Concurrent Programming Languages

Channel-based Concurrency Module

Lecture 1: Introduction to Go

12 October 2021

MIEI - Integrated Masters in Comp. Science and Informatics
Specialization Block

Bernardo Toninho
(with António Ravara and Carla Ferreira)

NOVALINCS

Admin Stuff — Planning

4 Lectures:

1. Introduction to the Go programming language
   1.1. Basic language features & program organization
   1.2. Channel-based concurrency in Go
2. Coordination using Channels: Patterns and Perils
3. (Advanced) Channel-based Programming Patterns
4. Selected Research Topic (TBD)

Admin Stuff — Planning

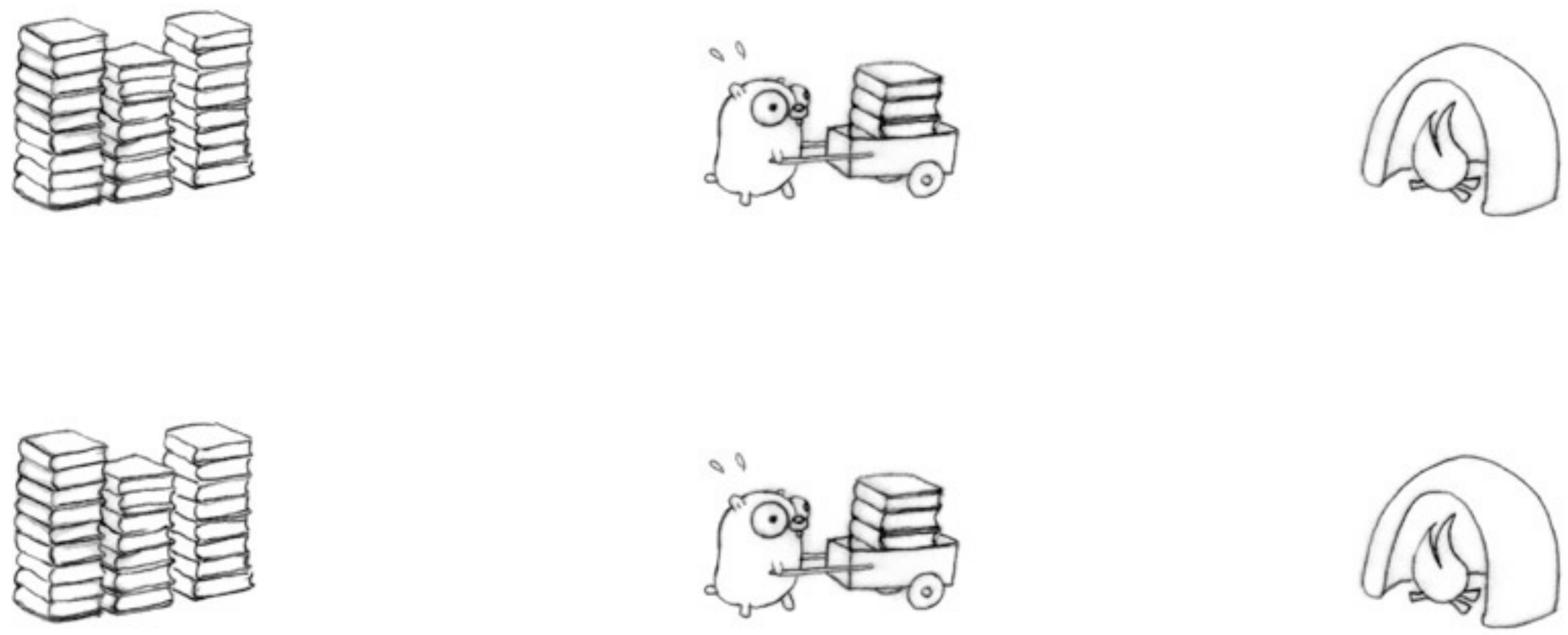
4 Labs:

1. Go introduction
2. Mini-Project
3. Project
4. Project

Parallelism vs Concurrency

Parallelism: Programming as the simultaneous execution of (possibly related) computations.

Concurrency: Programming as the composition of independently executing processes.

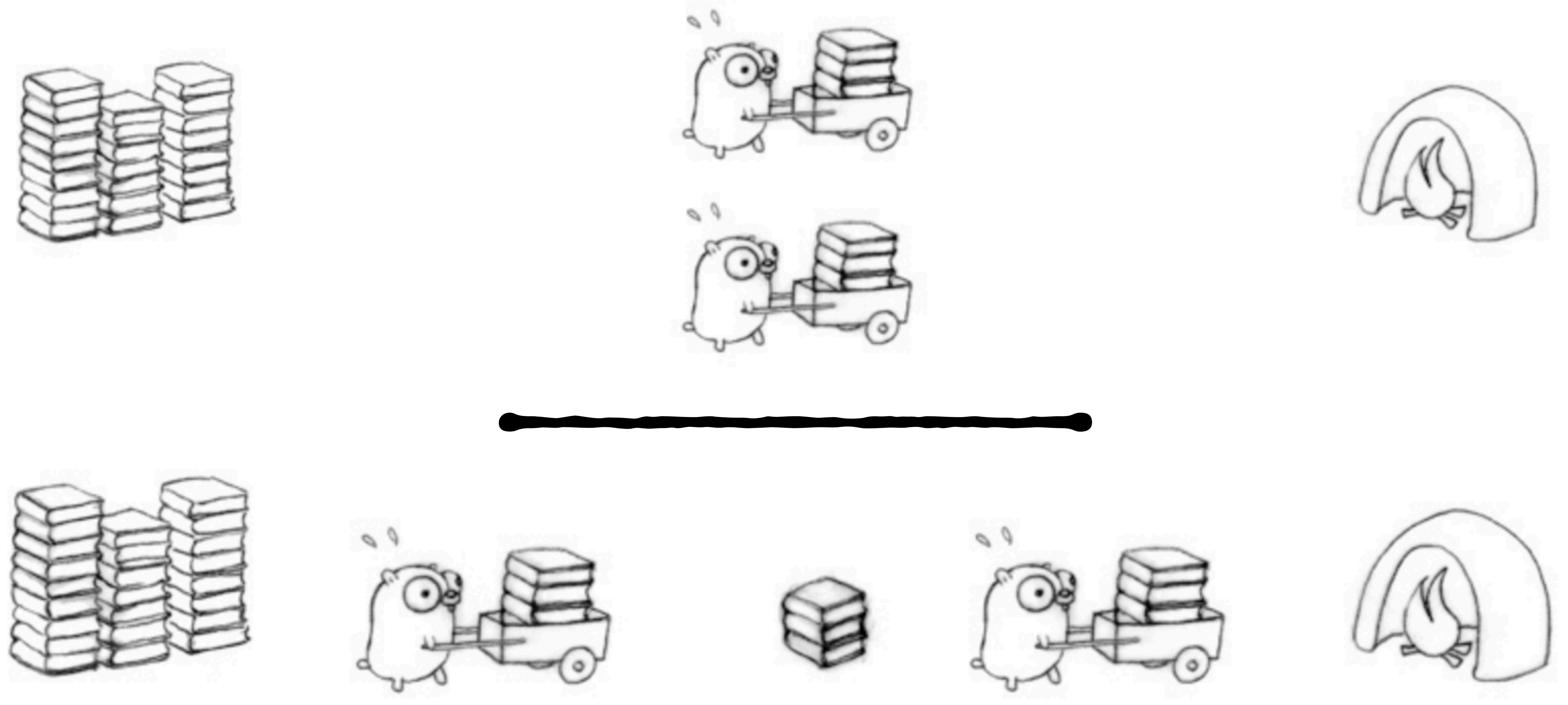


picture from https://talks.golang.org/

Parallelism vs Concurrency

Parallelism: Programming as the simultaneous execution of (possibly related) computations.

Concurrency: Programming as the composition of independently executing processes.



Concurrency vs Parallelism

- Concurrency is not parallelism, but parallelism is enabled by concurrency!
- Programs can be concurrent and have 0 parallelism.
- Well-written concurrency may run better on a multiprocessor.

Concurrency and Independence

> - Concurrency is a way to **structure** work into independent pieces …
>
> … but then you have to coordinate those pieces
>
> *Andrew Gerrand (Golang)*

- "*Independent*" here refers to a way of *thinking* about problems, and *structuring* their solutions.

- Concurrent processes may indeed *interfere/interact*

The Go Language

- Designed by Pike, Griesemer, Thompson and others in late 2007 at Google.

- A simple but powerful language
- C without (most of) the scary parts
- **Channel-based concurrency** primitives built-in, closures, garbage collection, proper strings, …
- Motivated by software problems at “Google scale” (good performance, fast builds, easy to understand)

The Go Language

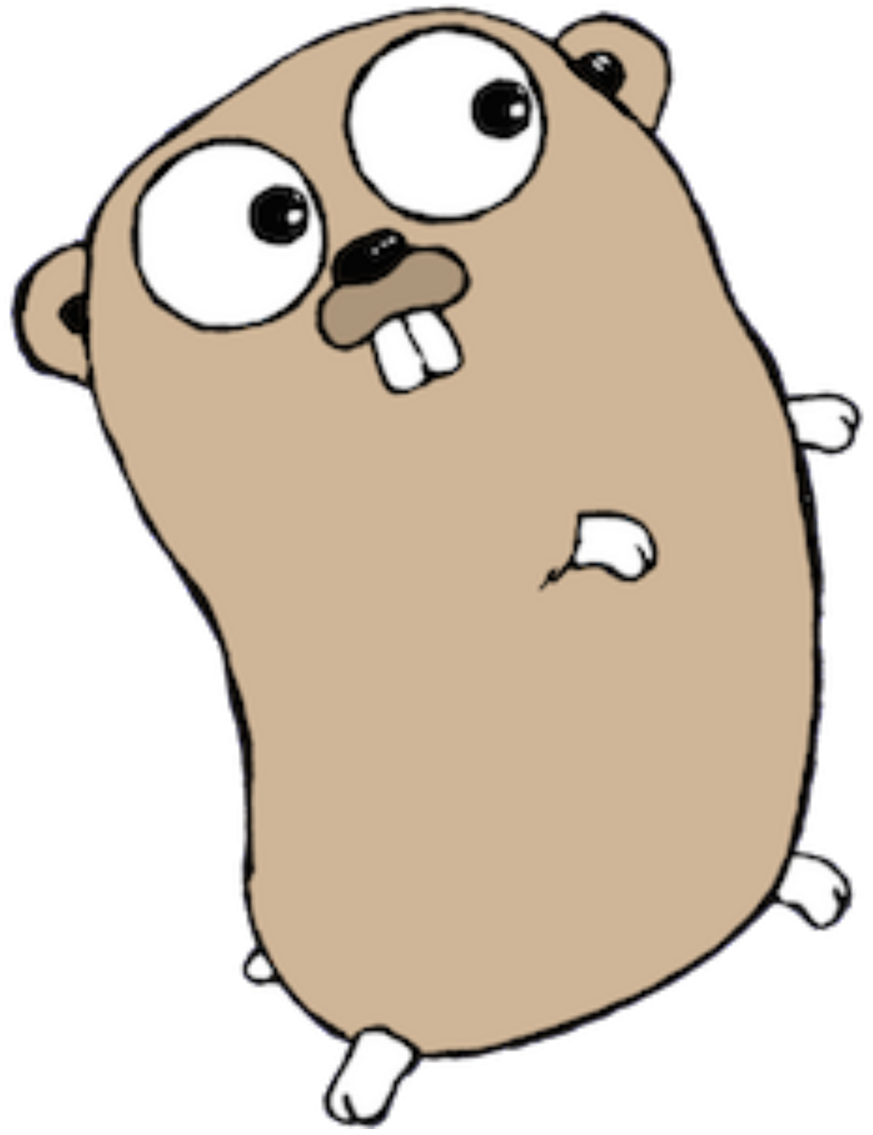
- **Note:** This will not be a complete introduction / tutorial on Go. Just enough to get us going.

- Many good resources are available:

    - https://tour.golang.org/

    - https://golang.org/doc/code.html

    - https://golang.org/doc/effective_go.html



Design Philosophy

- Go is a strongly-typed **imperative** language:
  - Programs are collections of `structs` and `func`tions that manipulate them.
  - Pointers, but no pointer arithmetic (for safety).
  - All functions copy their arguments (more later).
  - Channel-based concurrency and green threads built-in and “easy to use” (based on CSP, CCS).

Don’t communicate by sharing memory,
share memory by communicating

Variables and Assignment

```
var a,b int          //creates two variables of type int
                     //initially 0

b = 10               //assigns 10 to b
                     //after it has been created

b := 10              //creates and initializes b (as int)
                     //b must be a new name
```

- **All** types have a so-called *zero value* (recursive for composite types).
- **All** variables and declared imports **must** be used (compile-time error)
- Types for variables can often be omitted (lightweight type inference).

Hello World :)

```go
package main

import (                          // imports packages
  "fmt"                           // fmt for printing
)

func main() {
  fmt.Println("Hello, world!")
}
```

- All go source files must define some package (doesn’t need to match file name or folder).

Hello World :)

```go
package main

import (                                      // imports packages
    "fmt"                                     // fmt for printing
)

func main() {
    fmt.Println("Hello, world!")
}
```

- Unused imports are flagged as compiler errors.

Hello World :)

```go
package main

import (                                // imports packages
  "fmt"                                 // fmt for printing
)

func main() {
  fmt.Println("Hello, world!")
}
```

- Program entry point. Function without args, no return value.

Loops

```
for initialisation ; condition; post {
        // zero or more statements
}
```

```
sum := 0
for i := 0; i < 10; i++ {
        sum += i
}
```

```
sum := 1
for ; sum < 10; {
    sum += sum                                  //doubles sum until it is 16
}
```

```
for ; ; {
    …                                          //forever
}
```

Quiz

```go
func incr(x int) int {
  x = x+1
  return x
}
```

```go
func main() {
  a := 22
  incr(a)
  fmt.Println(a)
}
```

What number does this program print?

Quiz

```
func incr(x int) int {
  x = x+1
  return x
}
```

```
func main() {
  a := 22
  incr(a)
  fmt.Println(a)
}
```

What number does this program print? 22!

In go, all functions copy the value of their arguments…

Quiz

```
func incr(x *int) {
  *x = *x+1
}
```

```
func main() {
  a := 22
  incr(&a)
  fmt.Println(a)
}
```

By passing pointers we can modify a, “as expected”.

Arrays and Slices

```
var a [2]string              // creates a as array of 2 strings
a[0] = "Hello"
a[1] = "World"

primes := [6]int{2, 3, 5, 7, 11, 13} //creates & initializes
```

- `[n]T` is the type of an array of size n with elements of type `T`

Arrays and Slices

- `[n]T` is the type of an array of size n with elements of type `T`

```
func printer(arr [6]int) {
  for i := 0; i < len(arr); i++ {
    fmt.Println(arr[i])
  }
}
…
primes := [6]int{2, 3, 5, 7, 11, 13}
morePrimes := [7]int{2, 3, 5, 7, 11, 13, 17}
printer(primes)
printer(morePrimes)
```

Arrays and Slices

- `[n]T` is the type of an array of size n with elements of type `T`

```
func printer(arr [6]int) {
   for i := 0; i < len(arr); i++ {
      fmt.Println(arr[i])
   }
}
…
primes := [6]int{2, 3, 5, 7, 11, 13}
morePrimes := [7]int{2, 3, 5, 7, 11, 13, 17}
printer(primes)              // OK
printer(morePrimes)          // Type error
```

Arrays and Slices

- `[n]T` is the type of an array of size n with elements of type `T`

```
func printer(arr [6]int) {
   for i := 0; i < len(arr); i++ {
      fmt.Println(arr[i])
   }
}
…
primes := [6]int{2, 3, 5, 7, 11, 13}
morePrimes := [7]int{2, 3, 5, 7, 11, 13, 17}
printer(primes)              // OK
printer(morePrimes)          // Type error
```

Note: Arrays are **values**! A lot of copying above…

Arrays and Slices

- Arrays in Go are quite rigid. Not used often.
- *Slices* build on arrays to provide flexibility.

```go
primes := [6]int{2, 3, 5, 7, 11, 13}     //array literal
otherPrimes := []int{19, 23, 29}         //slice literal
```

```go
somePrimes := primes[0:3]    //slicing an array

nums := make([]int, 5)       //allocate + return slice
```

Arrays and Slices

- Arrays in Go are quite rigid. Not used often.
- *Slices* build on arrays to provide flexibility.

```
primes := [6]int{2, 3, 5, 7, 11, 13}     //array literal
otherPrimes := []int{19, 23, 29}         //slice literal
```

```
somePrimes := primes[0:3]   //slicing an array

nums := make([]int, 5)      //allocate + return slice
```

Arrays and Slices

- Slices have a length and a capacity:

```
primes := [6]int{2, 3, 5, 7, 11, 13}

somePrimes := primes[0:3]   //slicing an array

fmt.Println(len(somePrimes)) // 3
fmt.Println(cap(somePrimes)) // 6

fmt.Println(somePrimes)       // [2 3 5]
fmt.Println(somePrimes[3:cap(somePrimes)]) // [7 11 13]
```

Arrays and Slices

- Slices can be copied and appended:

```
primes := []int{2, 3, 5, 7, 11, 13}
p := []int{19, 23, 29}

s := make([]int, len(primes), cap(primes)*2 )
copy(s, primes)
primes = s                     //doubled capacity

a := append(primes,p[0],p[1],p[2]) //[2 3 5 7 11 13 19 23 29]
b := append(primes,p…)             //[2 3 5 7 11 13 19 23 29]
```

Arrays and Slices

- Slices can be copied and appended:

```
primes := []int{2, 3, 5, 7, 11, 13}
p := []int{19, 23, 29}

s := make([]int, len(primes), cap(primes)*2 )
copy(s, primes)
primes = s                   //doubled capacity

a := append(primes,p[0],p[1],p[2]) //[2 3 5 7 11 13 19 23 29]
b := append(primes,p…)             //[2 3 5 7 11 13 19 23 29]
```

Quiz

```
func incr(s []int) {
  for i,n := range s {     //iterates over s, providing
    s[i] = n+1             //the index and value
  }
}

func main() {
  a := make([]int,5)
  fmt.Println(a)
  incr(a)
  fmt.Println(a)
}
```

What does this program print?

Quiz

```
func Filter(s []int, fn func(int) bool) []int {
  var p []int // == nil
  for _, v := range s {
   if fn(v) {
      p = append(p, v)
   }
  }
  return p
}
func main() {
  a := []int{1,2,3,4,5,6,7,8,9,10}
  a = Filter(a,func (x int) bool { return x%2==0 })
  fmt.Println(a)
}
```

…and this one? :)

Structs and Methods

- Composite types in Go are defined as structs:

```
type Person struct {
   name string
   age int
}

func main() {
   p1 := Person{“Bob”,20}
   p2 := Person{name:”Alice”}
   p2.age = 23
   p3 := Person{p2.name,p2.age}
}
```

Structs and Methods

- Structs literals are values

```
func setAgeBad(p Person, age int) {
  p.age = age
}
```

//Modifies a copy

```
func setAgeBetter(p *Person, age int) {
 p.age = age
}

func (p *Person) setAge(age int) {
 p.age = age
}
```

Structs and Methods

- Structs literals are values

```
func setAgeBad(p Person, age int) {
  p.age = age
}

func setAgeBetter(p *Person, age int) {    //Modifies via pointer
 p.age = age
}

func (p *Person) setAge(age int) {
 p.age = age
}
```

Structs and Methods

- Structs literals are values

```
func setAgeBad(p Person, age int) {
  p.age = age
}

func setAgeBetter(p *Person, age int) {
 p.age = age
}

func (p *Person) setAge(age int) {     //Method syntax
 p.age = age
}
```

- Methods can be defined on structs or struct *pointers*.

Quiz

```
type Person struct {
   name string
   age int
}

func (p Person) setAge(age int) { p.age = age }


func main() {
    p := Person{name:"Alice"}
    p.setAge(23)
    fmt.Println(p.age)
}
```

What does this program print?

Quiz

```
type Person struct {
   name string
   age int
}

func (p *Person) setAge(age int) { p.age = age }


func main() {
    p := Person{name:”Alice”}
    p.setAge(23)
    fmt.Println(p.age)
}
```

This is probably the one you want to write.

Interfaces

- Interfaces are just sets of methods.
- A type implements an interface implicitly by implementing its methods.
- Functions (and methods) can take interface valued arguments.

```
type Stringer interface {
  String() string
}

func (p Person) String() string {
  return fmt.Sprintf("%v (%d)", p.name, p.age)
}

…
fmt.Println(Person{name:"Bob",age:23}) //Bob (23)
```

Interfaces

- Interfaces can embed other interfaces:

```
type Reader interface {
  Read(b []byte) (n int, err error)
}

type Writer interface {
  Write(b []byte) (n int, err error)
}

type ReadWriter interface {
  Reader
  Writer
}
```

Error Handling

- Go does not have exceptions.
- For “catastrophic” errors, built-in function **`panic`**.
- Non-fatal errors in Go are just values of **`error`** type:

```
type error interface {
  Error() string
}
```

- Package **`errors`** provides some facilities for manipulating errors.

Error Handling

- A common idiom:

```
func Hello(name string) (string, error) {
   if name == “” {
      return “”, errors.New(“Empty Name”)
   }
   message := fmt.Sprintf("Hi, %v. Welcome!", name)
   return message, nil
}
```

Error Handling

- A common idiom:

```
func Hello(name string) (string, error) {
   if name == “” {
      return “”, errors.New(“Empty Name”)
   }
   message := fmt.Sprintf("Hi, %v. Welcome!", name)
   return message, nil
}
```

Error Handling

- A common idiom:

```
func Hello(name string) (string, error) {
   if name == “” {
      return “”, errors.New(“Empty Name”)
   }
   message := fmt.Sprintf("Hi, %v. Welcome!", name)
   return message, nil
}
```

Error Handling

- On the client side code:

```
func main() {
  reader := bufio.NewReader(os.Stdin)
  text , _ := reader.ReadString('\n')
  text = strings.Replace(text,"\n","",-1)
  message, err := Hello(text)
  if err != nil {
     //…
  }
  fmt.Println(message)
```

Error Handling

- On the client side code:

```
func main() {
  reader := bufio.NewReader(os.Stdin)
  text , _ := reader.ReadString('\n')
  text = strings.Replace(text,"\n","",-1)
   message, err := Hello(text)
   if err != nil {
      //…
   }
   fmt.Println(message)
```

- Design and conventions encourage explicit error checking.

Error Handling

- Custom errors are easy to define:

```
type SyntaxError struct {
    Line int
    Col int
    Token string
}
func (er *SyntaxError) Error() string {
   return fmt.Sprintf("%d:%d: Syntax error on token %v.",
 e.Line, e.Col, e.Token)
}
```

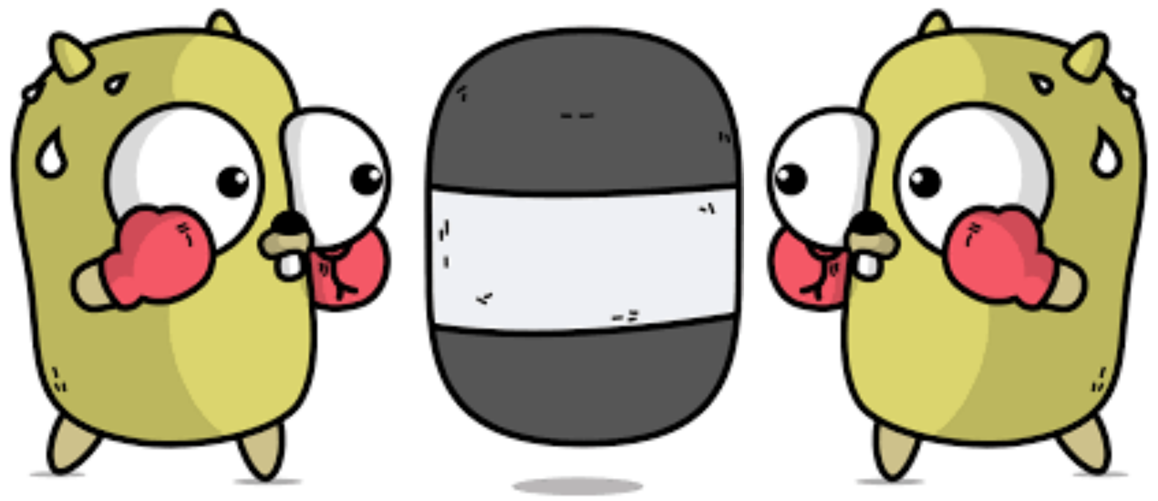
Error Handling

- A common pattern:

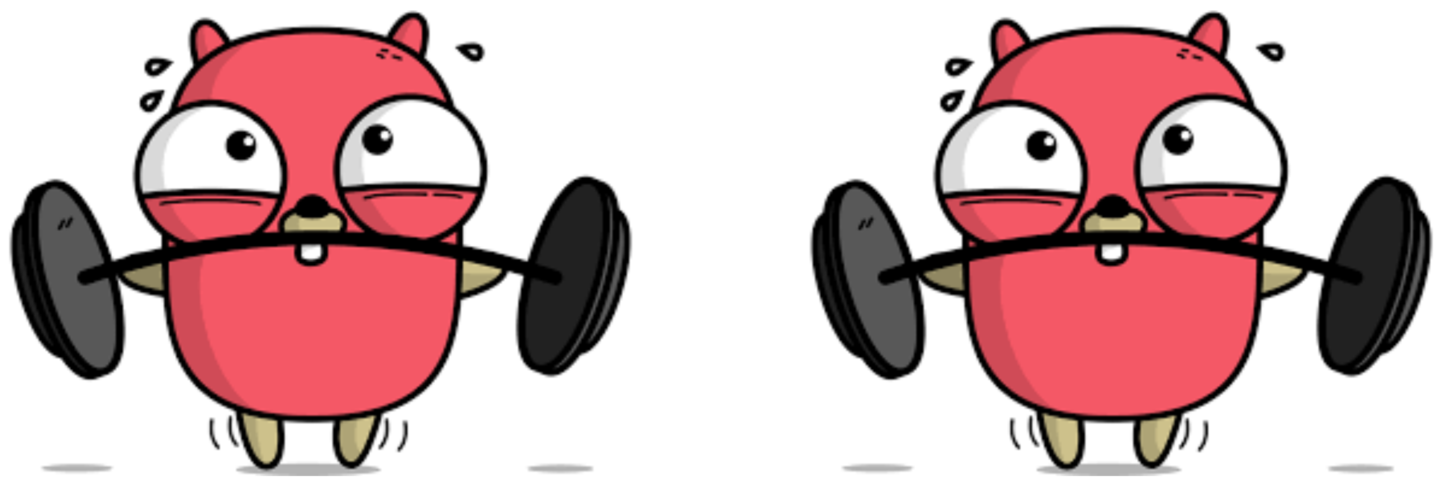
```
ast, err := parse(s)
if err != nil {
   switch err.(type) {
      case SyntaxError:
           …
      case ScopingError:
           …
   }
}
```

Concurrency in Go

- Go supports **concurrency**
  - goroutines
  - channels

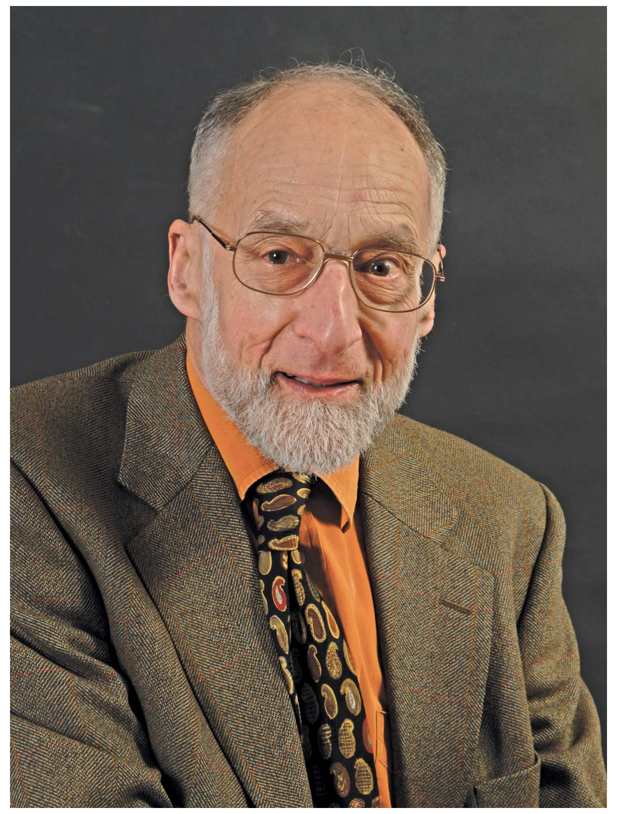


- Go supports **parallelism**
  - multi-processors
  - memory model



Concurrency in Go

- Go favors **channel-based** concurrency over **shared-memory** concurrency.
- Shared-memory concurrency features available in go standard library, mostly for low-level programming (`sync` package)
- Channel-based concurrency built into the language (type system and syntax).
- Inspired by Hoare's CSP, Milner's CCS and π-calculus.

Goroutines

- A running program is made up of **one or more** goroutines.
- A goroutine is a function that:
  - executes **concurrently** to other goroutines
  - in the **same** address space
- Goroutines are **green threads.**

Goroutines vs Threads

- Goroutines are much more lightweight than OS threads
- Stack size
  - **OS threads** - large fixed size
  - **Goroutines** - independent call stack, grows dynamically
- Scheduling
  - **OS threads** - OS…
  - **Goroutines** - Go runtime
    - “clever” scheduling

Goroutines are very inexpensive, can have 1000s!

Concurrency vs Parallelism

- Concurrency is not parallelism, but parallelism is enabled by concurrency!
- Programs can be concurrent and have 0 parallelism.
- Well-written concurrency may run better on a multiprocessor.

Creating Goroutines

- To start a goroutine, just invoke a function and say “**go**”

```
func print_digits() {
  for number := 1; number < 27; number++ {
    fmt.Printf("%d ", number)
  }
}
func main(){
  go print_digits();
  go print_digits();
  fmt.Printf(“finished”);
}
```

Creating Goroutines

- To start a goroutine, just invoke a function and say “**go**”

```
func print_digits() {
  for number := 1; number < 27; number++ {
    fmt.Printf("%d ", number)
  }
}
func main(){
  go print_digits();
  go print_digits();
  fmt.Printf(“finished”);
}
```

When main goroutine finishes,
the program terminates!

```
finished
Program exited.
```

Synchronizing goroutines

- Two ways of synchronizing goroutines in Go :
  - **locks** (package "`sync`")
  - **channels** (synchronous or asynchronous)
- Threaded programming is complex
  - Shared memory and locks are difficult to reason about
  - Risk of latent deadlocks and race conditions

Don't communicate by sharing memory,
share memory by communicating

Channel Types

- Go provides primitives based on **channels** to synchronize go routines.

- In its simplest form, the type looks like this

```
var c chan <payload type>
```

- Channels are a reference type (use make to allocate one)

```
var c = make(chan int)   or   c := make(chan int)
```

Communicating with Channels

```
c:=make(chan int)    //create a channel for int of size 0
                     //(synchronous)

c <- 1               //send 1 on c

v = <- c             //receive on c, assign to v

v := <-c             //receive and initialize v

<-c                  //receives on c and discards value
```

Goroutines & channels

```
func main(){
   c := make(chan bool)
   c <- true
   b := <-c
   fmt.Println(b)
}
```

Goroutines & channels

```
func main(){
   c := make(chan bool)
   c <- true
   b := <-c
   fmt.Println(b)
}
```

main goroutine is stuck on the send, waiting for someone to receive on c…

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:
main.main()
  /Users/btoninho/bitbucket.org/progconc/bad.go:14 +0x59
exit status 2
```

Goroutines & channels

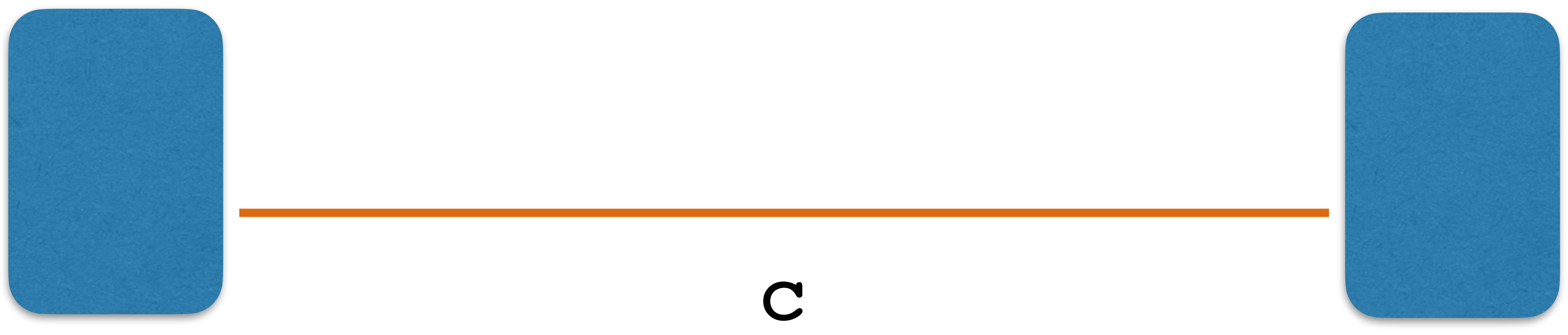
```
func print_digits(c chan bool) {
   for number := 1; number < 27; number++ {
      fmt.Printf("%d ", number)
   }
   c<-true
}

func main(){
   c := make(chan bool)
   go print_digits(c)
   go print_digits(c)
   <-c;<-c;                     //receive twice & discard values
   fmt.Printf("finished");
}
```

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24 25 26 19 20 21 22 23 24 25 26 finished
```

Synchronous channels

```
c := make(chan bool)
```

- By default channels have **size 0** (c has size 0)
- Channels with size 0 support synchronous communication
- Both sender and receiver *block* until a *handshake* can occur



goroutine teacher

```
c <- “kitcat”
```

goroutine student

```
h := <- c
fmt.Printf(“Eating %s“,h)
```

Asynchronous channels

```
c := make(chan bool,5)
```

- c is now a channel with size 5
- Channels of non-0 size provide **asynchronous** comm.
- Sending blocks if channel is full; receive blocks if empty.

```
func main(){
   c := make(chan bool,5)
   c <- true
   b := <-c
   fmt.Println(b)
}
```

```
true
```

Challenges

- Channel-based concurrency tries to alleviate the challenges of sharing data across threads
- Races on shared variables (**Bad**) vs races on channel access (**OK**)
- Not a panacea:
  - Deadlocks (Cyclic dependency on channel accesses)
  - Starvation (Some threads never access what they need)
  - Go programs still manipulate state (races on data structures / variables can still happen, accesses must be disciplined).

That’s it for today…

- Go - the very basics
- Goroutines
- Channels
    - Synchronous and asynchronous channels
    - Basic channel ops
- **Next Lecture**: putting it all to use, selective communication, wait groups, etc.