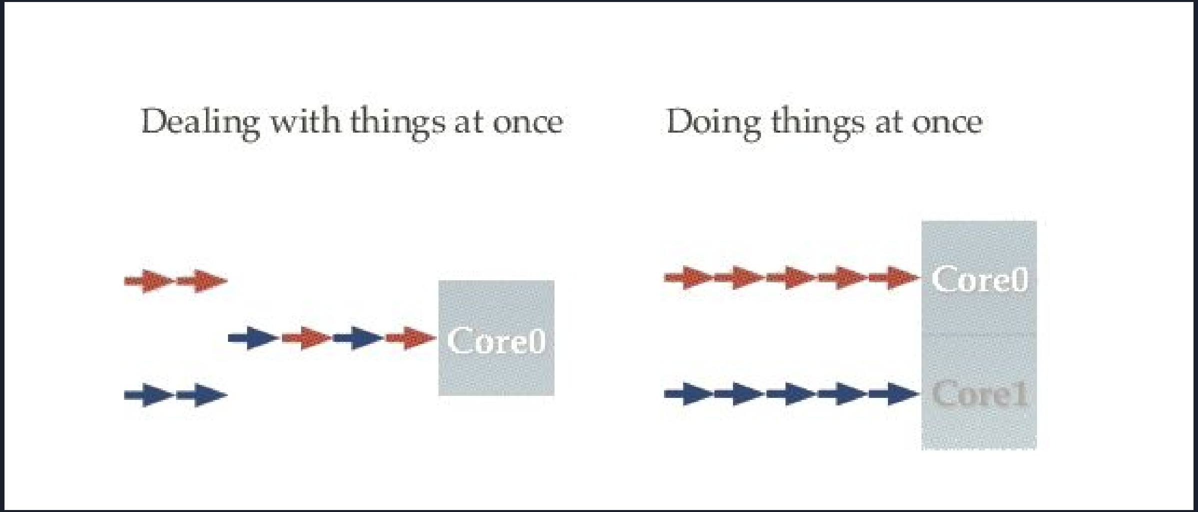
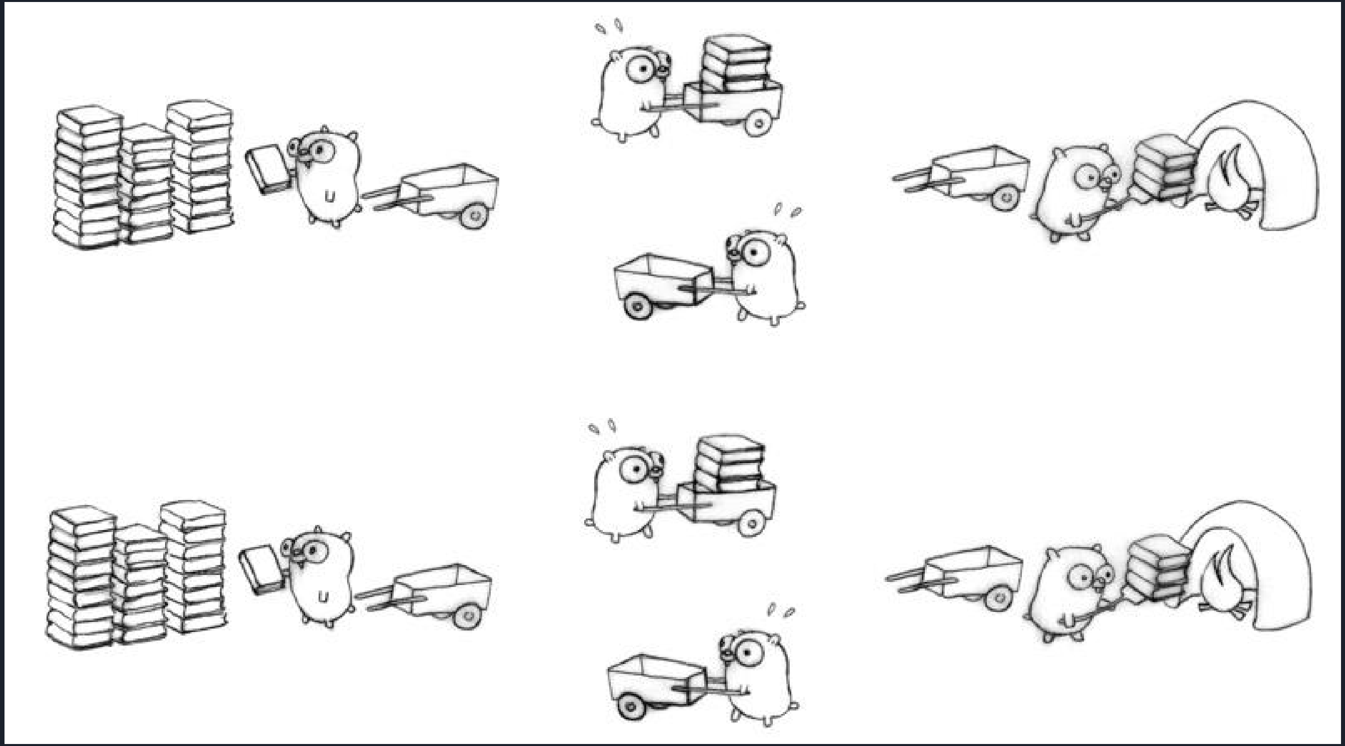
Gophers, Concurrency and Parallelism

Gwendal Leclerc
@skillo1989
Go developer at OVH

Concurrency vs Parallelism



Concurrency with Parallelism



Channels

- Can be viewed as a FIFO (first in, first out) message queue
- Transfer **ownership** of data between goroutines
- Composite types like array, slice and map
- Have **nil** as zero value
- Can be directional
  - chan T <=> Bidirectional channel
  - chan<- T <=> Send-only channel
    - Compiler doesn't allow receiving values
  - <-chan T <=> Receive-only channel
    - Compiler doesn't allow sending values
- Can be **buffered**

Channels rules

Actions / Channels	Nil channel	Closed channel	Active channel
Close	panic	panic	close
Send to	block forever	panic	send or block
Receive from	block forever	never block (sending default value)	receive or block

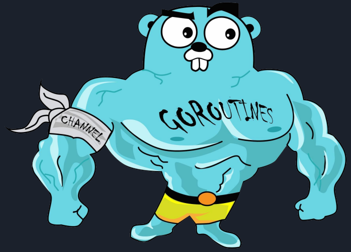
Why those rules ?

To drive the way you code your programs

- Only senders can know when they finished sending data
- It’s only necessary to close a channel if the receiver is looking for a close (even if you’re looking for close most of the time)

It's OK to leave a Go channel open forever and never close it, if it’s not used and referenced anymore. It will be garbage collected.

But beware of memory leaks.



For-Range on Channels

```go
for v = range aChannel {
    // use v
}
```

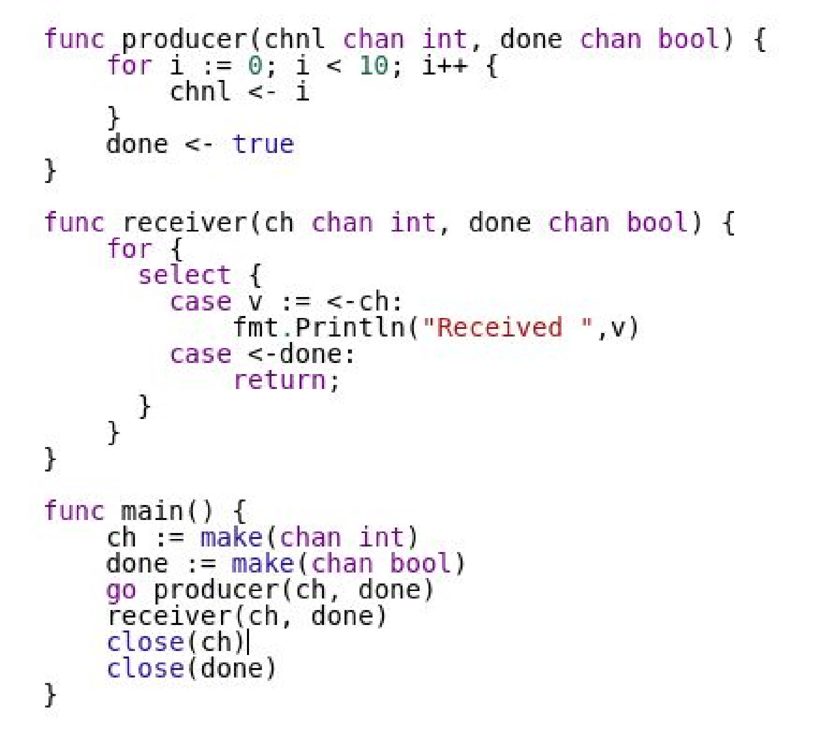
is equivalent to

```go
for {
    v, ok = <-aChannel
    if !ok {
        // v take the zero value of the channel’s type
        break
    }
    // use v
}
```

Select over Channels

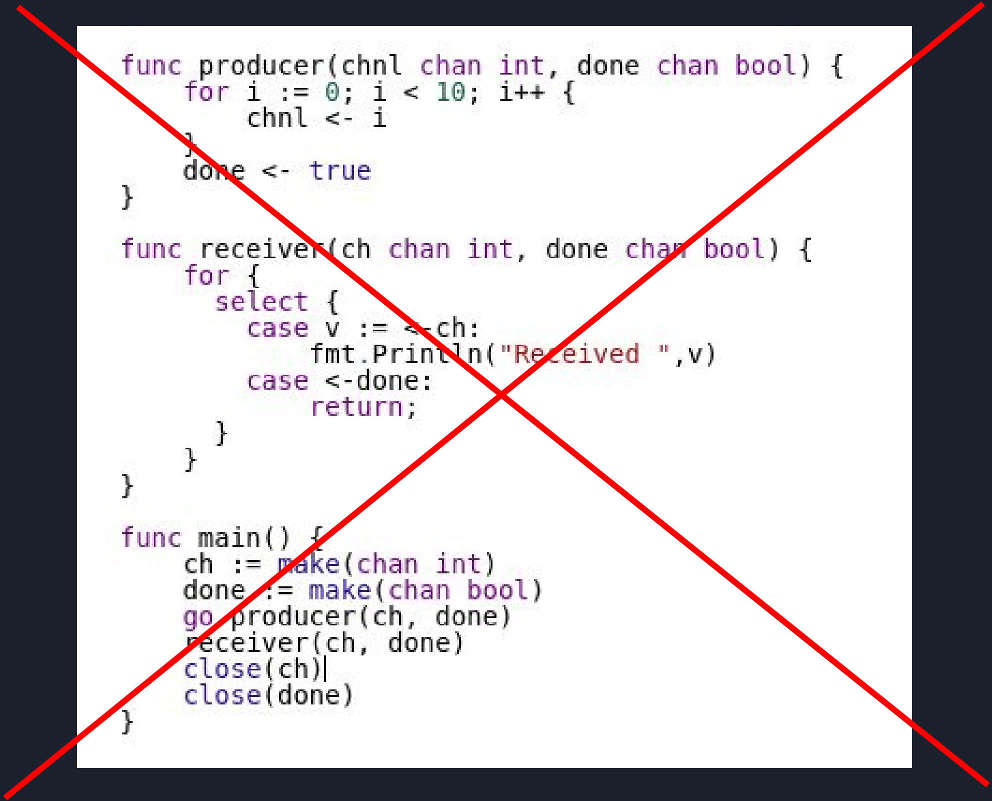
```
messages := make(chan string)
signals := make(chan bool)

select {
 case msg := <-messages:
     fmt.Println("received message", msg)
 case sig := <-signals:
     fmt.Println("received signal", sig)
 default:
     fmt.Println("other cases are blocked (waiting)")
}
```

Bad Example

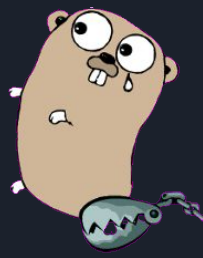


```go
func producer(chnl chan int, done chan bool) {
    for i := 0; i < 10; i++ {
        chnl <- i
    }
    done <- true
}

func receiver(ch chan int, done chan bool) {
    for {
      select {
        case v := <-ch:
            fmt.Println("Received ",v)
        case <-done:
            return;
      }
    }
}

func main() {
    ch := make(chan int)
    done := make(chan bool)
    go producer(ch, done)
    receiver(ch, done)
    close(ch)
    close(done)
}
```

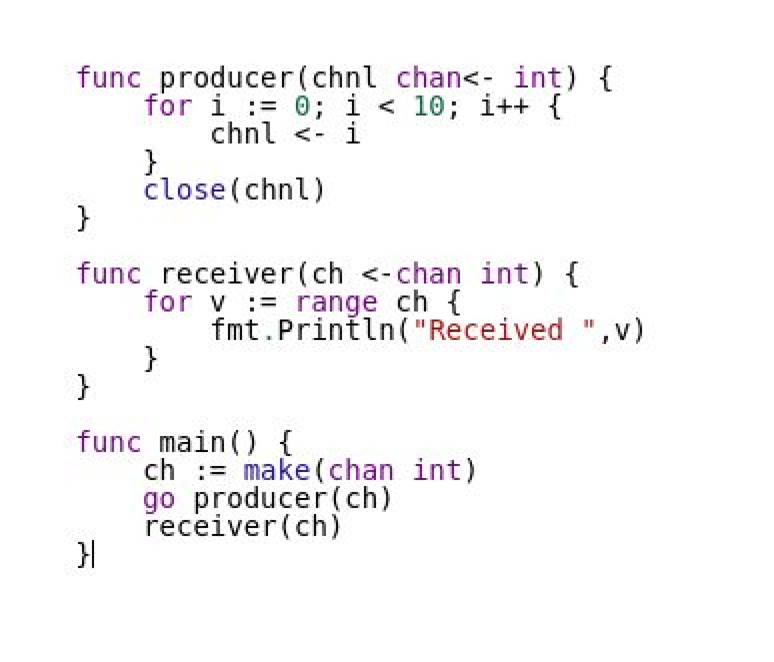
Bad Example



```go
func producer(chnl chan int, done chan bool) {
	for i := 0; i < 10; i++ {
		chnl <- i
	}
	done <- true
}

func receiver(ch chan int, done chan bool) {
	for {
	  select {
		case v := <-ch:
			fmt.Println("Received ",v)
		case <-done:
			return;
	  }
	}
}

func main() {
	ch := make(chan int)
	done := make(chan bool)
	go producer(ch, done)
	receiver(ch, done)
	close(ch)
	close(done)
}
```

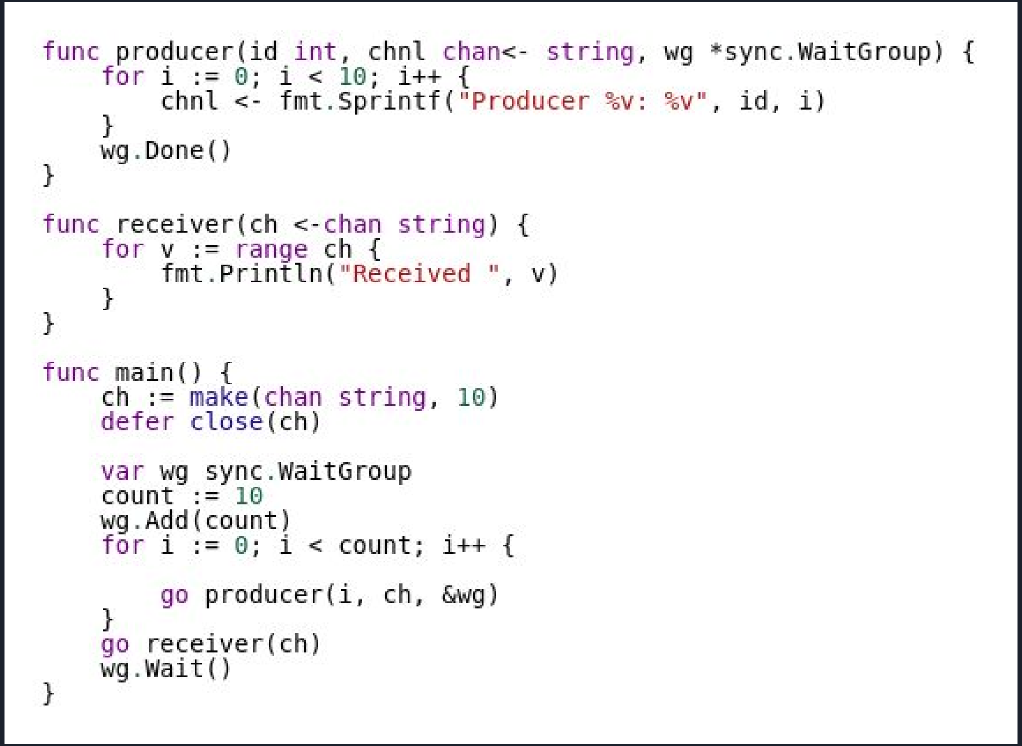


Correct way



```go
func producer(chnl chan<- int) {
    for i := 0; i < 10; i++ {
        chnl <- i
    }
    close(chnl)
}

func receiver(ch <-chan int) {
    for v := range ch {
        fmt.Println("Received ",v)
    }
}

func main() {
    ch := make(chan int)
    go producer(ch)
    receiver(ch)
}
```

Deal with multiple senders (sync.WaitGroup)



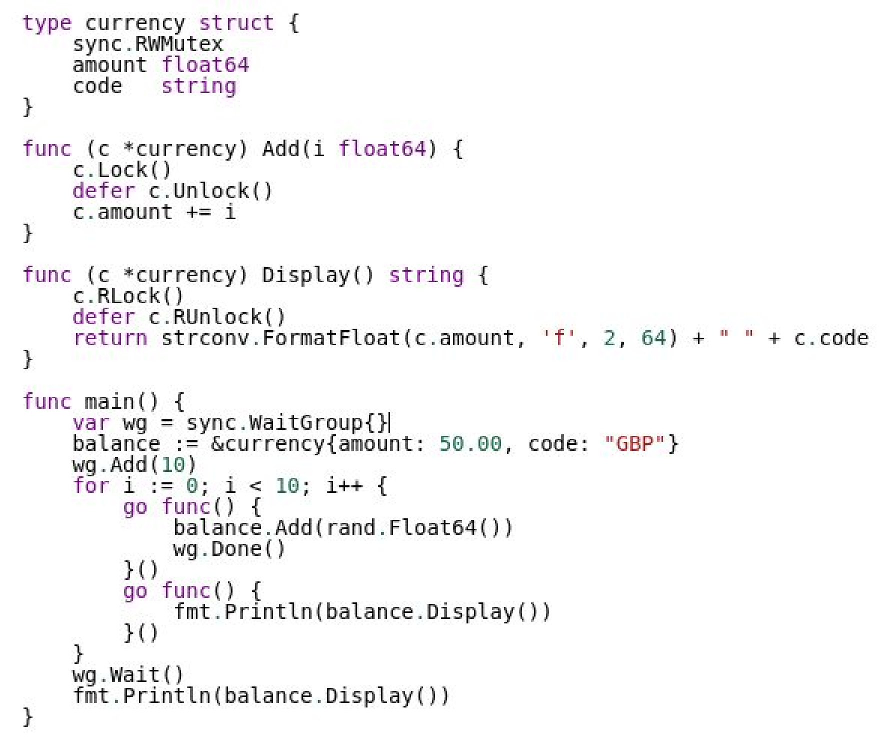
```go
func producer(id int, chnl chan<- string, wg *sync.WaitGroup) {
    for i := 0; i < 10; i++ {
        chnl <- fmt.Sprintf("Producer %v: %v", id, i)
    }
    wg.Done()
}

func receiver(ch <-chan string) {
    for v := range ch {
        fmt.Println("Received ", v)
    }
}

func main() {
    ch := make(chan string, 10)
    defer close(ch)

    var wg sync.WaitGroup
    count := 10
    wg.Add(count)
    for i := 0; i < count; i++ {

        go producer(i, ch, &wg)
    }
    go receiver(ch)
    wg.Wait()
}
```

Mutexes

- Use mutexes, it's **not bad**
- Mutexes will synchronize access to data
- Mutexes are really cheap in the non-blocking case
- ***"Threading isn't hard — locking is hard."***

Channels	Mutexes
● passing ownership of data ● distributing units of work ● communicate async results	● Concurrent access to data (cache / state)

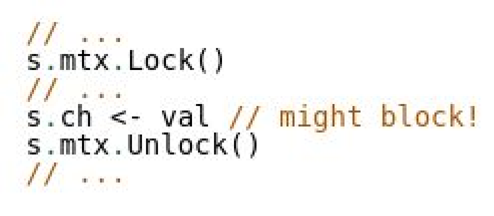
Warp mutexes into structures



```go
type currency struct {
    sync.RWMutex
    amount float64
    code   string
}

func (c *currency) Add(i float64) {
    c.Lock()
    defer c.Unlock()
    c.amount += i
}

func (c *currency) Display() string {
    c.RLock()
    defer c.RUnlock()
    return strconv.FormatFloat(c.amount, 'f', 2, 64) + " " + c.code
}

func main() {
    var wg = sync.WaitGroup{}
    balance := &currency{amount: 50.00, code: "GBP"}
    wg.Add(10)
    for i := 0; i < 10; i++ {
        go func() {
            balance.Add(rand.Float64())
            wg.Done()
        }()
        go func() {
            fmt.Println(balance.Display())
        }()
    }
    wg.Wait()
    fmt.Println(balance.Display())
}
```

Mutexes rules

- Don’t block inside a lock
- Be careful when using it with channels (due to channels rules)



```
// ...
s.mtx.Lock()
// ...
s.ch <- val // might block!
s.mtx.Unlock()
// ...
```

- Use race detector https://golang.org/doc/articles/race_detector.html

Warnings

- Keep lisibility and maintainability in mind
- Concurrency is not simple even with channels
- Using channel can break performance in some use cases
  - https://youtu.be/ySy3sR1LFCQ
- Avoid concurrency in your API
  - https://talks.golang.org/2013/bestpractices.slide#25
- Avoid goroutine leaks
  - https://medium.com/golangspec/goroutine-leak-400063aef468

To go further

- How to wait for all goroutines to finish
- JustForFunc Youtube
- Channels are bad and you should feel bad
- Dancing with go mutexes
- Tapirgames blog golang channel

Questions ?

Thanks