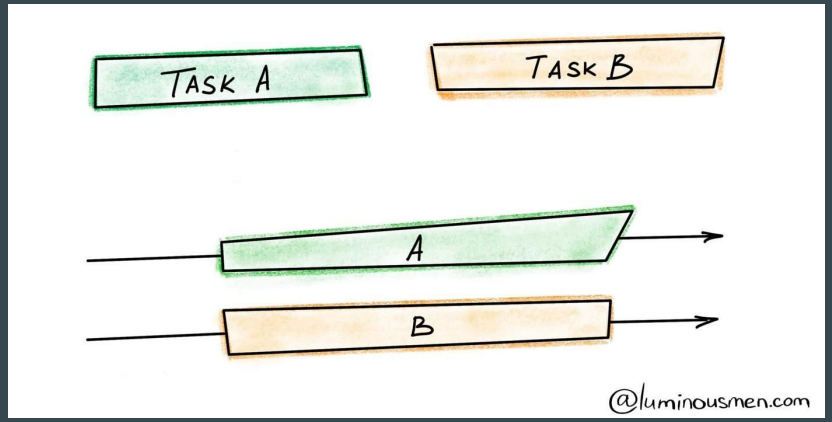
# Getting Started with Google GoLang

...

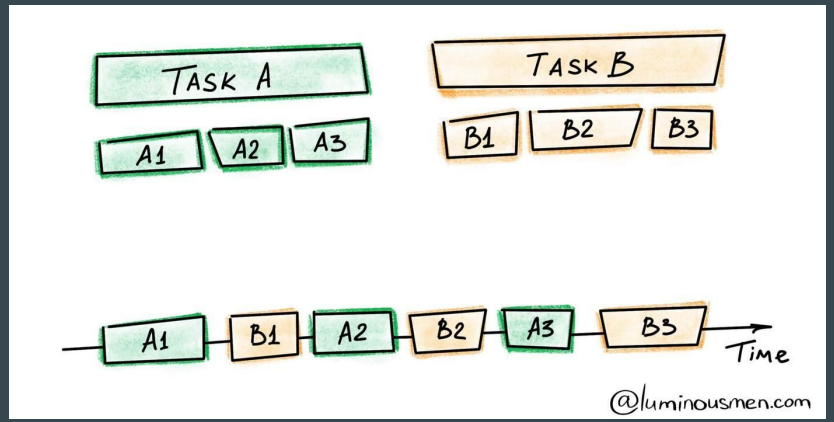
Week 3

# Parallelism and Concurrency

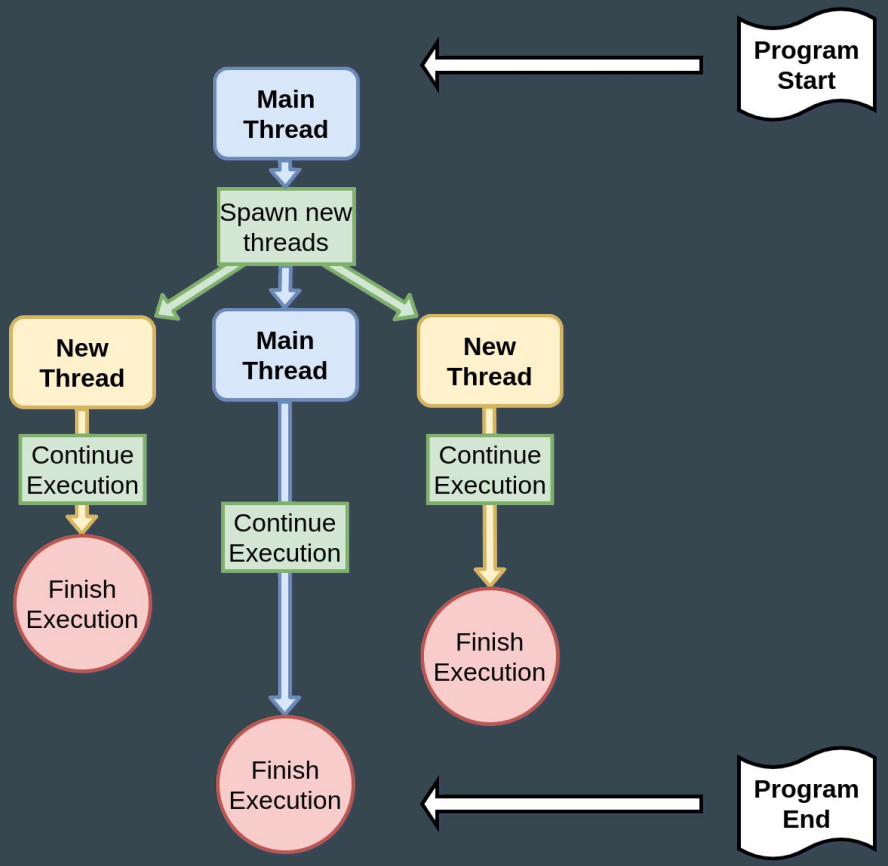



**Parallelism**

**Concurrency**



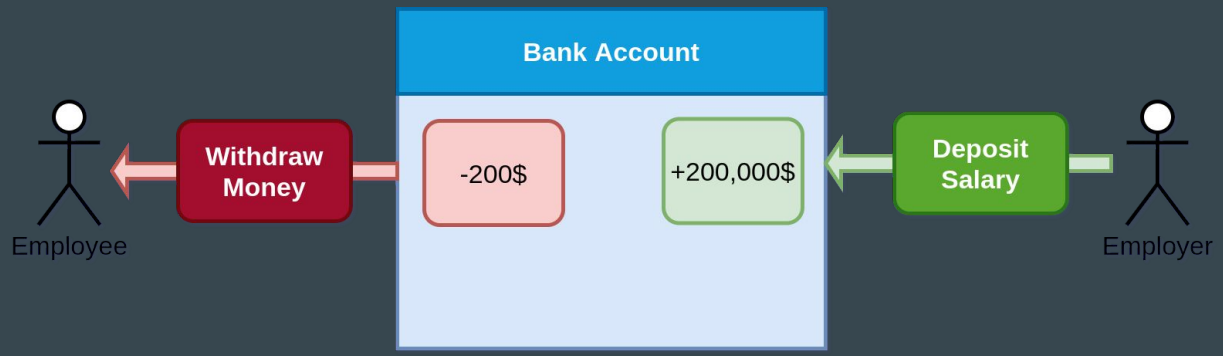
# Threads


Program Start
Main Thread
Spawn new threads
New Thread
Main Thread
New Thread
Continue Execution
Continue Execution
Continue Execution
Finish Execution
Finish Execution
Finish Execution
Program End

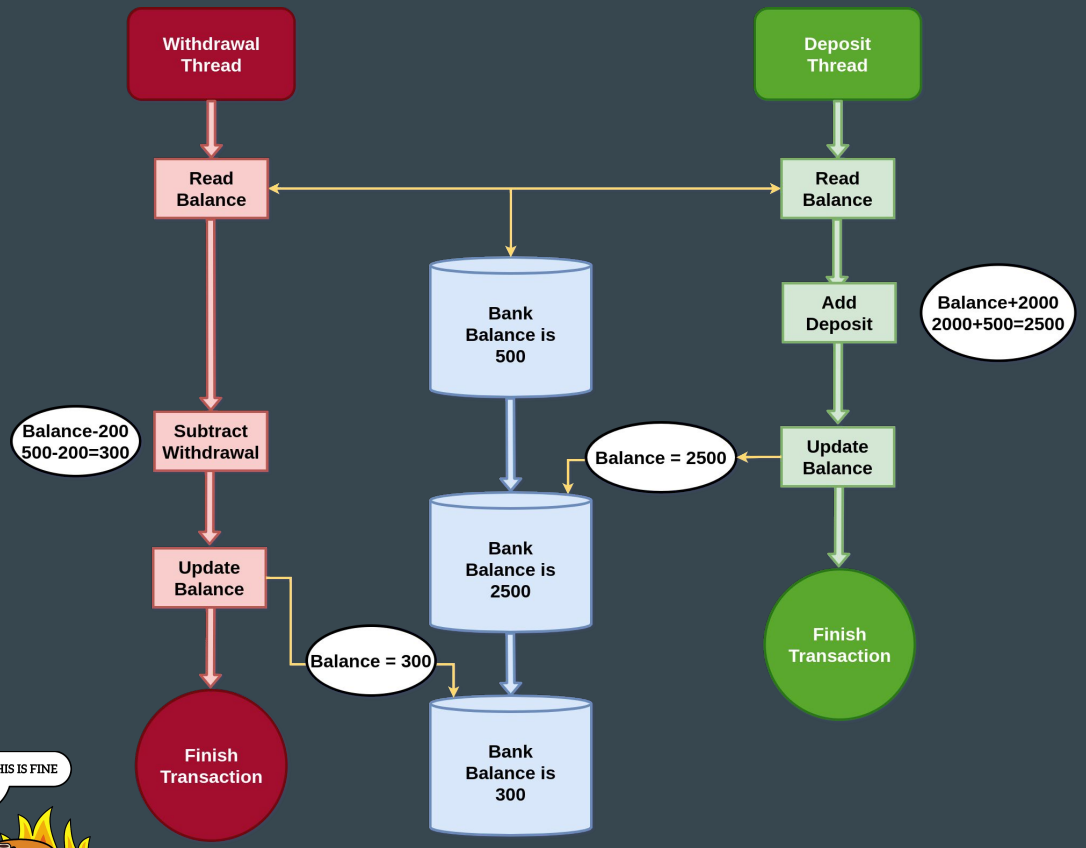
# Issues with concurrency & parallelism

Consider the following scenario:

- An employee is withdrawing money from his bank account.
- At the same time, his employer is depositing his salary.



# Race Conditions

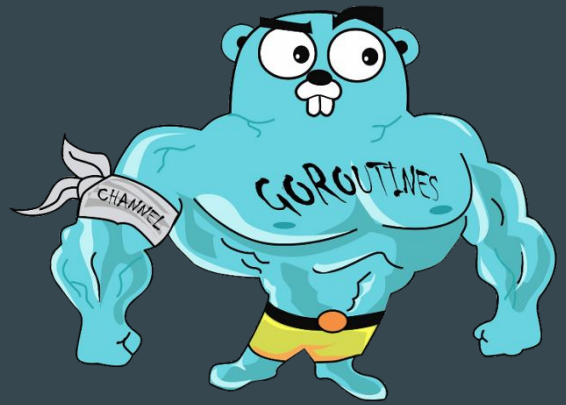



# GoRoutines

Routines are Go's equivalent of threads.

The default routine which runs is the main routine.

Easy to spawn and manage.



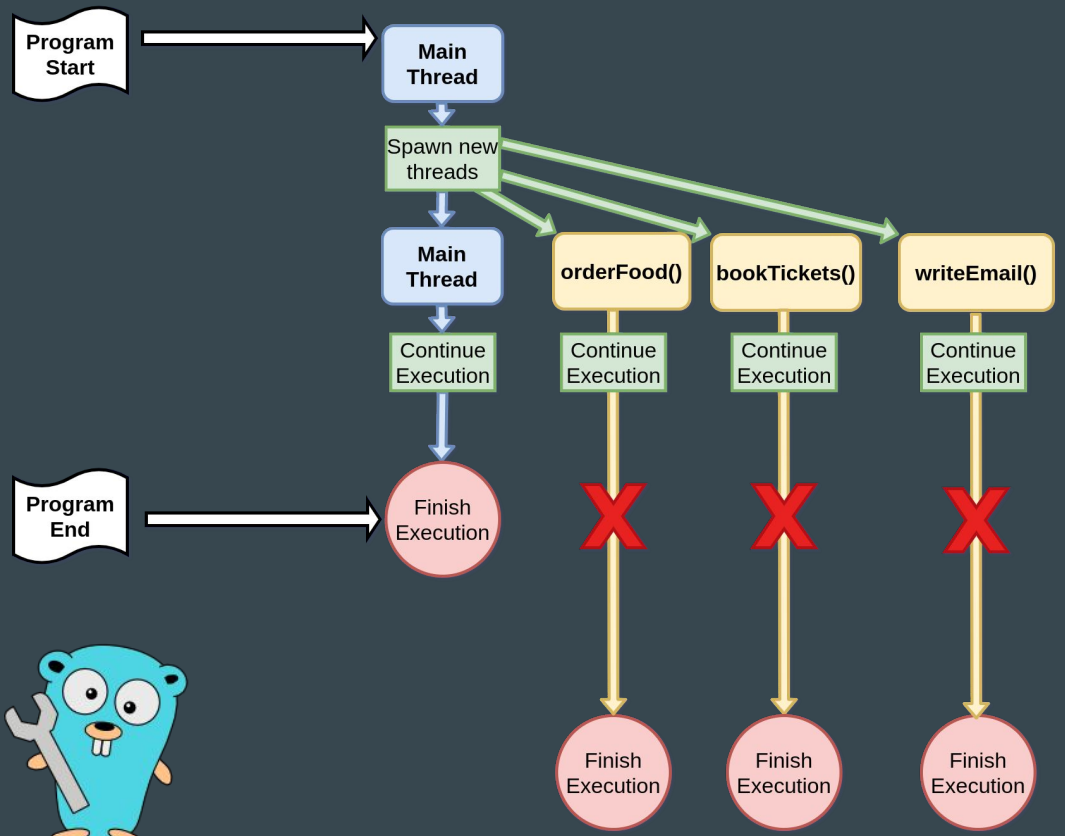
# How to create routines?

goroutines.go

```go
package main

import (
  "fmt"
)

func writeEmail() {
  fmt.Println("Finished writing an email!")
}

func orderFood() {
  fmt.Println("Finished ordering food!")
}

func bookTickets() {
  fmt.Println("Finished booking tickets!")
}

func main() {
  go writeEmail()
  go orderFood()
  go bookTickets()
  fmt.Println("Started routines!")
}
```

```
Week-3 ❯ go run goroutines.go
Started routines!
Week-3 ❯
```

# Routines: What went wrong?




# Ugly Fix

```go
func main() {
  go writeEmail()
  go orderFood()
  go bookTickets()
  fmt.Println("Started routines!")
  time.Sleep(10 * time.Second)
}
```

```
Week-3 ❯ go run goroutines.go
Started routines!
Finished booking tickets!
Finished writing an email!
Finished ordering food!
Week-3 ❯
```

# The right way to do it

```go
package main

import (
  "fmt"
  "sync"
)

func writeEmail(wg *sync.WaitGroup) {
  fmt.Println("Finished writing an email!")
  wg.Done()
}

func orderFood(wg *sync.WaitGroup) {
  fmt.Println("Finished ordering food!")
  wg.Done()
}

func bookTickets(wg *sync.WaitGroup) {
  fmt.Println("Finished booking tickets!")
  wg.Done()
}
```

```go
func main() {
  // Create WaitGroup variable
  var wg sync.WaitGroup

  // Initialise WaitGroup
  wg.Add(3)

  // Start routines
  go writeEmail(&wg)
  go orderFood(&wg)
  go bookTickets(&wg)

  fmt.Println("Started routines!")

  // Wait for routines to finish
  wg.Wait()
}
```

# Any routine can finish first!

```
Week-3 ❯ go run goroutines-fixed.go
Started routines!
Finished booking tickets!
Finished writing an email!
Finished ordering food!
Week-3 ❯ go run goroutines-fixed.go
Started routines!
Finished booking tickets!
Finished ordering food!
Finished writing an email!
Week-3 ❯
```

# The critical section problem

In the bank situation we saw earlier, how do we ensure a shared resource or variable does not undergo such mishaps?

A simple solution is to use a mutex.

Mutexes allow only one routine to enter the critical section at a time.

# Modifying shared resources

race.go

```go
import (
  "fmt"
  "sync"
)

var balance = 0

// Simple function which increments
// balance by 1
func deposit(wg *sync.WaitGroup) {
  // critical section start
  balance = balance + 1
  // critical section end
  wg.Done()
}

func main() {

  var w sync.WaitGroup

  // Spawn 1000 go routines
  for i := 0; i < 1000; i++ {
    w.Add(1)
    go deposit(&w)
  }

  w.Wait()
  fmt.Println("Bank balance is", balance)
}
```

```
Week-3 ❯ go run race.go
Bank balance is 945
Week-3 ❯ go run race.go
Bank balance is 896
Week-3 ❯ go run race.go
Bank balance is 902
Week-3 ❯ go run race.go
Bank balance is 944
Week-3 ❯
```

# The right way to do it

race-fixed.go

```go
package main

import (
  "fmt"
  "sync"
)

// Shared resource
var balance = 0

// Simple function which increments
// balance by 1
func deposit(wg *sync.WaitGroup, m *sync.Mutex) {
  // critical section start
  m.Lock()

  balance = balance + 1

  m.Unlock()
  // critical section end

  wg.Done()
}
```

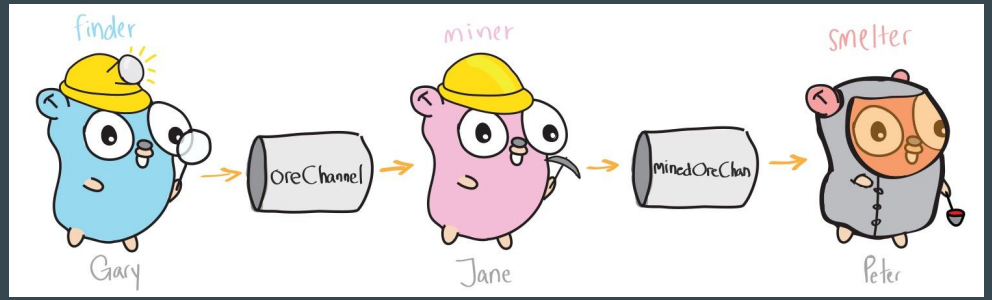
```go
func main() {

  // Mutex to be used
  var m sync.Mutex
  var w sync.WaitGroup

  // Spawn 1000 go routines
  for i := 0; i < 1000; i++ {
    w.Add(1)
    go deposit(&w, &m)
  }

  w.Wait()
  fmt.Println("Bank balance is", balance)
}
```

```
Week-3 ❯ go run race-fixed.go
Bank balance is 1000
Week-3 ❯ go run race-fixed.go
Bank balance is 1000
Week-3 ❯ go run race-fixed.go
Bank balance is 1000
Week-3 ❯
```

# Inter Routine Communication

How do routines send messages to each other?

They make use of **channels**.



# Creating & using channels

channels.go

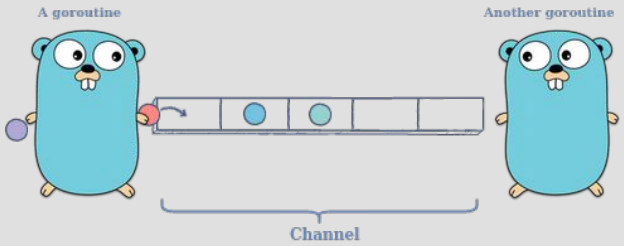
```go
package main

import "fmt"

func generate(ch chan int) {
  for i := 0; i < 5; i++ {
    fmt.Println("Sending", i)
    ch <- i
  }
  close(ch)
}

func main() {
  chan1 := make(chan int, 0)

  go generate(chan1)

  num := <-chan1
  fmt.Println("Got", num)

  for num := range chan1 {
    fmt.Println("Got", num)
  }
}
```

```
Week-3 ❯ go run channels.go
Sending 0
Sending 1
Got 0
Got 1
Sending 2
Sending 3
Got 2
Got 3
Sending 4
Got 4
Week-3 ❯
```

# Buffered channels

buffered-channels.go

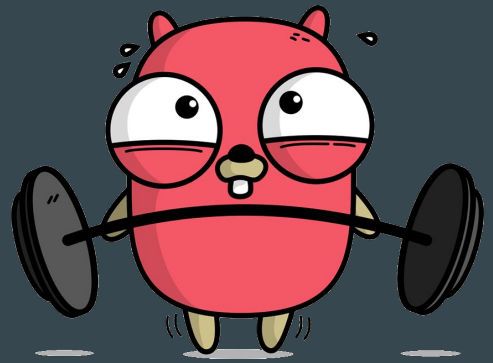
```go
package main

import "fmt"

func generate(ch chan int) {
    for i := 0; i < 5; i++ {
        fmt.Println("Sent", i)
        ch <- i
    }
    close(ch)
}

func main() {
    chan1 := make(chan int, 5)

    go generate(chan1)

    num := <-chan1
    fmt.Println("Got", num)

    for num := range chan1 {
        fmt.Println("Got", num)
    }
}

```

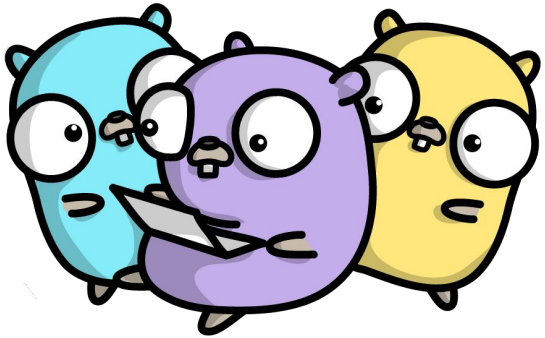



```
Week-3 ❯ go run buffered-channels.go
Sent 0
Sent 1
Sent 2
Sent 3
Sent 4
Got 0
Got 1
Got 2
Got 3
Got 4
Week-3 ❯
```

# Resources on Channels

- geeksforgeeks.org/channel-in-golang/
- sohamkamani.com/blog/2017/08/24/golang-channels-explained/





# Thank You!

Source Code and Slides available at:
github.com/Gituser143/PESU-IO-Go