Walchand College of Engineering, Sangli

Walchand Linux Users' Group

WLUG
COMMUNITY | KNOWLEDGE | SHARE

METAMORPHOSIS

An Insightful Workshop

9th & 10th March 2024

DOCKER

- Basics Of Containerization
- Dockerfile
- Networking & Compose

GOLANG

- Basics Of GoLang
- GoRoutines & Concurrency
- Setting up a GoLang Project

CONNECT WITH US

Main & Mini CCF

LIMITED SEATS

REGISTER AT

EXCITING PRIZES

Entry Fee: Rs 299/-

meta2k24.wcewlug.org

Mr. D. N. Gangji
President
Walchand Linux Users' Group

Dr. M. A. Shah
HoD Computer Science and Engineering

Dr. R. R. Rathod
HoD Information Technology

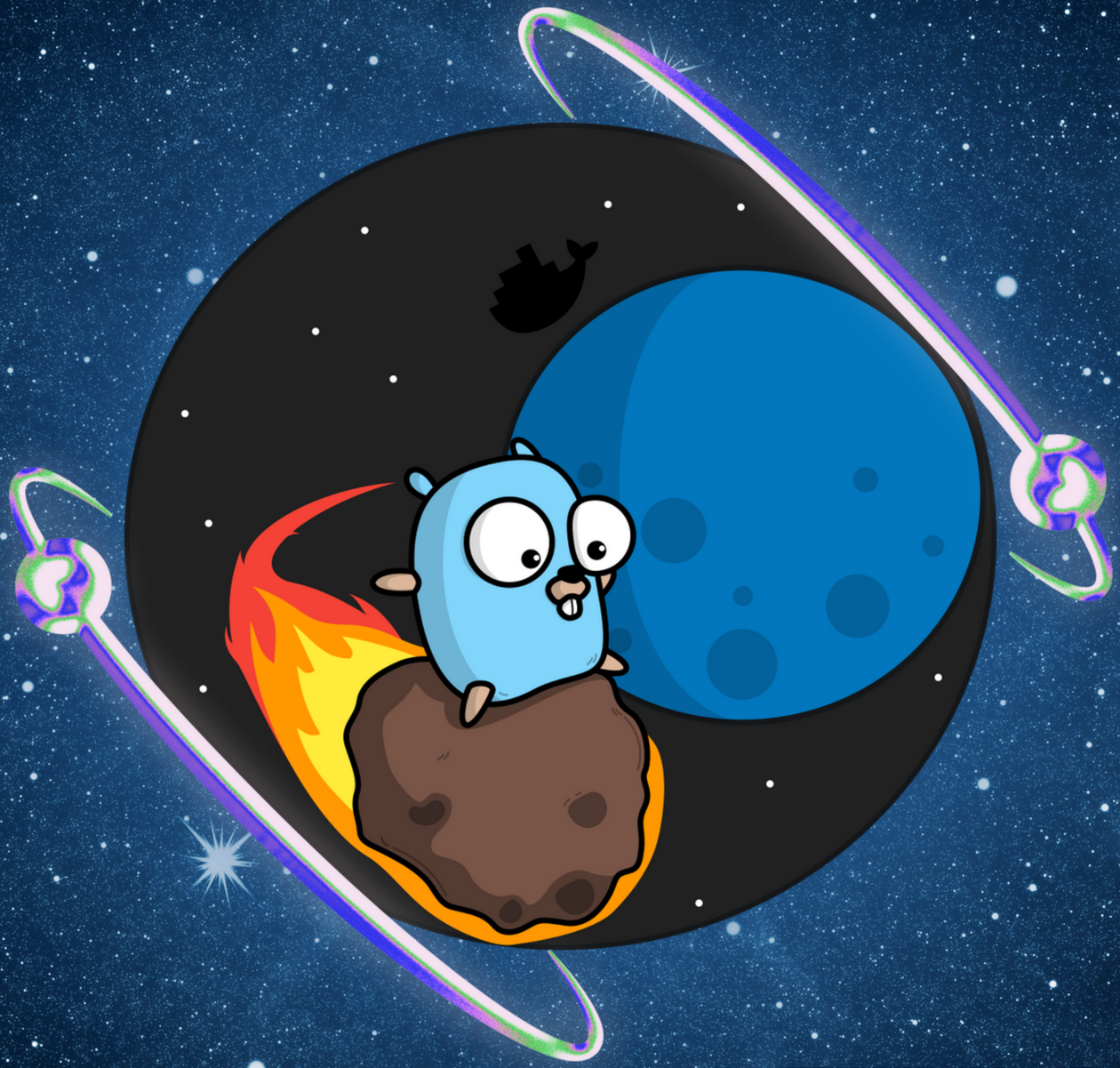
Dr. A. J. Umbarkar
Staff Advisor
Walchand Linux Users' Group

Dr. A. R. Surve
Assoc. Dean Student Activities and Staff Advisor

Dr. U. A. Dabade
I/C Director
Walchand College of Engineering

WLUG

COMMUNITY | KNOWLEDGE | SHARE



SESSION 1

LET'S GOPHER IT!!

WLUG
COMMUNITY | KNOWLEDGE | SHARE

TABLE OF CONTENTS

Introduction

Advanced Concepts

Fundamentals

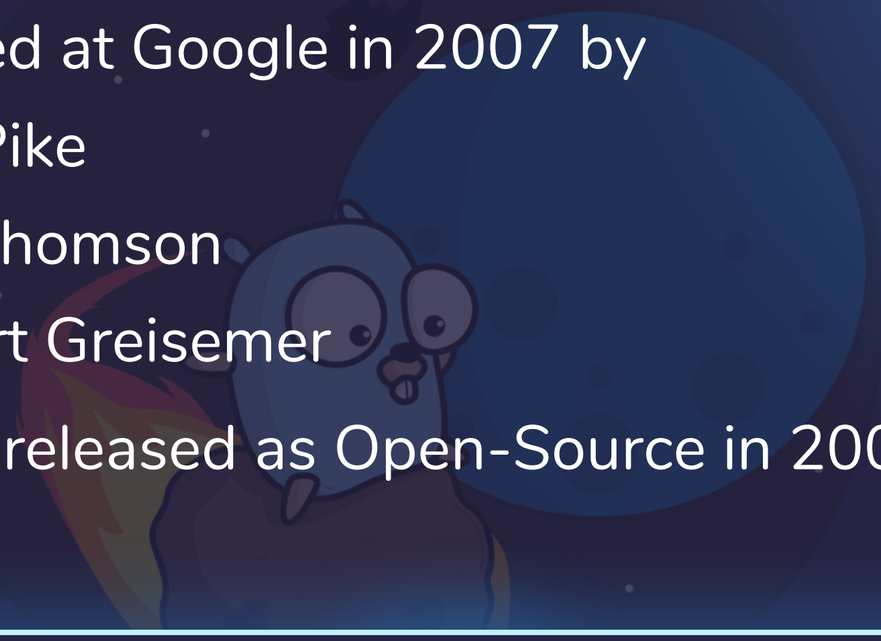
Concurrency

Project

WLUG
COMMUNITY | KNOWLEDGE | SHARE

GOLANG

- Developed at Google in 2007 by
  - ➤ Rob Pike
  - ➤ Ken Thomson
  - ➤ Robert Greisemer
- Officially released as Open-Source in 2009



Born - 2007

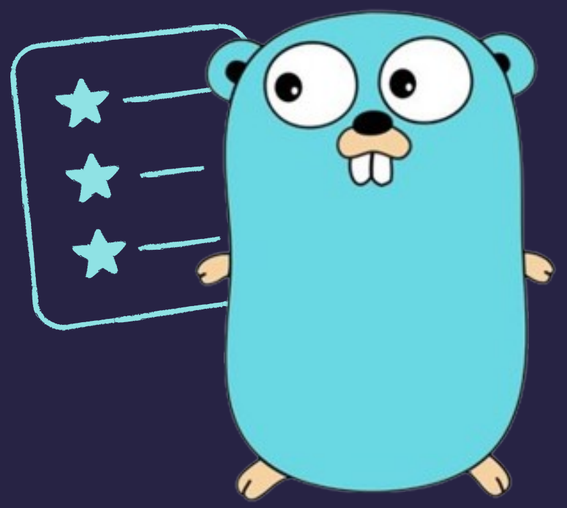
Released as Open-source - 2009

First stable release (Go1**)** - 2012

WLUG
COMMUNITY | KNOWLEDGE | SHARE

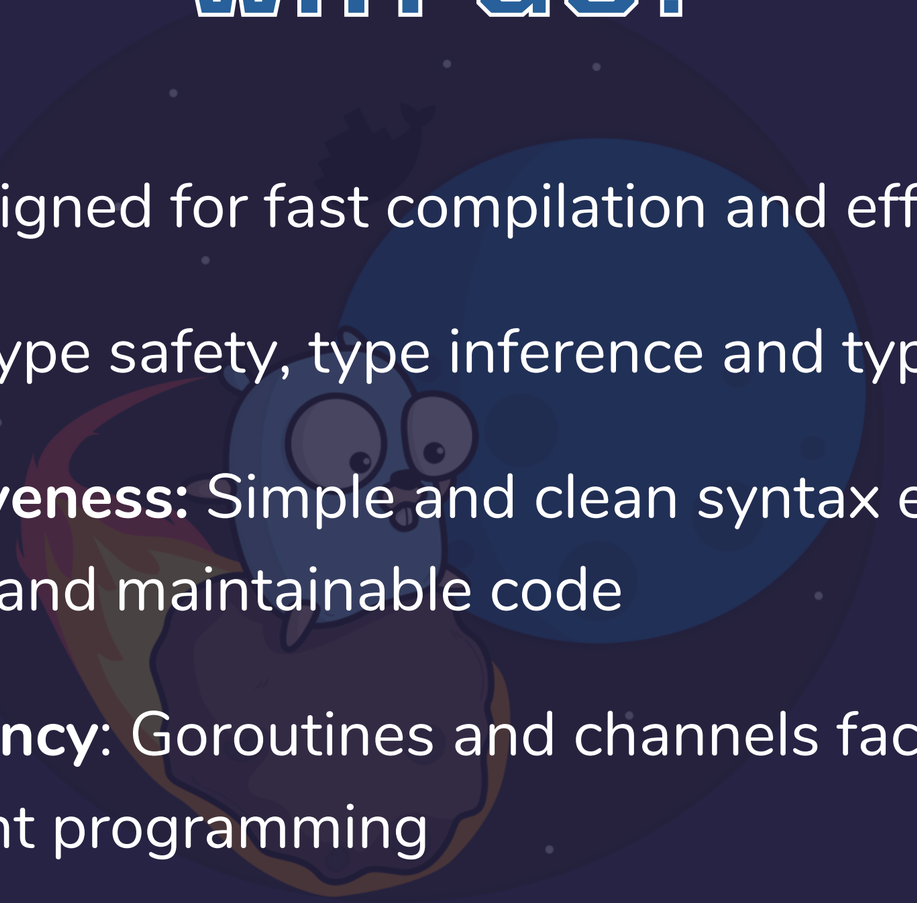
FEATURES

- Open-Source
- Compiled Language
- Cross-Platform
- Statically Typed
- Garbage Collection



WLUG
COMMUNITY | KNOWLEDGE | SHARE

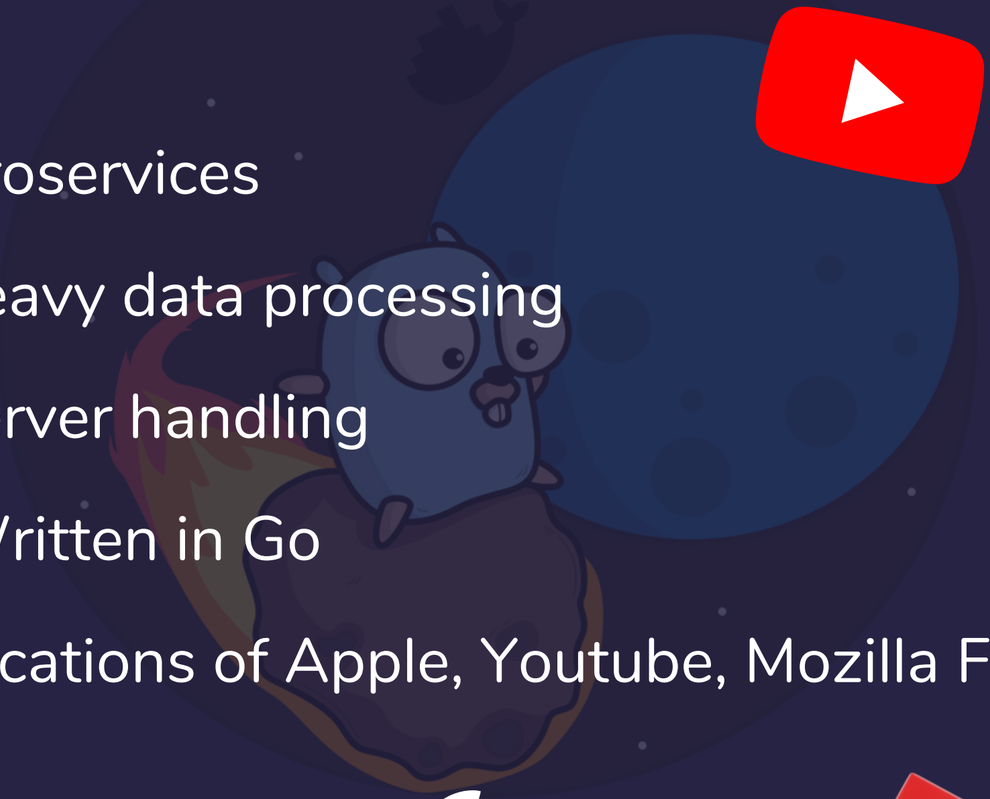
WHY GO?



- **Fast**: Designed for fast compilation and efficient runtime
- **Safety:** Type safety, type inference and type annotation
- **Expressiveness:** Simple and clean syntax encourages readable and maintainable code
- **Concurrency**: Goroutines and channels facilitate concurrent programming

WLUG
COMMUNITY | KNOWLEDGE | SHARE

WHO USES GO



- Uber - Microservices
- Netflix - Heavy data processing
- Adobe - Server handling
- Docker - Written in Go
- Other applications of Apple, Youtube, Mozilla Firefox

WLUG
COMMUNITY | KNOWLEDGE | SHARE

Installation

```
sudo apt install golang
```

WLUG
COMMUNITY | KNOWLEDGE | SHARE

Hello World!!

```go
package main

import "fmt"

func main() {
	name := "Hello World"
	fmt.Println(name)
}
```

WLUG
COMMUNITY | KNOWLEDGE | SHARE

package main

```go
package main

import "fmt"

func main() {
	name := "Hello World"
	fmt.Println(name)
}
```

- Collection of common source code files
- Inside package there can be multiple files

WLUG
COMMUNITY | KNOWLEDGE | SHARE

import “fmt”

- Import formatting package

```
package main

import "fmt"

func main() {
	name := "Hello World"
	fmt.Println(name)
}
```

WLUG
COMMUNITY | KNOWLEDGE | SHARE

func main( ){ ... }

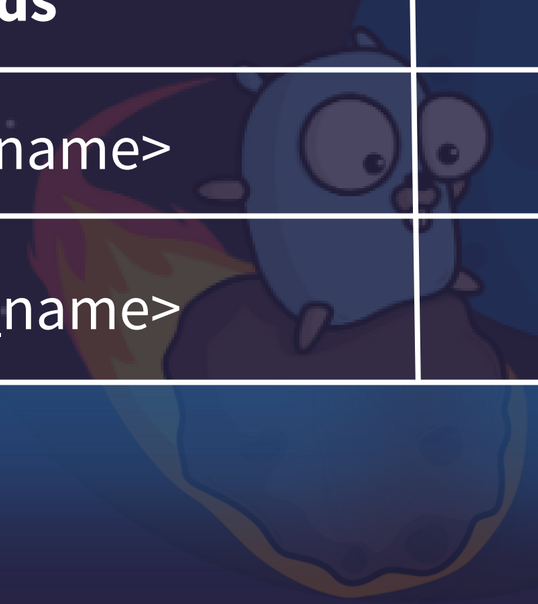
```go
package main

import "fmt"

func main() {
	name := "Hello World"
	fmt.Println(name)
}
```

- Entry point for executable program

WLUG
COMMUNITY | KNOWLEDGE | SHARE

How do we run the code?

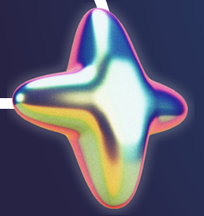
Commands	Functions
go run <file_name>	Runs the program
go build <file_name>	Builds programs into binaries



WLUG
COMMUNITY | KNOWLEDGE | SHARE

FUNDAMENTALS OF GO

- Data Types & Variables
- If/Else
- For Loop
- Switch Statement
- Arrays & Slices
- Maps



WLUG
COMMUNITY | KNOWLEDGE | SHARE

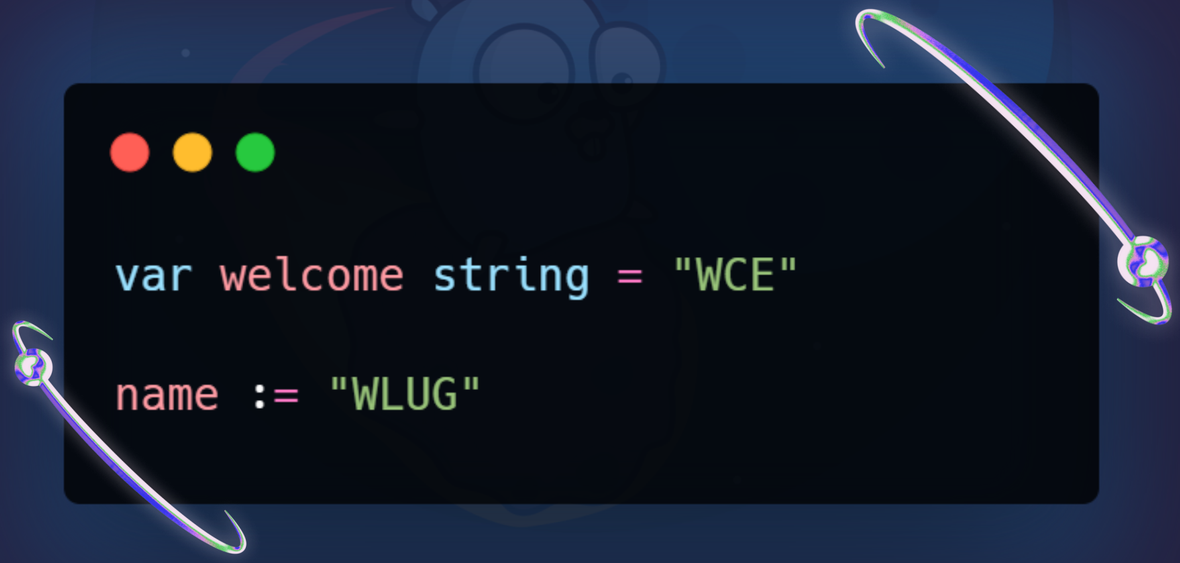
DATA TYPES

- Integer
- Float
- Boolean
- String

WLUG
COMMUNITY | KNOWLEDGE | SHARE

VARIABLES

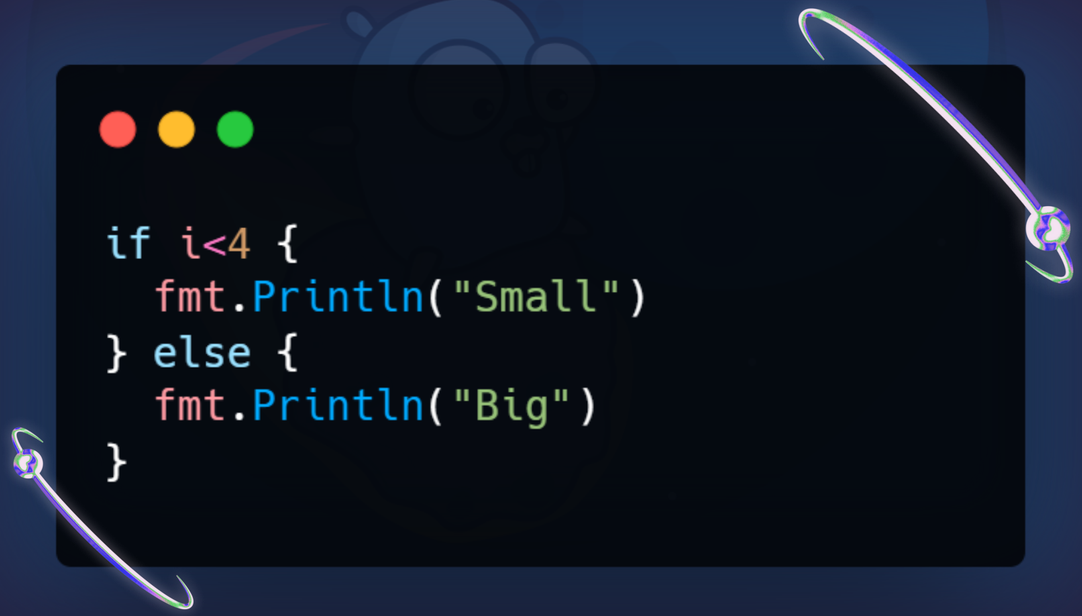
- Variables stores data
- Uninitialized declarations default to zero



WLUG
COMMUNITY | KNOWLEDGE | SHARE

IF/ELSE

- Conditional Statement
- No parentheses needed for conditions

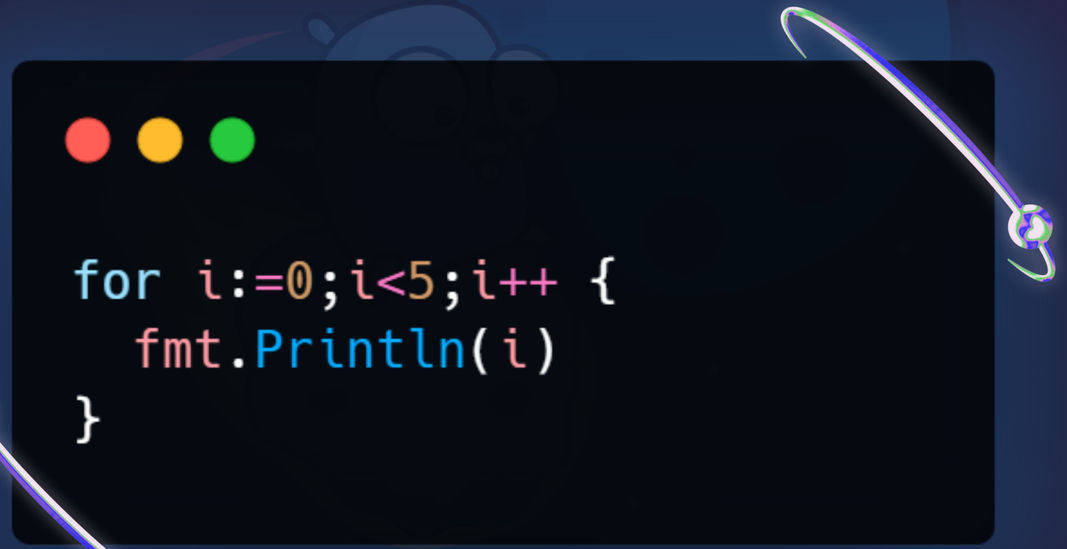


```
if i<4 {
  fmt.Println("Small")
} else {
  fmt.Println("Big")
}
```

WLUG
COMMUNITY | KNOWLEDGE | SHARE

FOR LOOP

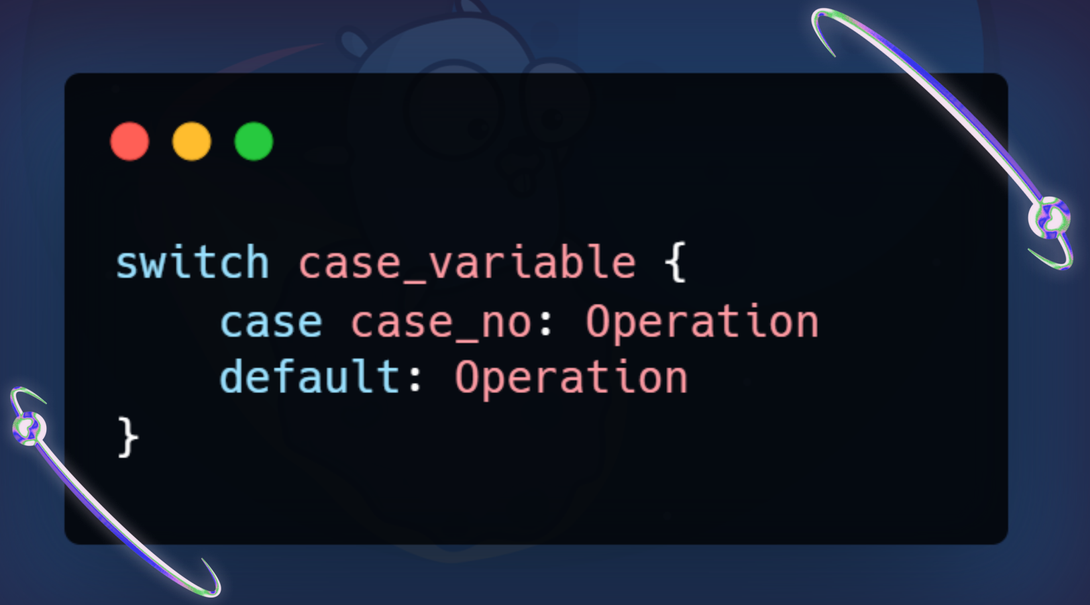
- Repeats code based on conditions
- Go’s only looping construct



WLUG
COMMUNITY | KNOWLEDGE | SHARE

SWITCH STATEMENT

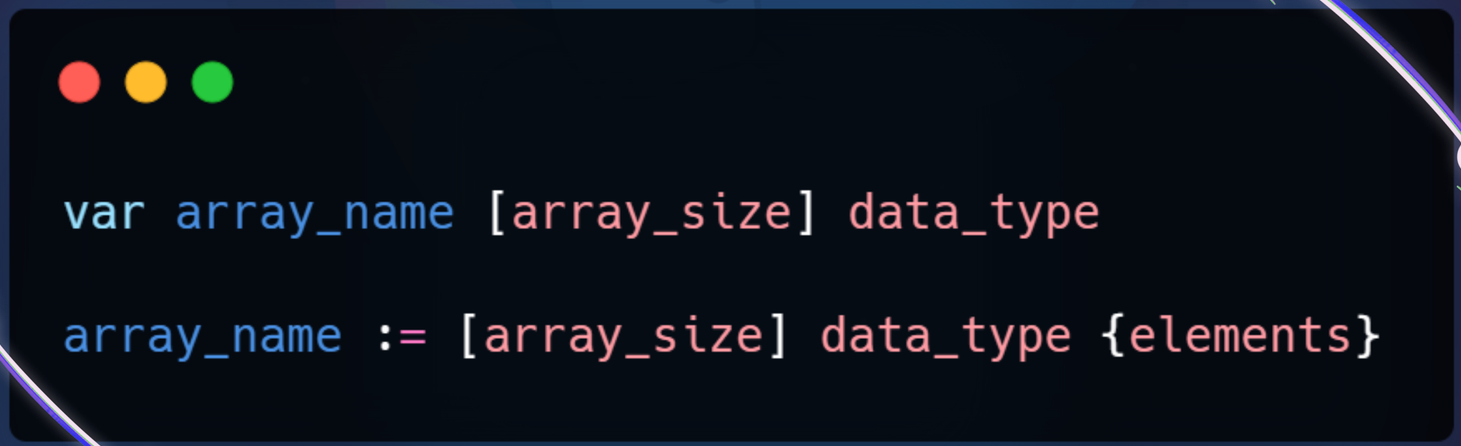
- Condense conditionals over multiple branches
- Separate multiple expressions



WLUG
COMMUNITY | KNOWLEDGE | SHARE

ARRAYS

- Array is a sequential collection of elements
- Arrays have fixed size
- Zero based indexing



```
var array_name [array_size] data_type

array_name := [array_size] data_type {elements}
```

WLUG
COMMUNITY | KNOWLEDGE | SHARE

SLICES

- Variable-length data type
- Appending elements is possible

```
slice_name := make([]data_type, size)

slice_name := []data_type{elements}
```

WLUG
COMMUNITY | KNOWLEDGE | SHARE

MAPS

- Key-Value data type
- Appending elements is possible

```
map_name := make(map[key_data_type]value_data_type)
```

WLUG
COMMUNITY | KNOWLEDGE | SHARE

FUNCTIONS

- Functions are the blocks of code in a program that can perform a specific task
- Functions give users the ability to reuse code
- Increase the readability of our code

WLUG
COMMUNITY | KNOWLEDGE | SHARE

FUNCTION DECLARATION

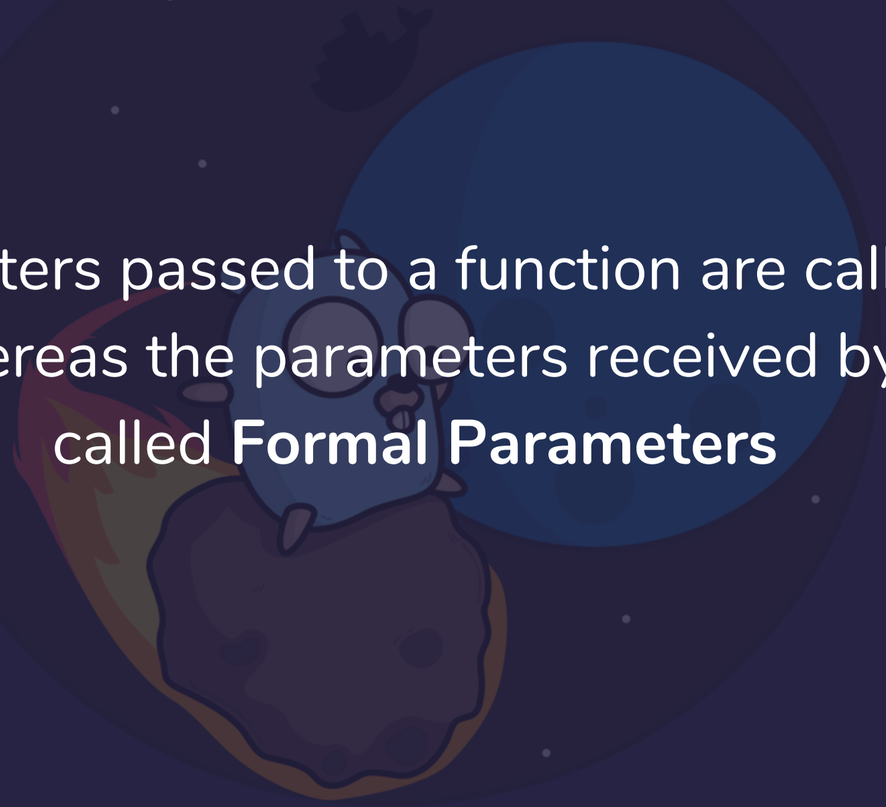
```go
func add(a int, b int) int {
    var res = a + b
    return res
}
```

FUNCTION CALLING

```go
sum := add(7, 5)
```

WLUG
COMMUNITY | KNOWLEDGE | SHARE

FUNCTION ARGUMENTS



The parameters passed to a function are called **Actual Parameters**, whereas the parameters received by a function are called **Formal Parameters**

WLUG
COMMUNITY | KNOWLEDGE | SHARE

MULTIPLE RETURN TYPES

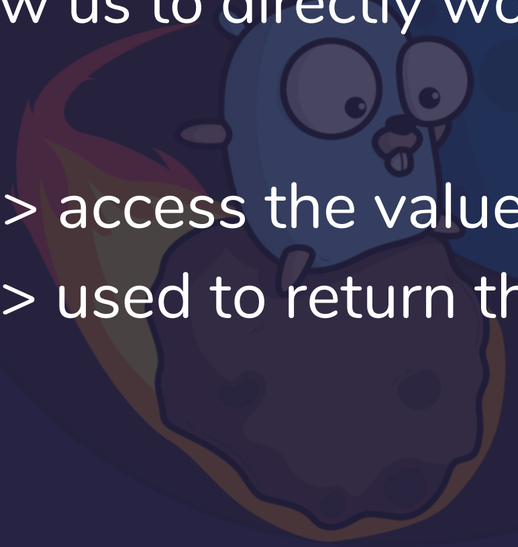
In GoLang, we can return multiple values of the same or different data types at a time from a single function

```
func arithmeticOperations(a, b int) (int, int, int,
float32) {
  var add = a + b
  var sub = a - b
  var mul = a * b
  var div = float32(a) / float32(b)
  return add, sub, mul, div
}
```

WLUG
COMMUNITY | KNOWLEDGE | SHARE

POINTERS

- Pointer is a variable that is used to store the memory address of another variable
- Pointers allow us to directly work with memory addresses

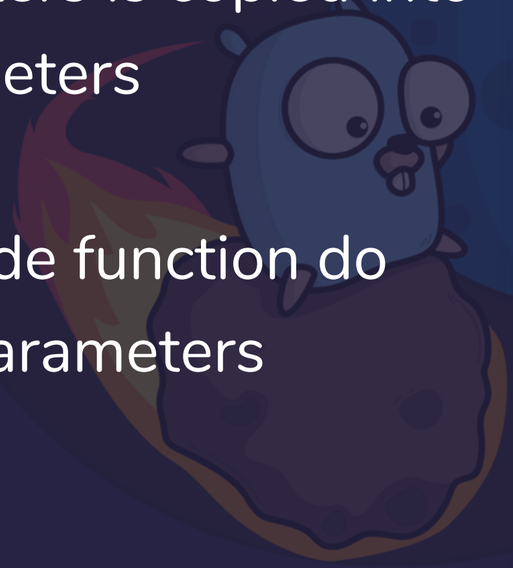


* Operator --> access the value stored in the address
& Operator --> used to return the address of a variable

WLUG
COMMUNITY | KNOWLEDGE | SHARE

CALL BY VALUE

- Value of actual parameters is copied into function’s formal parameters
- Any changes made inside function do **not** impact the actual parameters

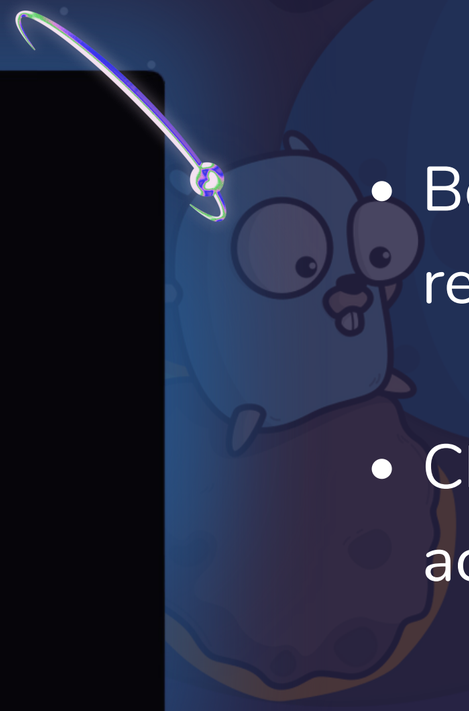


```
func swap(a, b int) {
    var temp int
    temp = a
    a = b
    b = temp
}

swap(a, b)
```

WLUG
COMMUNITY | KNOWLEDGE | SHARE

CALL BY REFERENCE

```
func swap(a, b *int) {
    var temp int
    temp = *a
    *a = *b
    *b = temp
}

swap(&a, &b)
```

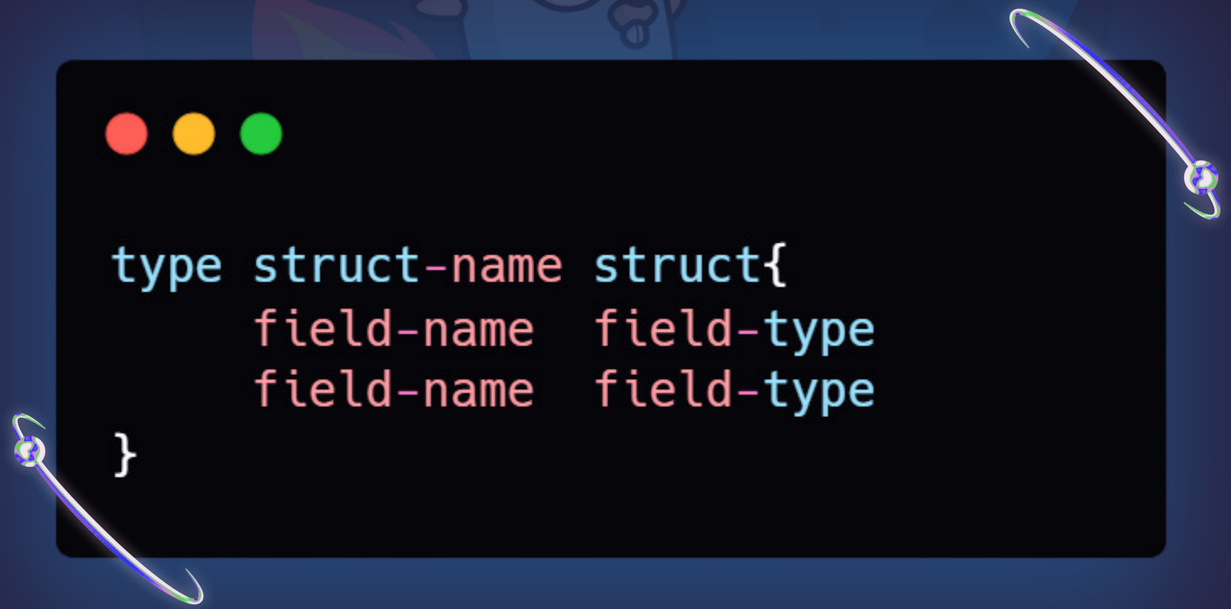


- Both the actual and formal parameters refer to the same location
- Changes made inside the function are actually reflected in the actual parameters

WLUG
COMMUNITY | KNOWLEDGE | SHARE

STRUCTURES

- A structure is a user-defined data type that combines different types of variables into a single type
- It is like a collection of variables, called fields, and every field can have different data type



WLUG
COMMUNITY | KNOWLEDGE | SHARE

STRUCTURE DECLARATION

```
type Player struct {
    firstName string
    lastName  string
    country   string
    age       int
}
```

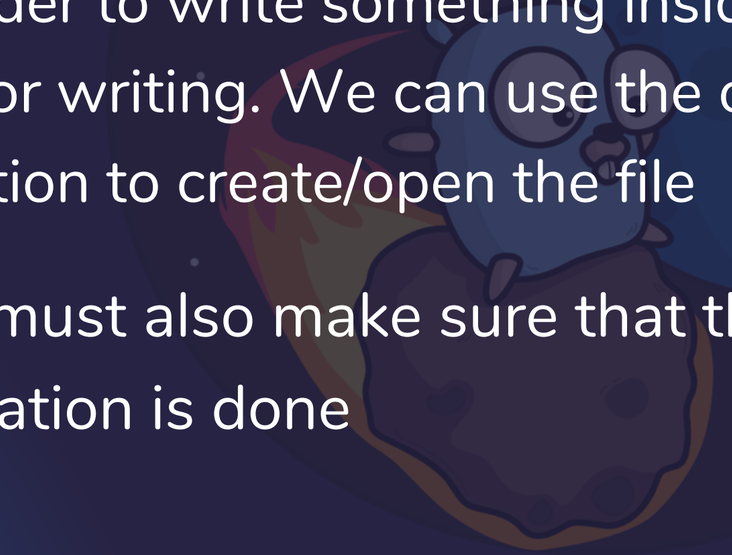
STRUCTURE INITIALIZATION

```
var player1 = Player{"Virat", "Kohli", "India", 30}

var player2 = Player{firstName: "David", country: "Australia", age: 35}

var player3 = Player{}
```

WLUG
COMMUNITY | KNOWLEDGE | SHARE

FILE HANDLING

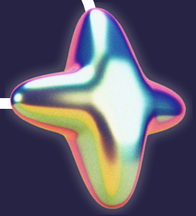
- Golang offers a vast in-built library that can be used to perform read and write operations on files
- In order to write something inside a file, we must create the file for writing. We can use the os package’s Create() function to create/open the file
- We must also make sure that the file is closed after the operation is done



WLUG
COMMUNITY | KNOWLEDGE | SHARE

AGENDA FOR CONCURRENCY IN GO

- Basics
- Goroutines
- WaitGroups
- Defer
- Channels
- Mutexes



WLUG
COMMUNITY | KNOWLEDGE | SHARE

SEQUENTIAL EXECUTION

WLUG
COMMUNITY | KNOWLEDGE | SHARE

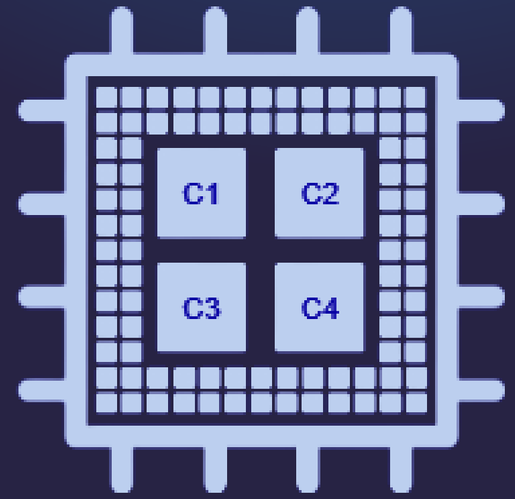
PARALLEL EXECUTION

Core 1

Core 2

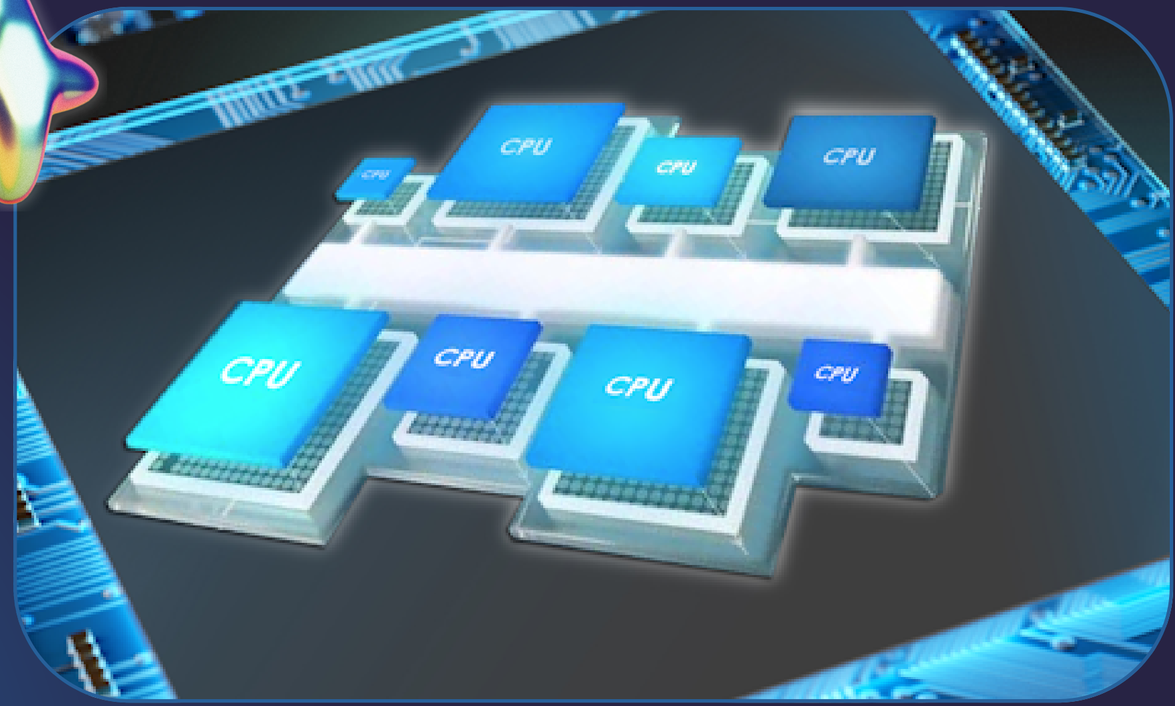
CPU

=



WLUG
COMMUNITY | KNOWLEDGE | SHARE

THEY ARE EXPENSIVE!



WLUG
COMMUNITY | KNOWLEDGE | SHARE

SEQUENTIAL EXECUTION

WLUG

COMMUNITY | KNOWLEDGE | SHARE

CONCURRENT EXECUTION

WLUG
COMMUNITY | KNOWLEDGE | SHARE

PROCESS

A process is a self-contained program in execution

It has its own memory space, code, resources, and stack

THREAD

A thread is the smallest unit of execution within a process

Threads within the same process share the same memory space, code and resources

WLUG

COMMUNITY | KNOWLEDGE | SHARE

```go
package main

import "fmt"

func main() {
    x := 3
    x++
    fmt.Println(x)
}
```

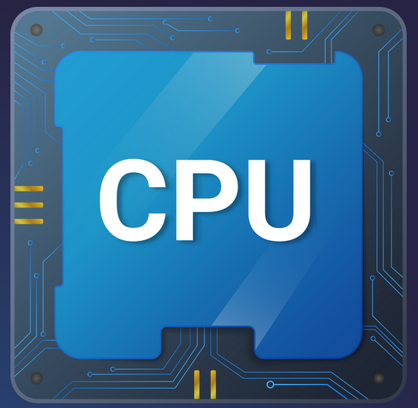
WLUG

COMMUNITY | KNOWLEDGE | SHARE

```go
package main

import "fmt"

func main() {
    x := 3
    x++
    fmt.Println(x)

    y := 10
    y--
    fmt.Println(y)
}
```

WLUG
COMMUNITY | KNOWLEDGE | SHARE

WHO SCHEDULES THREADS?



Go Scheduler

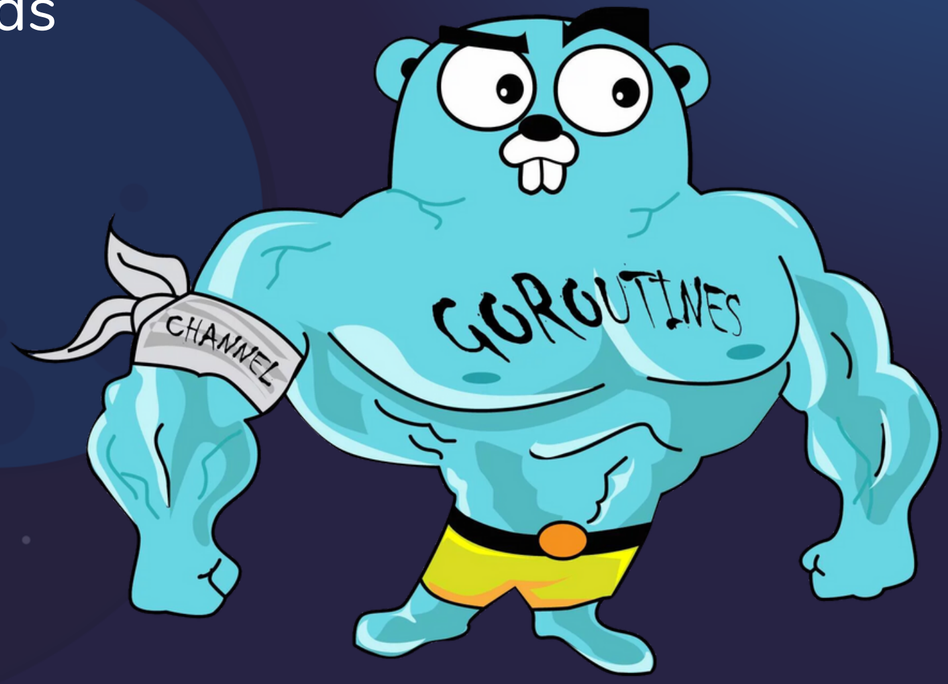
- Thread 1
- Thread 2
- Thread 3
- Thread 4

RAM

WLUG
COMMUNITY | KNOWLEDGE | SHARE

GOROUTINES

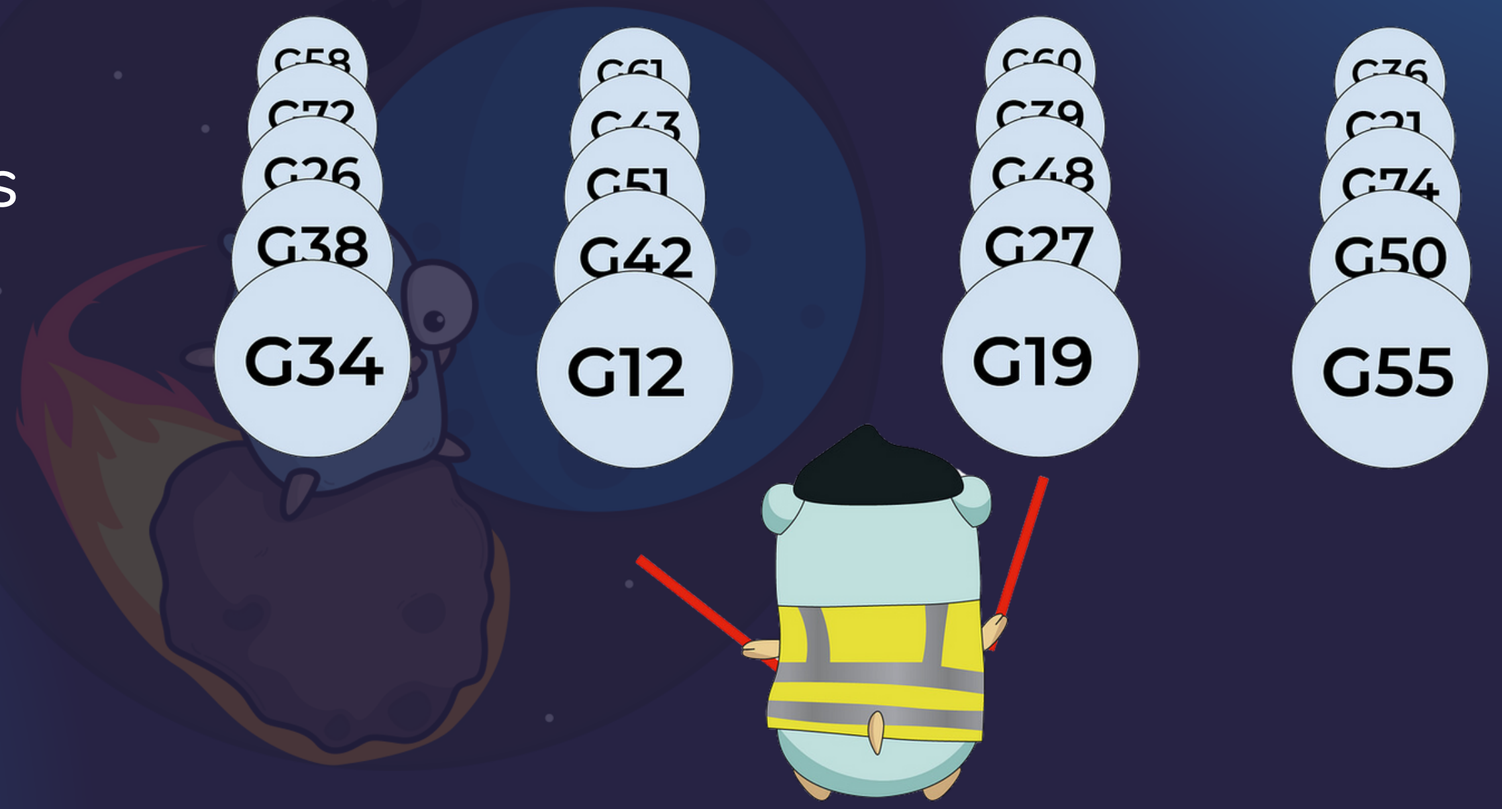
- It is possible to create multiple threads in a go program!
- These lightweight threads are called as goroutines
- Go runtime scheduler manages the concurrent execution of goroutines



WLUG
COMMUNITY | KNOWLEDGE | SHARE

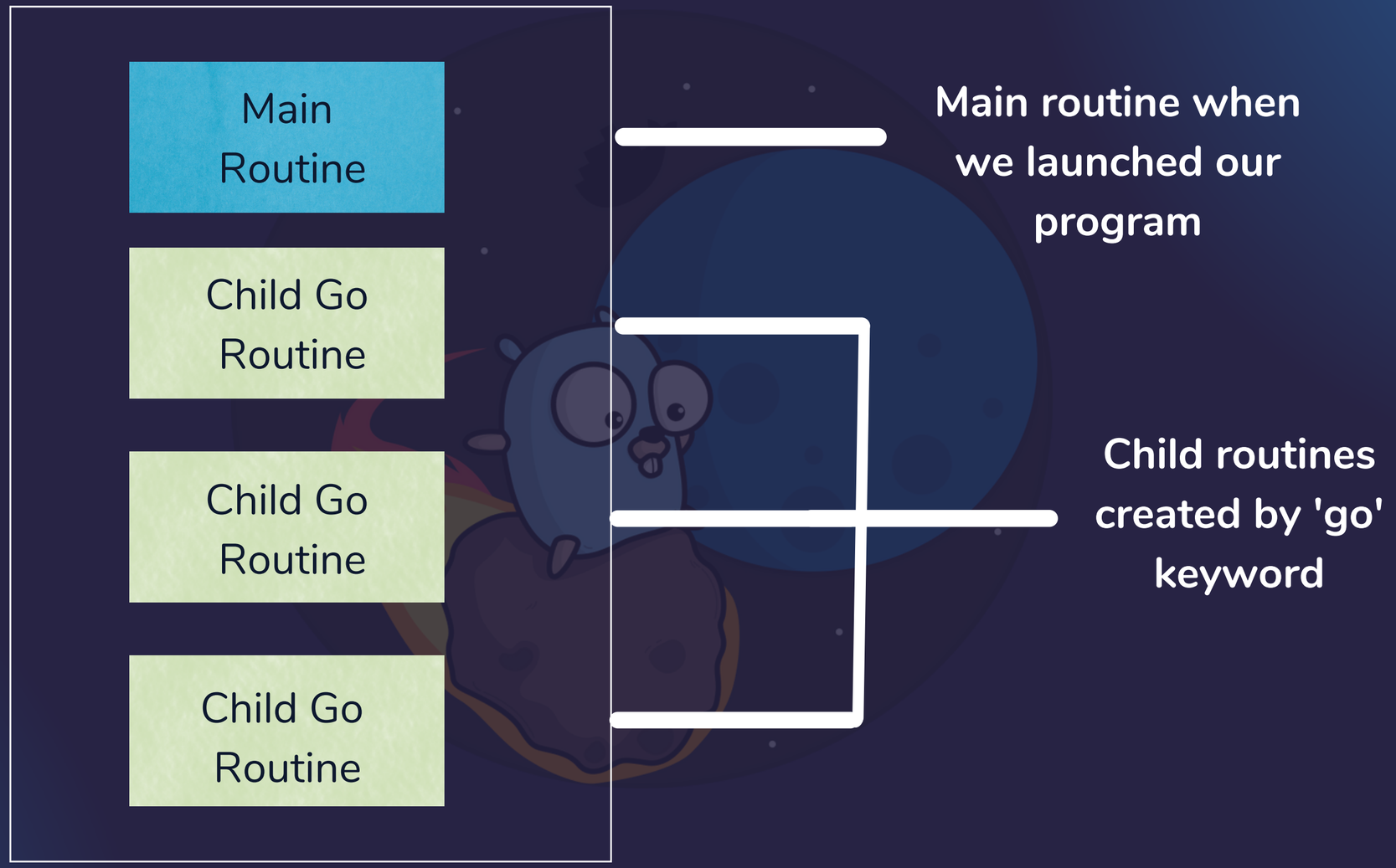
GOROUTINES

1. Main routine
2. Child goroutines
3. Go scheduler



WLUG
COMMUNITY | KNOWLEDGE | SHARE

Our Running Program



WLUG
COMMUNITY | KNOWLEDGE | SHARE

GOROUTINES

Syntax:

```
go function_name()
```

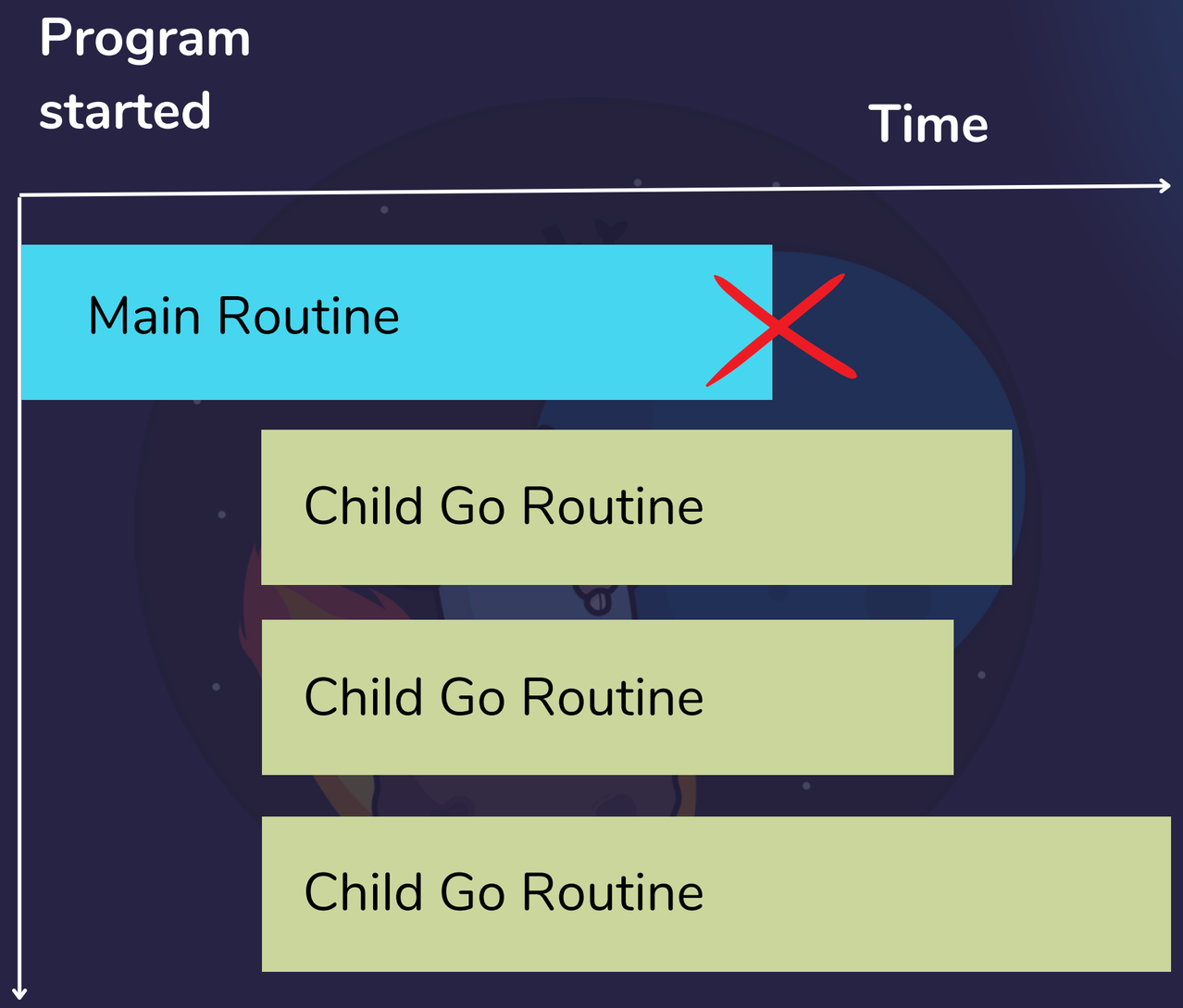
Just add ‘go’ keyword when calling the function to turn it into a goroutine!

WLUG
COMMUNITY | KNOWLEDGE | SHARE

ISSUES

- Main goroutine never cares about the state of other goroutines
- Goroutines never return anything

WLUG
COMMUNITY | KNOWLEDGE | SHARE



WLUG
COMMUNITY | KNOWLEDGE | SHARE

WAITGROUPS

- **WaitGroup:** A synchronization primitive in Go for waiting for multiple goroutines to complete
- **Use Case:** Ensures that the main goroutine waits for all other goroutines to finish their execution before proceeding

WLUG
COMMUNITY | KNOWLEDGE | SHARE

WAITGROUPS

Syntax:

```go
package main

import (
    "sync"
)

func main() {
    var wg sync.WaitGroup

    wg.Add(1)

    go func() {
        defer wg.Done()
        //do something
    }()

    wg.Wait()

    fmt.Println("Done")
}
```

WLUG
COMMUNITY | KNOWLEDGE | SHARE

CHANNELS

- Pipes using which GoRoutines communicate
- A programming construct that allows us to move data between different goroutines

Key Features

- Synchronization
- Data Sharing

WLUG
COMMUNITY | KNOWLEDGE | SHARE

CHANNELS

Syntax:

```go
package main

func main() {
    C := make(chan int)

    go func() {
        a := 0
        //do something
        C <- a
    }()

    output := <-C
}
```

WLUG
COMMUNITY | KNOWLEDGE | SHARE

DATA RACES

x = 8

Goroutine 1

++

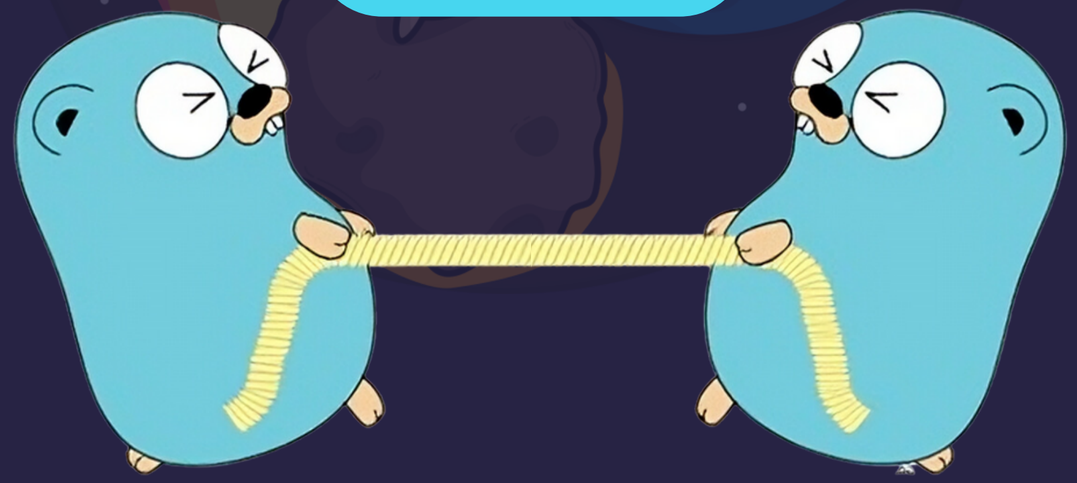
x

8

--

x = 8

Goroutine 2



WLUG
COMMUNITY | KNOWLEDGE | SHARE

MUTEXES

- Stands for "mutual exclusion"
- Lock-Unlock mechanism
- Ensures that only one goroutine can access a critical section of code at a time, preventing data races

Why Use Mutex?

- Concurrency Safety
- Preventing Data Races

WLUG
COMMUNITY | KNOWLEDGE | SHARE

MUTEXES

Syntax:

```
//Creating mutex
var mu sync.Mutex

//Locking a mutex
mu.Lock()

//Unlocking a mutex
mu.Unlock()
```

PROJECT

Main Go Routine

DB

Add Student

Main Go Routine

Add Student

DB

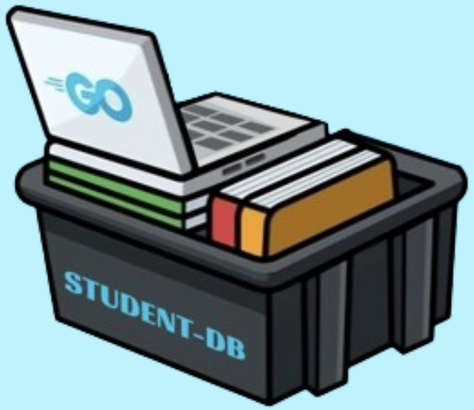
Main Go Routine

Add Student

DB

Main Go Routine

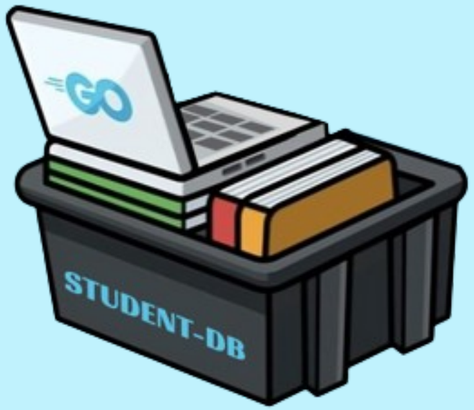
DB



Add Student

Main Go Routine

DB



Add Student

Main Go Routine

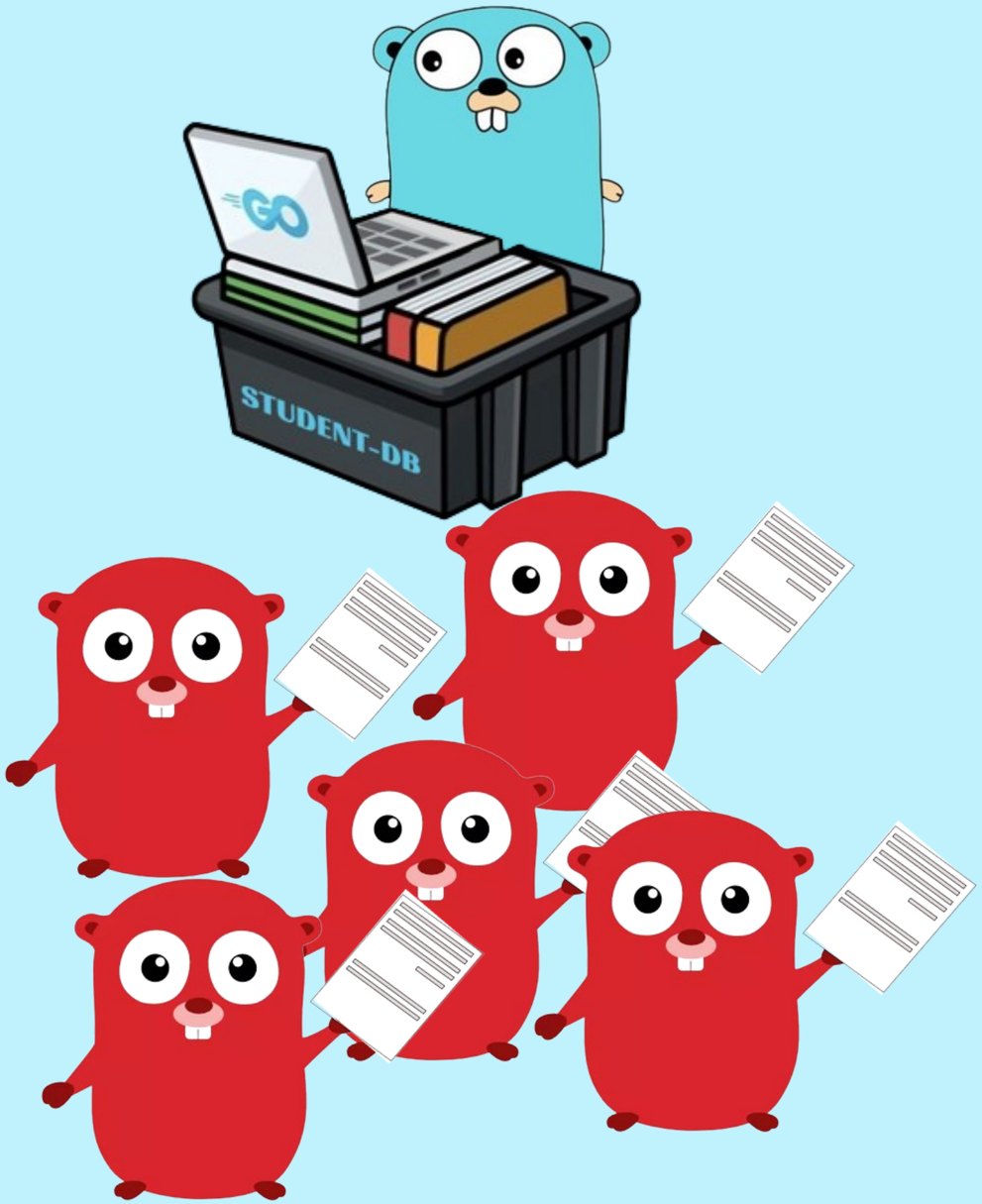
Add Student

DB

Main Go Routine

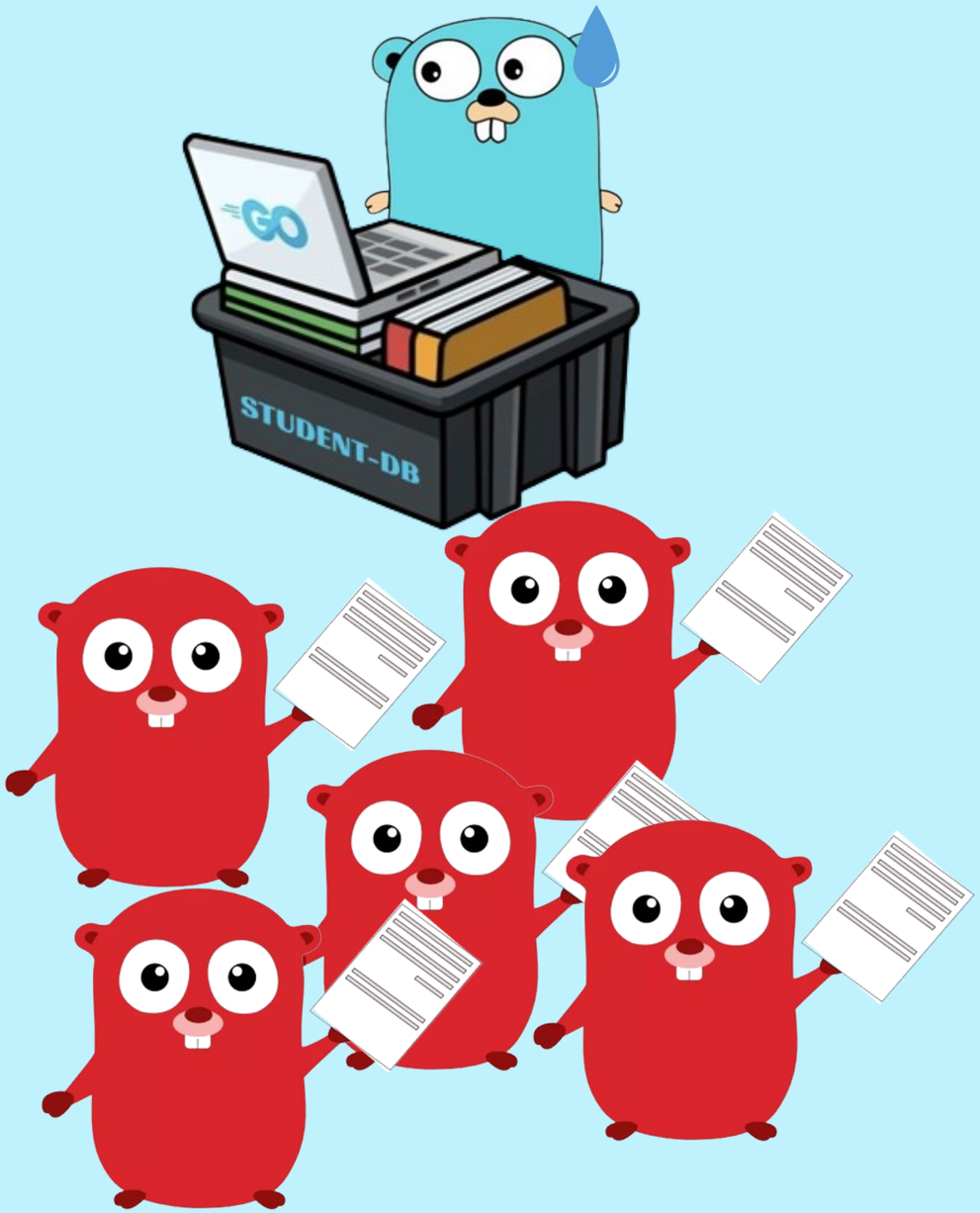
DB

Main Go Routine



DB

Main Go Routine



DB


Go

Main Go Routine

G-1

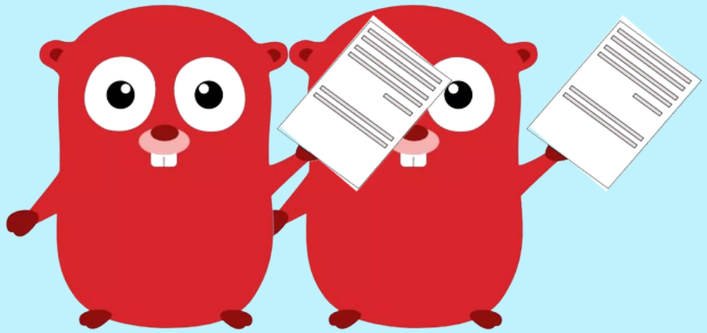
DB

Main Go Routine

G-1

DB

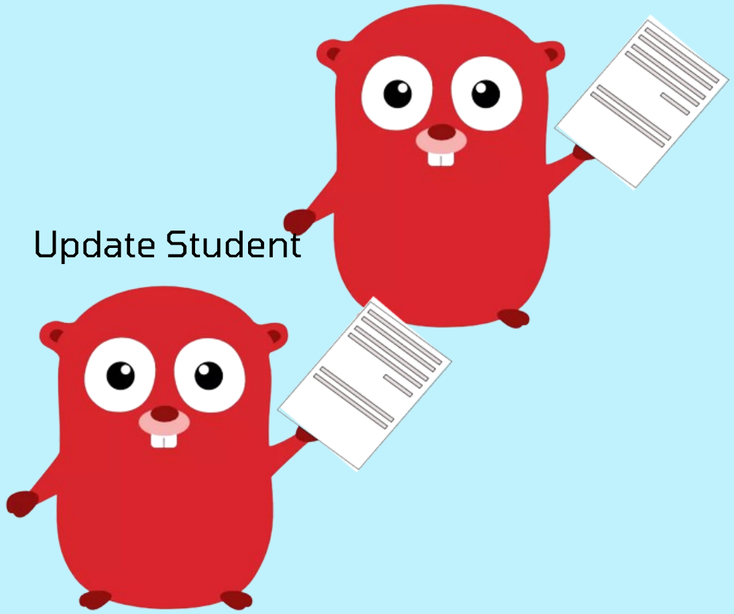
Update Student Add Student



Main Go Routine

Add Student

Update Student



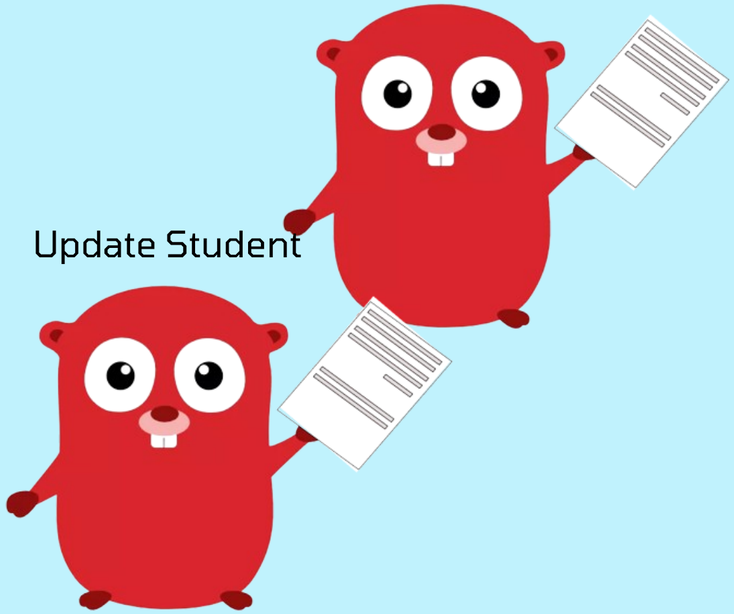
G-1

DB

Main Go Routine

Add Student

Update Student



G-1

DB

Main Go Routine

G-1

DB

Wait !

Add Student

Update Student

Main Go Routine

Update Student

G-1

DB

Wait !

Add Student

Main Go Routine

G-1

DB

Wait !

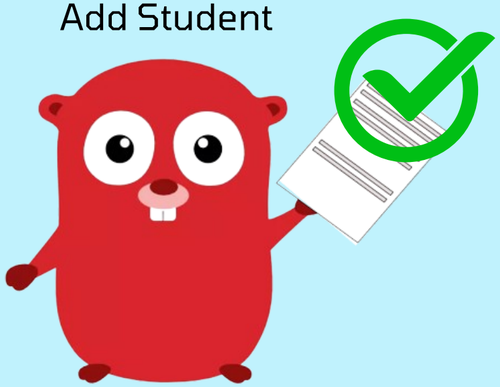
Add Student

Main Go Routine

G-1

DB

Add Student



THANK YOU