Go Lang

Presented by Siddharth Nema

Why use GO?

Go (also known as Golang) is a programming language developed at Google by **Robert Pike** and **Ken Thompson** that has gained popularity for its simplicity, efficiency, and concurrency support.

01. Go has a clean and simple syntax that reduces the cognitive load on developers, making code easier to read and maintain.

02. Go is **statically typed**, ensuring type safety, but it also supports type inference, allowing developers to write code without explicitly declaring variable types in many cases.

03. Go has built-in support for **concurrency** using goroutines and channels.

GO Syntax

The simplicity

If-Else

```
if day == "sunday" || day == "saturday" {
 rest()
} else if day == "monday" && isTired() {
 groan()
} else {
 work()
}
```

For-Loop

```
for count := 0; count <= 10; count++ {
 fmt.Println("My counter is at", count)
}
```

Arrays

```
numbers := [...]int{0, 0, 0, 0, 0}
```

Switch-case

```
switch day {
 case "sunday":
  // cases don't "fall through" by default!
  fallthrough

 case "saturday":
  rest()

 default:
  work()
}
```

Function Declaration

```
func split(sum int) (x, y int) {
 x = sum * 4 / 9
 y = sum - x
 return
}
```

Sequential Programming

So far in most programs, we've been dealing with sequential programs:

- The order of executing instructions is based on their textual ordering
- Overlap or interruption to execute another instruction is not allowed
- Cannot proceed to next instruction until current instruction is finished

A typical order of execution in a Go program

```go
func main() {
    fmt.Println("Hello world")

    numbers := []int{1,2,3,4,5}

    calculate_mean(numbers)
    calculate_median(numbers)
    calculate_mode(numbers)

}
```

1

2

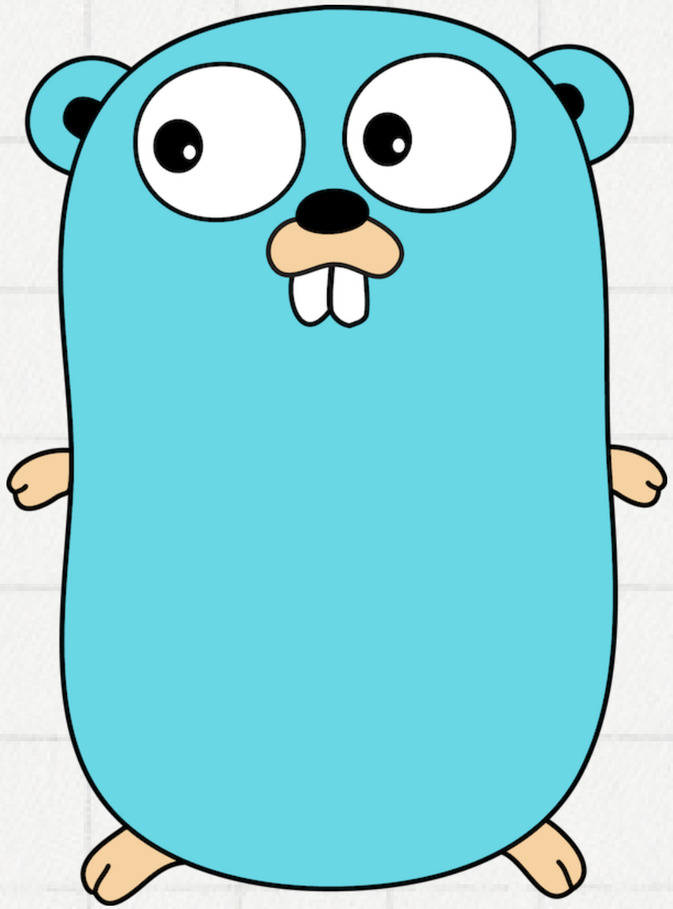
3

4

5

Concurrency

Concurrency is a crucial aspect of modern programming, essential for achieving high performance and efficient resource utilization. Among programming languages that facilitate concurrency, Go, or Golang, stands out for its unique approach.
Concurrency in computing enables different parts of a program to execute independently, potentially improving performance and allowing better use of system resources.



Sequential Programming

1.Write a letter
2.Listen to an audiobook
3.Go grocery shopping
4.Go for a run
5.Watch 5 Episodes of a Netflix Show
6.Book flight tickets
Pay your bills

Concurrent Programming

1.Watch 2 Netflix Episodes
2.Go for a run
3.Write 1/3 of the letter
4.Listen to half an audiobook
5.Go Grocery Shopping
6.Listen to other half of audiobook
7.Write 1/3 of the letter
8.Book flight tickets
9.Write 1/3 of the letter
10.Pay your bills
11.Watch 3 Netflix Episodes

Concurrency

Concurrency means that an application is making progress on more than one task at the same time (concurrently).

Concurrency means executing multiple tasks at the same time but not necessarily simultaneously.

Two tasks can start, run, and complete in overlapping time periods i.e Task-2 can start even before Task-1 gets completed.

Parallelism

Parallelism means that an application splits its tasks up into smaller subtasks which can be processed in parallel, for instance on multiple CPUs at the exact same time.

It literally physically run parts of tasks OR multiple tasks, at the same time using the multi-core infrastructure of CPU, by assigning one core to each task or sub-task.

Why Concurrency

IO-Bound

A workload that causes functions (goroutines) to naturally enter into waiting states

API Calls

A workload that waits for external API services.

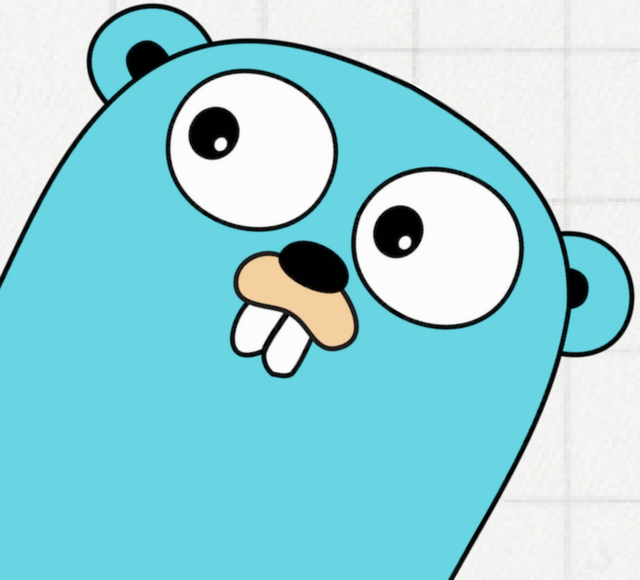
Implementation

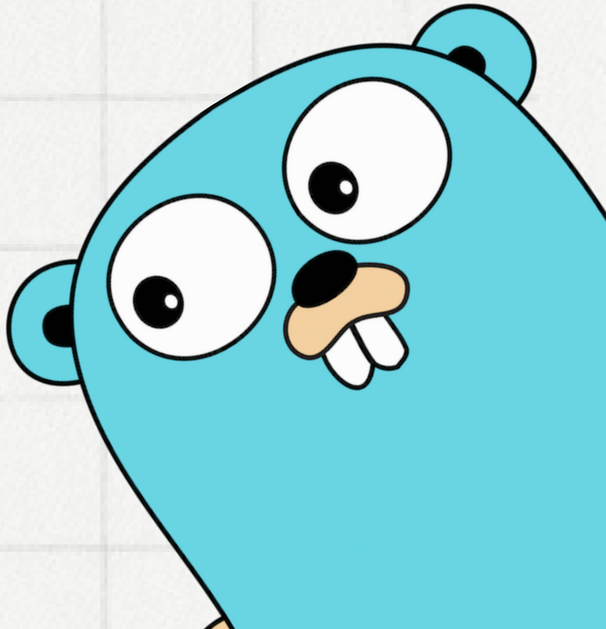
Now, let’s **go** about implementing concurrent programming in Go.

Goroutines

A goroutine is a lightweight thread managed by the Go runtime. You can create your own Goroutine simply by using go keyword as a prefix to the function or method call as shown in the below syntax:

```
go f(x, y, z)
```





Example

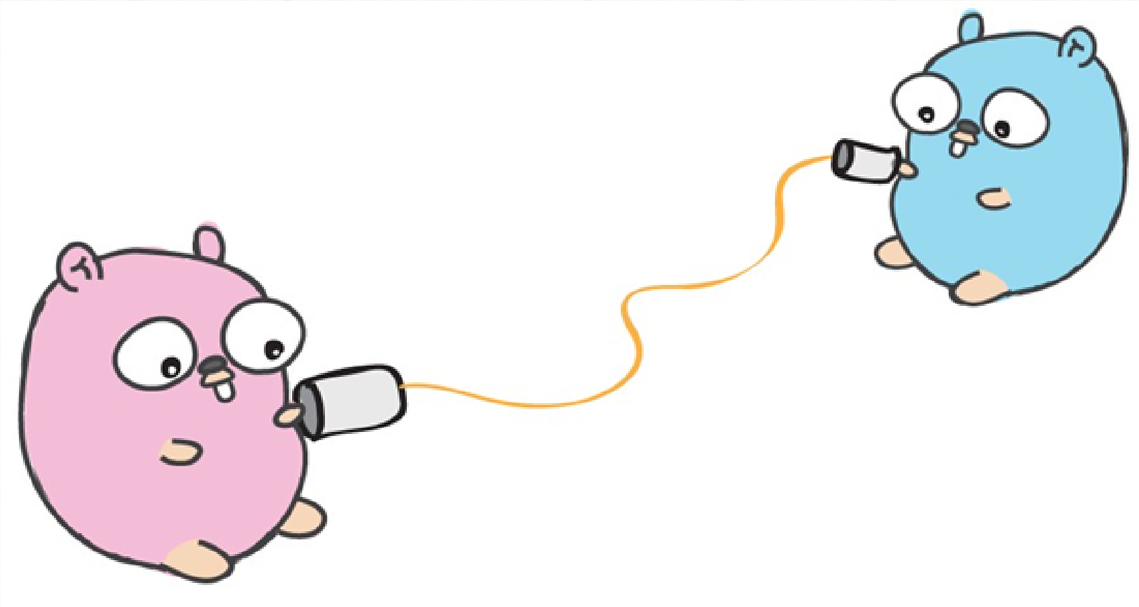
```
package main
import (
    "fmt"
    "time"
)
func main() {
    fmt.Println("Welcome!! to Main function")
    // Creating Anonymous Goroutine
    go func() {
        fmt.Println("Welcome!! to GeeksforGeeks")
    }()

    time.Sleep(1 * time.Second)
    fmt.Println("GoodBye!! to Main function")
}
```

Channels

Channels provide a way for two goroutines to communicate with one another

If you send a message on a channel, the goroutine sending the message is **“frozen” (blocked)** until the other goroutine is ready to receive the message (and vice-versa)

Similarly, the receiving goroutine is also frozen until it receives the message from the channel



Initialization

(The make function takes the type of channel corresponding to the type of value it sends and receives)

```
message := make(chan string)
```

Passing to Channel

```
message <- "Hello from Goroutine"
```

Recieving from Channel

```
fmt.Println(<-message)
```

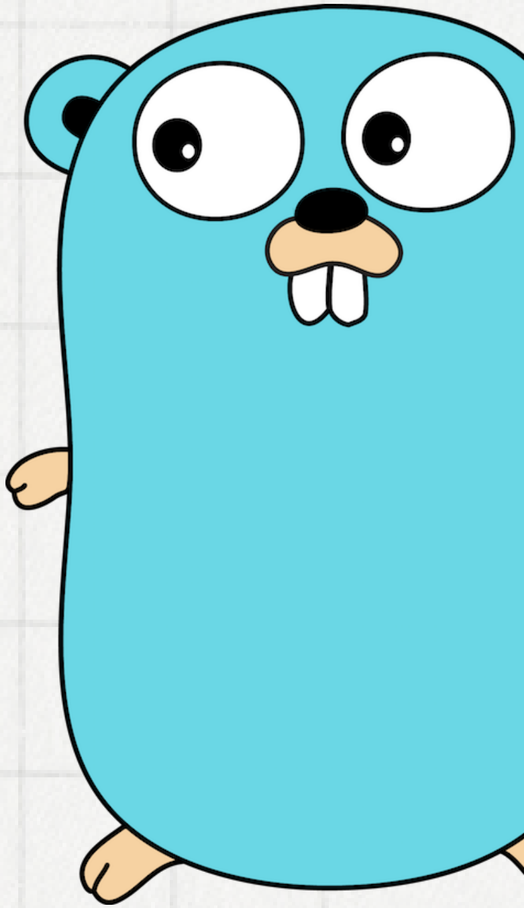
```go
package main

import (
 "fmt"
 "time"
)

func printMessage(message chan string) {
 time.Sleep(time.Second * 2)
 message <- "Hello from Goroutine"
}

func main() {
 message := make(chan string)
 go printMessage(message)
 fmt.Println("Hello from main function")
 fmt.Println(<-message)
}
```

sync package

The sync package offers primitives like WaitGroups, Mutexes, and Once that can help control the execution of Goroutines. We use these instead of trying to manually synchronize Goroutines, which can lead to subtle bugs.

WaitGroups

A WaitGroup waits for a collection of Goroutines to finish.
The main goroutine calls **Add** to set the number of goroutines to wait for. Then, each of the goroutines runs and calls **Done** when finished. At the same time, **Wait** can be used to block until all goroutines have finished.
It offers a simple, yet powerful way to coordinate multiple goroutines in the main driver function.



Example

```go
package main

import (
    "fmt"
    "sync"
    "time"
)

var wg sync.WaitGroup

func worker(id int) {
    defer wg.Done()
    fmt.Printf("Worker %d starting\n", id)
    time.Sleep(time.Second)
    fmt.Printf("Worker %d done\n", id)
}
```

```go
func main() {
    for i := 1; i <= 5; i++ {
        wg.Add(1)
        go worker(i)
    }
    wg.Wait()
}
```

Mutex

A mutex, short for **mutual exclusion**, is used to protect shared resources from simultaneous access by multiple goroutines. It ensures that only one goroutine can access a critical section of code at a time.
Mutexes prevent race conditions between goroutines by allowing only one goroutine to acquire the lock and access the shared resource, while other goroutines wait until the lock is released.

“If I Tell you in what order goroutines run, it won’t happen” - Dr Gopher Strange.

Example

```go
package main

import (
  "fmt"
  "sync"
)

var counter = 0
var mutex sync.Mutex

func increment(wg *sync.WaitGroup) {
  mutex.Lock()
  counter++
  mutex.Unlock()
  wg.Done()
}
```

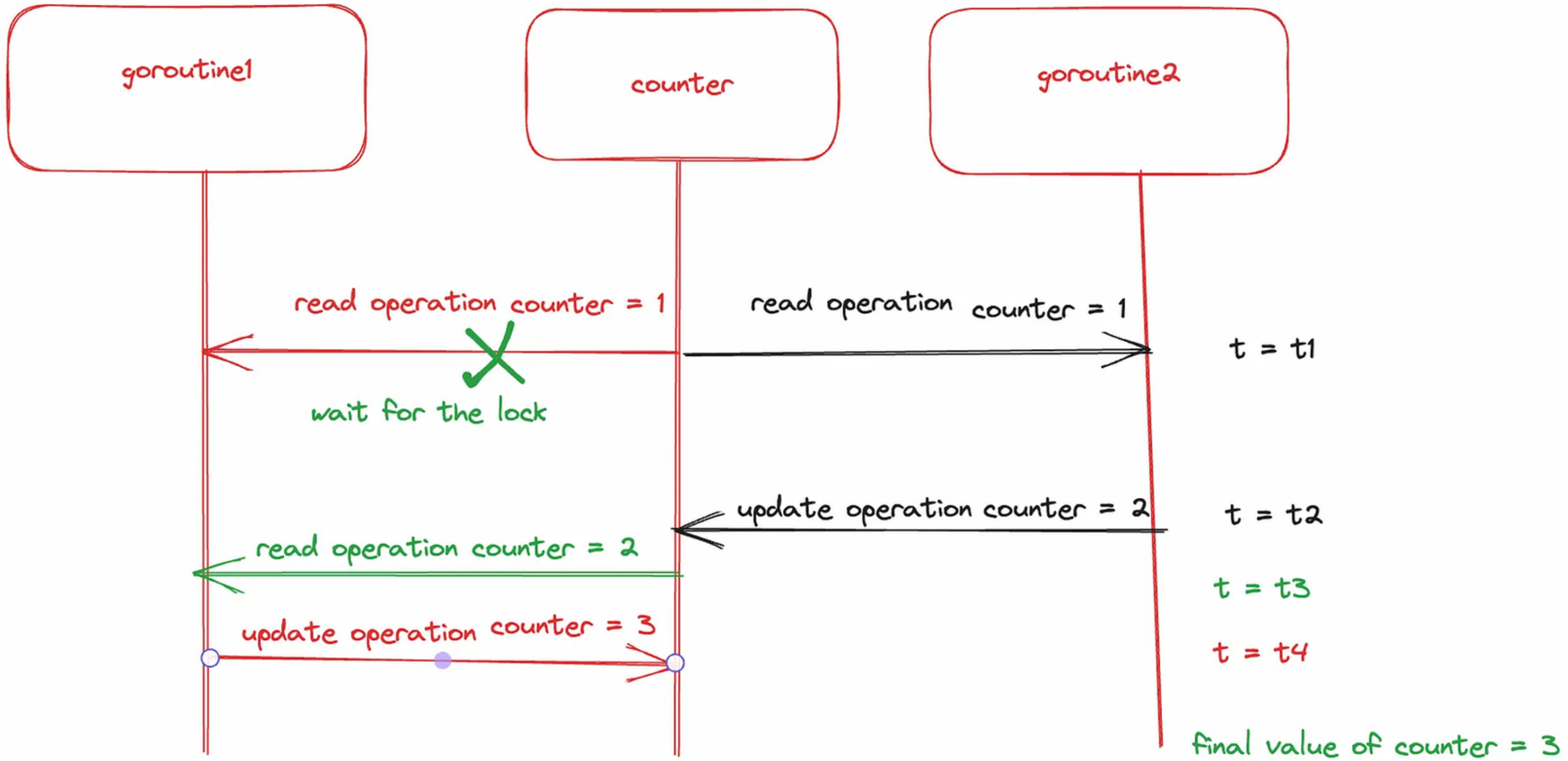
```go
func main() {
var wg sync.WaitGroup
expectedCounter := 1000

for i := 0; i < expectedCounter; i++ {
wg.Add(1)
go increment(&wg)
}

wg.Wait()
fmt.Println("Expected Counter:",
expectedCounter)
fmt.Println("Actual Counter:", counter)
}
```

Output:

```
Expected Counter: 1000
Actual Counter: 1000
```

Without the use of Mutex, the actual counter comes out less than the expected counter because the race conditions between the multiple goroutines causes the initial counter value to be the same for multiple goroutines. Thus, the increment in value remains ineffective

In brief, this is what Mutex does



Thank you very much!