mychannel1)

fmt.Printf("Type of the channel1: %T ", mychannel1)

}

**Output:**

Value of the channel:

Type of the channel: chan int

Value of the channel1: 0x432080

Type of the channel1: chan int

**Send and Receive Data From a Channel:**

Channel work with two principal operations one is sending and another one is receiving, both the operations collectively known as communication.

**1.Send operation:**

Mychannel <- element

**2.Receive Operation:**

element := <-Mychannel

**Program:**

package main

import "fmt"

func myfunc(ch chan int) {

fmt.Println(234 + <-ch)

}

func main() {

fmt.Println("start Main method")

// Creating a channel

ch := make(chan int)

go myfunc(ch)

ch <- 23

fmt.Println("End Main method")

}

**Output:**

start Main method

257

End Main method

**Closing a Channel:**

You can also close a channel with the help of close() function

**Syntax:**

ele, ok:= <- Mychannel

**Program:**

package main

import "fmt"

// Function

func myfun(mychnl chan string) {

for v := 0; v < 4; v++ {

mychnl <- "GeeksforGeeks"

}

close(mychnl)

}

// Main function

func main() {

// Creating a channel

c := make(chan string)

// calling Goroutine

go myfun(c)

for {

res, ok := <-c

if ok == false {

fmt.Println("Channel Close ", ok)

break

}

fmt.Println("Channel Open ", res, ok)

}

}

**Output:**

Channel Open GeeksforGeeks true

Channel Open GeeksforGeeks true

Channel Open GeeksforGeeks true

Channel Open GeeksforGeeks true

Channel Close false

**Length of the Channel**:

In Channel, you can find the length of the channel using len() function. Here, the length indicates the number of value queued in the channel buffer.

**Program:**

package main

import "fmt"

func main() {

mychnl := make(chan string, 4)

mychnl <- "GFG"

mychnl <- "gfg"

mychnl <- "Geeks"

mychnl <- "GeeksforGeeks"

// Using len() function

fmt.Println("Length of the channel is: ", len(mychnl))

}

**Output:**

Length of the channel is: 4

**Capacity of the channel:**

In channel, you can find the capacity of the channel using cap() function. Here, the capacity indicates the size of the buffer.

Program:

// Go program to illustrate how to

// find the capacity of the channel

package main

import "fmt"

// Main function

func main() {

// Creating a channel

// Using make() function

mychnl := make(chan string, 5)

mychnl <- "GFG"

mychnl <- "gfg"

mychnl <- "Geeks"

mychnl <- "GeeksforGeeks"

// Finding the capacity of the channel

// Using cap() function

fmt.Println("Capacity of the channel is: ", cap(mychnl))

}

**Output:**

Capacity of the channel is:5

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

**Mutex in Golang:**

A Mutex is a method used as a locking mechanism to ensure that only one Goroutine is accessing the critical section of code at any point of time. This is done to prevent race conditions from happening.

Two methods defined on Mutex

\*Lock

\*Unlock

**Syntax:**

mutex.Lock()

x = x + 1 // this statement be executed

// by only one Goroutine

// at any point of time

mutex.Unlock()

**Program:**

package main

import (

"fmt"

"sync"

)

var GFG = 0

func worker(wg \*sync.WaitGroup, m \*sync.Mutex) {

m.Lock()

GFG = GFG + 1

m.Unlock()

// On return, notify the

// WaitGroup that we’re done.

wg.Done()

}

func main() {

var w sync.WaitGroup

var m sync.Mutex

for i := 0; i < 1000; i++ {

w.Add(1)

go worker(&w, &m)

}

w.Wait()

fmt.Println("Value of x", GFG)

}

**Output:**

Value of x 1000

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

**Atomic Variable in Golang:**

Atomic Variables are utilized in order to control state. Here, the “sync/atomic” package must be used to use these variables. Moreover, it also prevents race conditions which allows two or more Goroutines to access identical sources.

**Program:**

// Including main package

package main

import (

"fmt"

"math/rand"

"sync"

"sync/atomic"

"time"

)

var waittime sync.WaitGroup

var atmvar int32

func hike(S string) {

// For loop

for i := 1; i < 7; i++ {

time.Sleep(time.Duration((rand.Intn(5))) \* time.Millisecond)

atomic.AddInt32(&atmvar, 1)

fmt.Println(S, i, "count ->", atmvar)

}

// Wait completed

waittime.Done()

}

// Main function

func main() {

waittime.Add(2)

// Calling hike method with

// values

go hike("cat: ")

go hike("dog: ")

// Calling wait method

waittime.Wait()

// Prints the value of last count

fmt.Println("The value of last count is :", atmvar)

}

**Output:**

dog: 1 count -> 1

cat: 1 count -> 2

dog: 2 count -> 3

dog: 3 count -> 4

cat: 2 count -> 5

cat: 3 count -> 6

cat: 4 count -> 7

cat: 5 count -> 8

cat: 6 count -> 9

dog: 4 count -> 10

dog: 5 count -> 11

dog: 6 count -> 12

The value of last count is : 12