NEWSTORE

Concurrency Patterns in Go

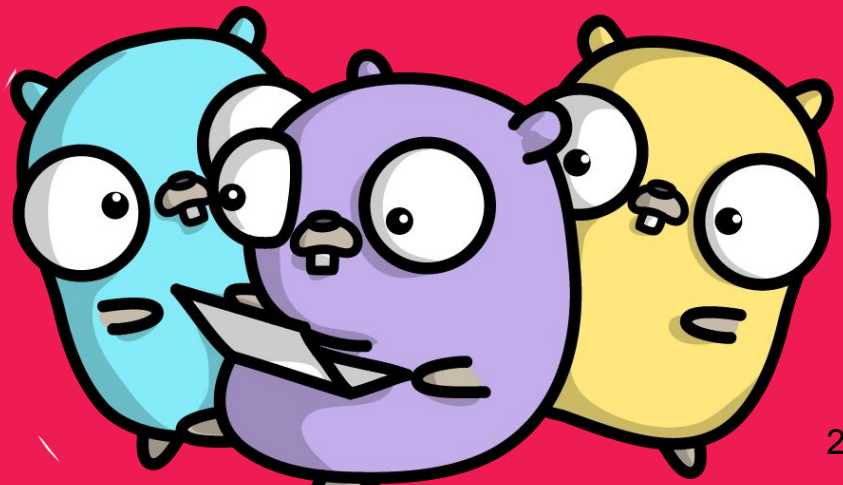
Serdar Tahir Kabaoglu
#team-customer-and-catalog

Codes & Presentation available @
`https://github.com/seredot/go-concurrency`

2/14/2020

Agenda

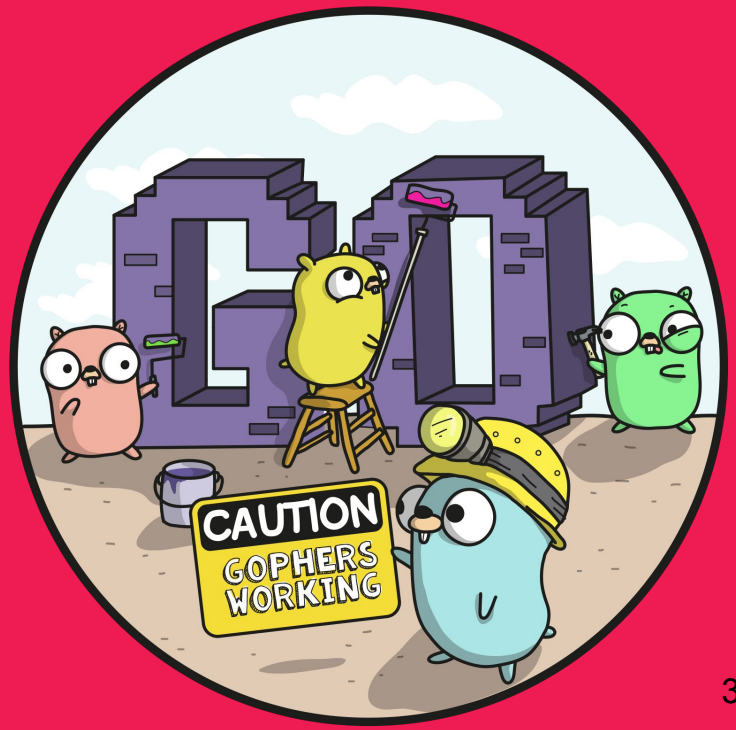
-> Concurrency vs Parallelism
-> CPU vs IO Bound Workloads
-> Goroutines
-> `WaitGroup`
<- Demo: WaitGroup
-> Channels
<- Demo: Throttle
-> sync Package
<- Demo: Worker pool
-> `select`
<- Demo: Timeout
?: Q&A



Concurrency vs Parallelism

Parallelism
Execution on different processing units at the same time.

Concurrency
Execution out of order.



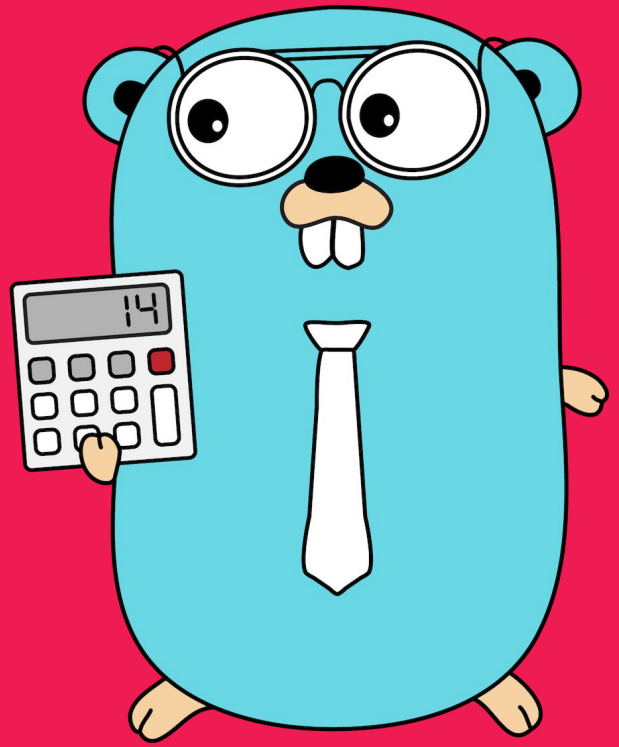
IO vs CPU Bound Workloads

IO Bound

Waiting for events, network operations, database queries, file operations, locks.

CPU Bound

Long calculations like: finding n-th prime, compression, encoding, encryption, etc.

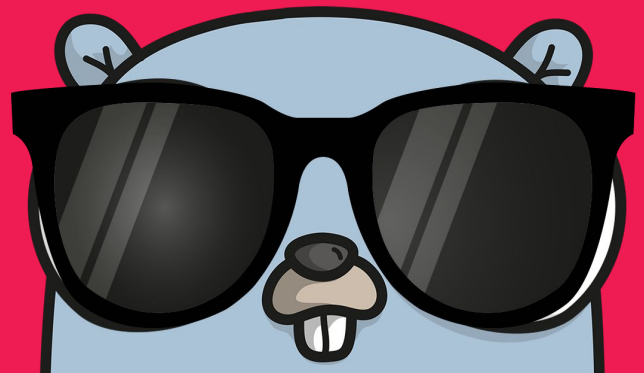


Goroutines

A Goroutine is a function that runs concurrently.

A Goroutine can be thought like a lightweight thread.

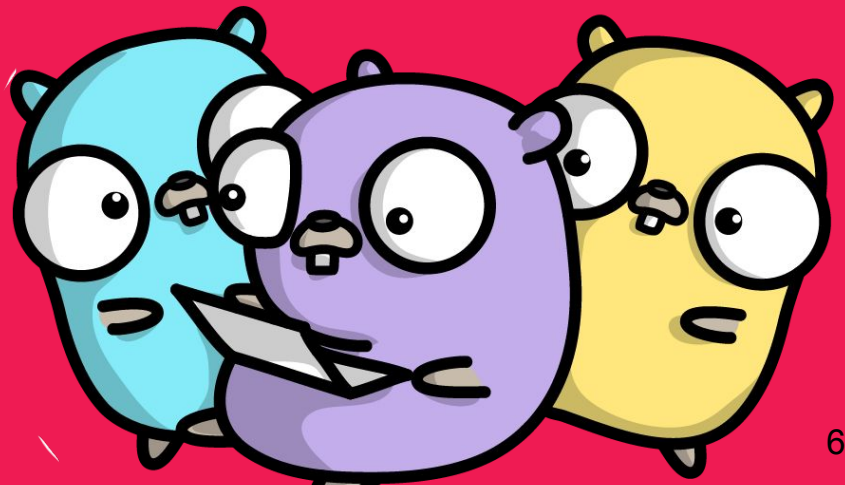
```
go functionForSomeTask()
```



WaitGroup

A WaitGroup waits for a collection of goroutines to finish.

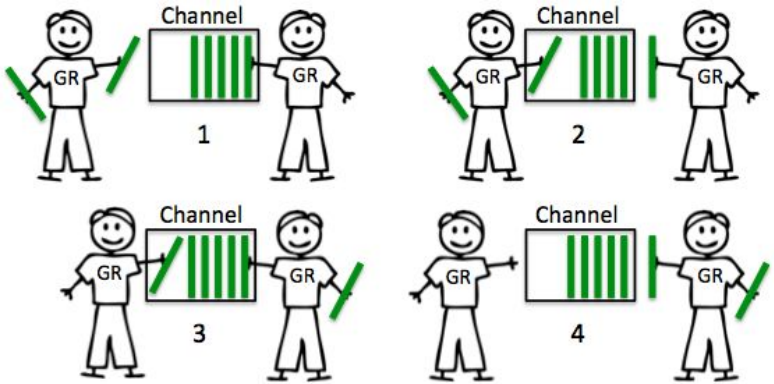
```
var wg sync.WaitGroup
wg.Add(1)
wg.Done()
wg.Wait()
```

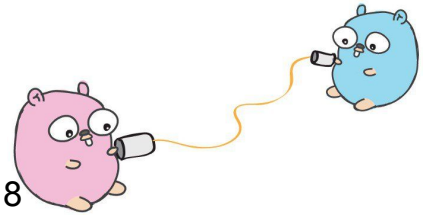


DEMO

`wait-group`

Channels





```go
// Create unbuffered channel.
c := make(chan string)

// Create buffered channel.
c := make(chan int, 10)

// Send to the channel. The value is copied.
c <- 3

// Receive from the channel.
i := <-c

// Pass a readable channel.
func foo(r <-chan string) {}

// Pass a writable channel.
func bar(w chan<- string) {}

// Read/write channel.
func foobar(rw chan string) {}

// Returns a readable channel.
func numbers() <-chan int {}

// Close the channel.
close(c)
```

DEMO

`throttle`

sync Package

WaitGroup

.Add, .Done, .Wait

Mutex

.Lock, .Unlock

RWMutex

.RLock, RUnlock, .Lock, .Unlock

Once

.Do

Map

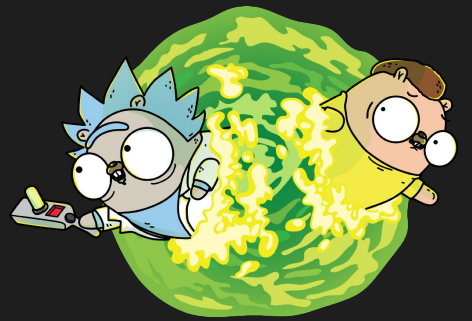
.Load, .Store, .Delete

Cond

.Wait, .Signal, .Broadcast

sync/atomic Package

```
atomic.Addint64()
atomic.LoadInt64()
atomic.StoreInt64()
atomic.SwapInt64()
atomic.CompareAddSwapint64()
atomic.Value
```



DEMO

`worker-pool`

select

Tries to read from multiple channels or write to multiple channels.

Blocks until one of the cases is available.

Does not block if there is a default case.

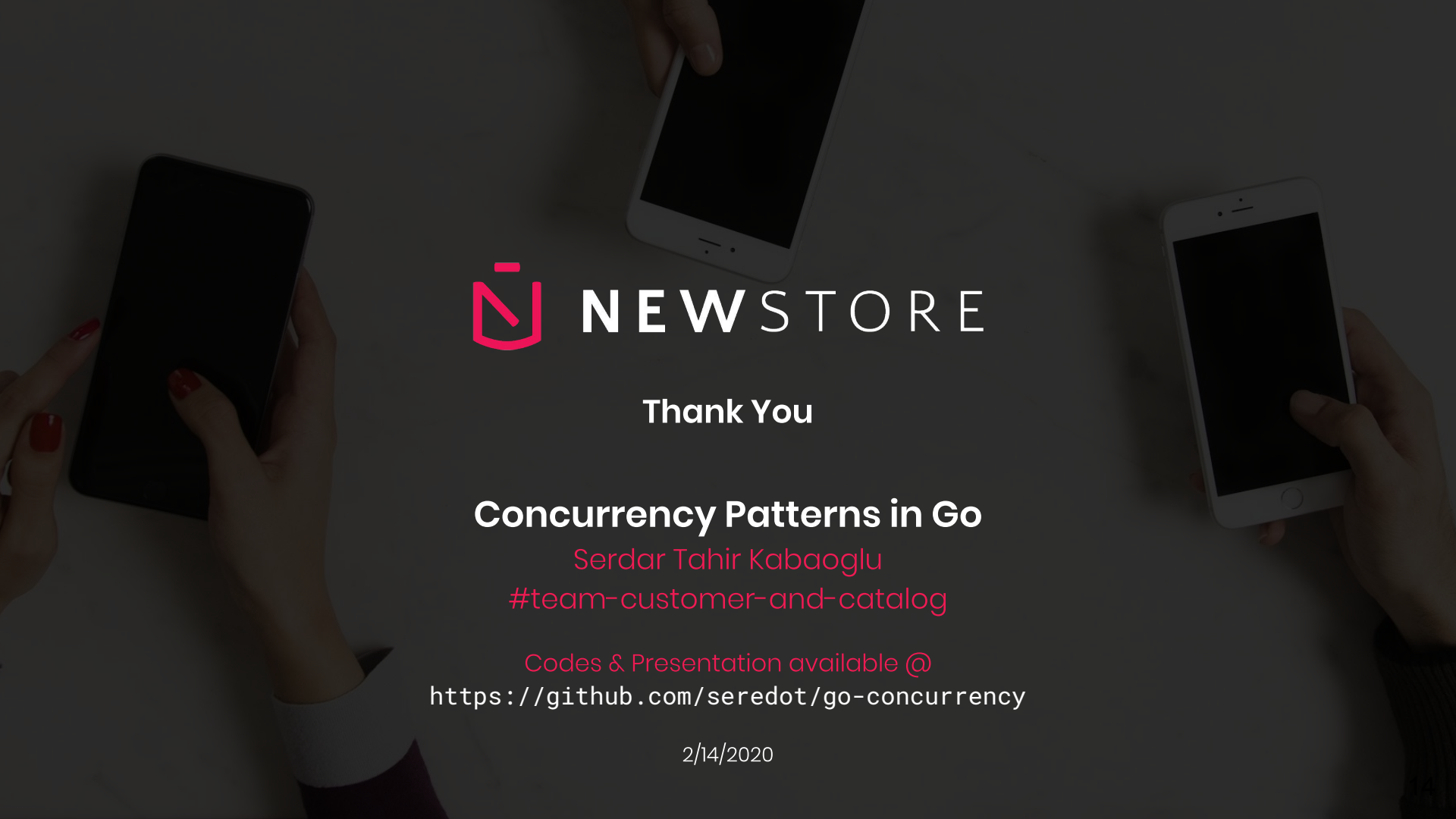
If multiple are channels are available, chooses **randomly**.

```
// Blocks until receive.
select {
case i := <-c1:
  fmt.Printf("Operation 1 returned %d\n", i)
case j := <-c2:
  fmt.Printf("Operation 2 returned %d\n", j)
}

// Does not block.
select {
case m := <-mails:
  fmt.Println("You've got mail!", m.subject)
default:
  fmt.Println("No mail :(")
}
```

DEMO

`timeout`



Thank You

Concurrency Patterns in Go

Serdar Tahir Kabaoglu
#team-customer-and-catalog

Codes & Presentation available @
`https://github.com/seredot/go-concurrency`

2/14/2020