Go for non-gophers

Natalie Pistunovich
@NataliePis

Most of the Gopher art
is by Ashley McNamara
and thanks to Renée French

About Me

B.Sc. Computer and Software Engineering, Technion, Israel

Backend Developer ATM and in several startups in the past

Hardware Engineering student at Intel

Co-organizer at GDG Berlin Golang and Women Techmakers Berlin

Co-founder of GopherCon Iceland

Workshop plan

- Intro to Go
- Environment Structure
- Syntax
- OSS Adventure Time!

Intro to Go

Logo
Smalltalk
Forth
Self
Ruby
Python
Lisp
Dylan
Objective-C
Lua
AWK
Common Lisp
T
JavaScript
Perl
Simula
Communicating sequential processes
Eiffel
CLU
ALGOL 68
C++
Scheme
SQL
C
Java
Ada
Object Pascal
Oberon
APL
COBOL
C#
Modula-3
BCPL
Modula-2
Pascal
Prolog
Erlang
FP
ALGOL
PL/I
Speedcoding

https://exploringdata.github.io/vis/programming-languages-influence-network/

About go

Go is a free and open source programming language created at Google in 2007 as an answer to scale problems at Google

v1.10 released on 02/2018

It's a **system programming** language,
that is **compiled**,
and **statically typed**,
with a **C-like** syntax,
garbage collection,
and explicit support for **concurrency**.

About go

Programs are constructed from **packages**,
whose properties allow efficient management of
dependencies. Best practice is using the standard lib.

The existing implementations use a traditional
compile/link model to generate executable binaries.

The grammar is compact and regular,
allowing easy analysis by automatic tools
such as **IDEs and code linters**:
https://github.com/golang/go/wiki/
IDEsAndTextEditorPlugins

Who is Using Go?

Who is Using Go?

- Atlassian
- Native Instruments
- Soundcloud
- Sixt
- Zalando
- Fiver
- Data Dog
- OLX
- CloudFlare
- SpaceX
- Sainsburys
- Google
- Youtube
- Twitter
- Microsoft
- VMWare
- Adobe
- BBC
- Booking.com
- BuzzFeed
- Circle CI
- Cockroach labs
- Comcast
- CoreOS
- Dell
- Docker
- DropBo
- eBay
- GitLab
- GitHub
- Kubernetes
- Yahoo
- HashiCorp
- Heroic
- Imgur
- Intel
- Medium
- MongoDB
- Netflix
- New York Times
- Oracle
- Reddit
- Tumblr
- Uber
- RyanAir
- Stripe
- Yandex

https://github.com/golang/go/wiki/GoUsers

Key Features

Binaries

Go generates executable binaries with all the dependencies built-in

Go conventions

Simplicity first

Make your code simple, not clever.
Be merciless in your push for simplicity.
Return an error if something can fail.
Write tests for confidence.
Take out time to test and refactor.

Go conventions

Try avoiding writing code like you would in your current language of choice. Spend some time to learn the “Go way”.

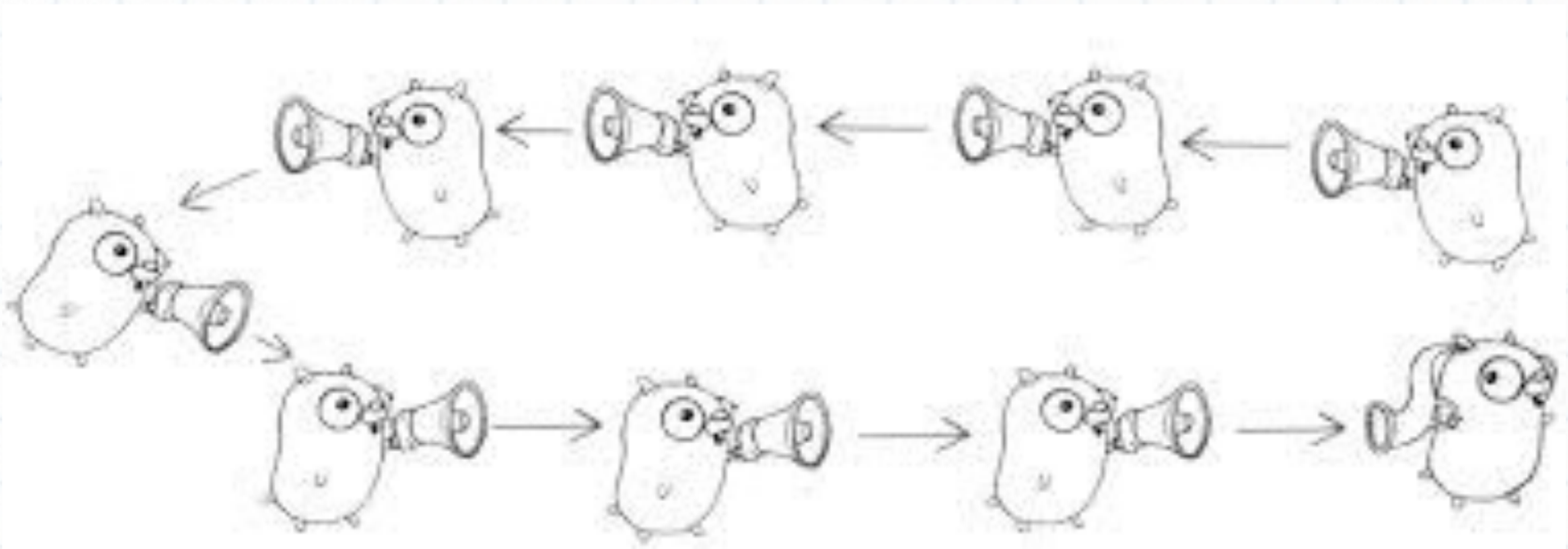
Learn what idiomatic Go looks like and what are the Go conventions, especially those related to documentation, packages and naming.

- Use the go tool chain
- Write tests
- Use interfaces
- Take version numbers seriously
- Document your code for godoc

Imports

Standard Library!

But theres also built in support for getting libraries and sharing them



Embedding over inheritance

Make things small and compose them together, instead of trying to make inheritance hierarchies

Go does not provide the typical, type-driven notion of subclassing, but it does have the ability to “borrow” pieces of an implementation by embedding types within a struct or interface

By embedding the structs directly the methods of embedded types come along for free

Embedding over inheritance

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

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

type ReadWriter interface {
    Reader
    Writer
{
```

Embedding over inheritance

```
type Animal interface {
    Name() string
}

type Dog struct {}

func (d *Dog) Name() string {
   return “Dog”
}

func (d *Dog) Bark() {
   fmt.Println(“Woof!”)
}
```

Embedding over inheritance

```
type Animal interface {
    Name() string
}

type Dog struct {}

func (d *Dog) Name() string {
   return “Dog”
}
func (d *Dog) Bark() {
   fmt.Println(“Woof!”)
}
```

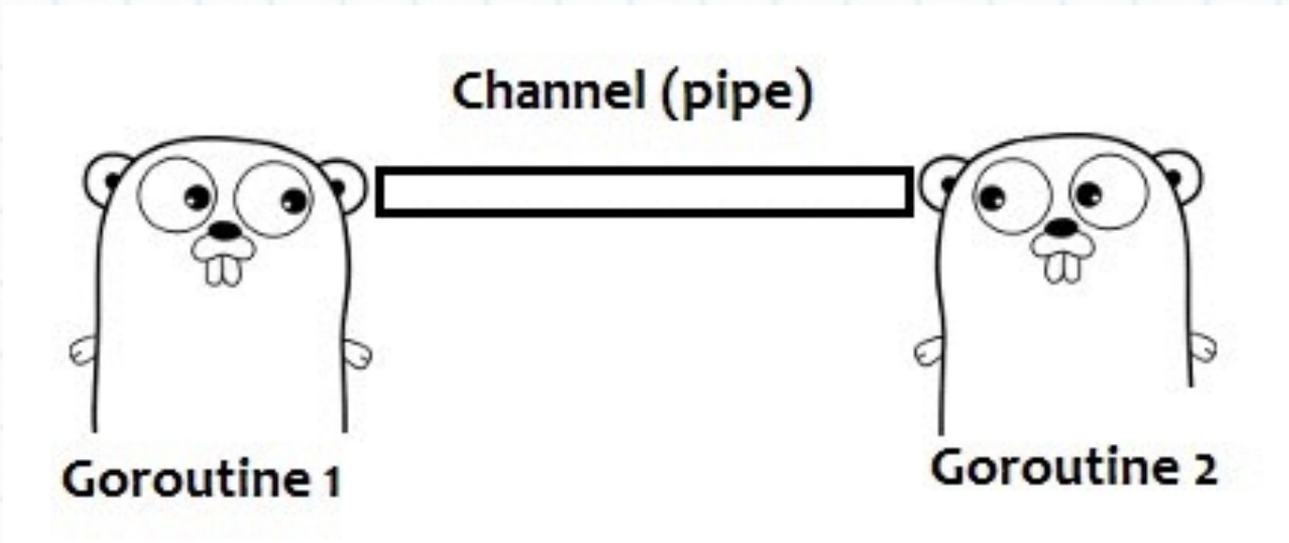
```
func main() {
      var animal Animal
      animal = &Dog{}
     fmt.Println(animal.Name())
}
```

Concurrency

Go Concurrency primitives via go routines and channels

Goroutines are functions or methods that run concurrently with other functions or methods

Channels are the pipes that connect concurrent goroutines. You can send values into channels from one goroutine and receive those values into another goroutine



Concurrency vs. Parallelism

Concurrency: programming as the composition of independently executing processes

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

Concurrency is about **dealing** with lots of things at once.
Parallelism is about **doing** lots of things at once.

Testing

Unit testing is part of the language

The tooling offers easy

- benchmarking
- code coverage
- documentation

Go fmt

```
func {
    //…
}
```

or

```
func
{
    //…
}
```

Go fmt

```
func {
    //…
}
```

or

```
func
{
    //…
}
```

Spaces or tabs?

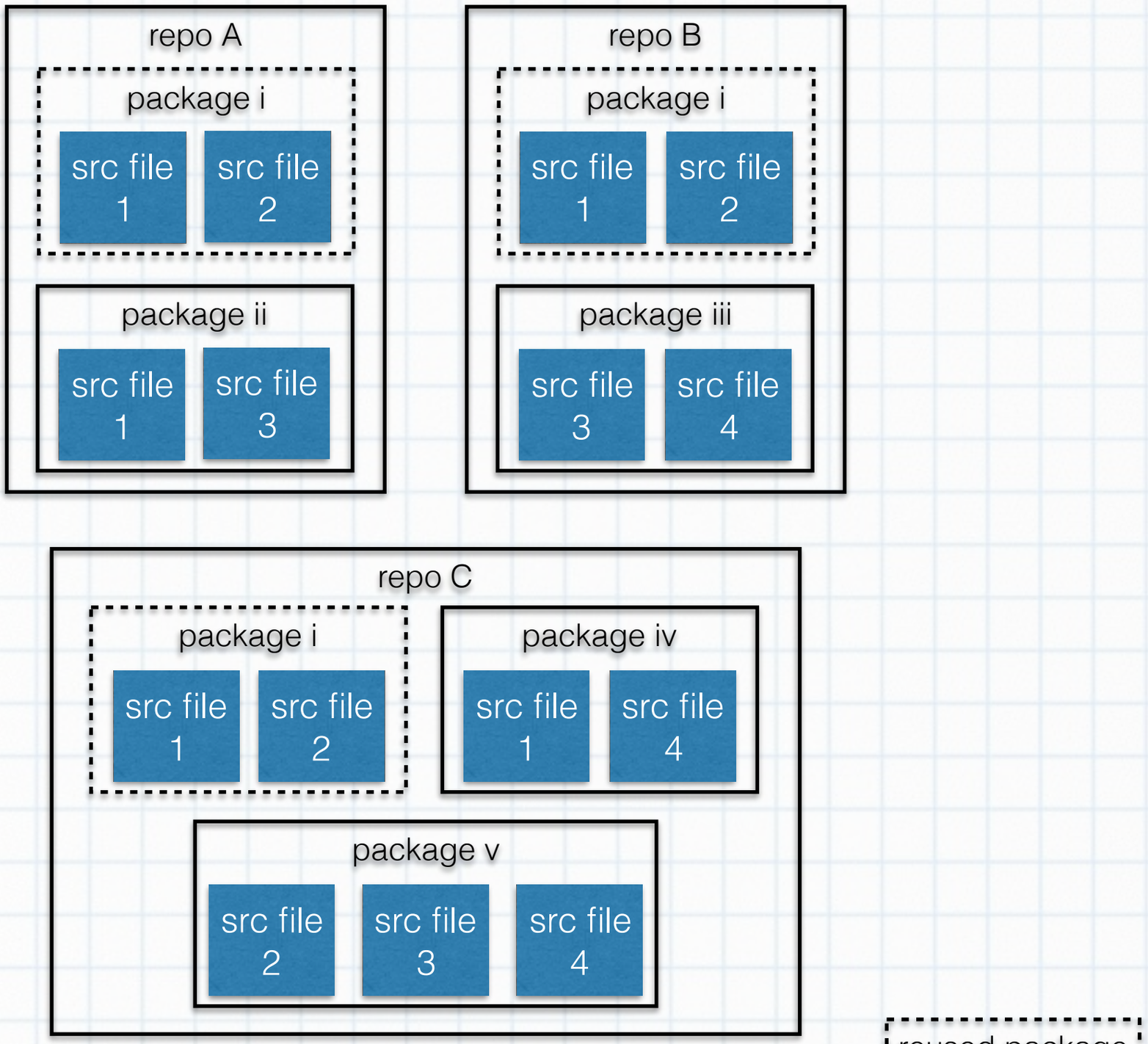
Go fmt

At least the endless formatting discussion is settled!

Go adopts a unique programming paradigm, so be prepared to change the way you design and write code.

Environment Structure

Workspace


repo A
package i
src file 1
src file 2
package ii
src file 1
src file 3
repo B
package i
src file 1
src file 2
package iii
src file 3
src file 4
repo C
package i
src file 1
src file 2
package iv
src file 1
src file 4
package v
src file 2
src file 3
src file 4
reused package

GOPATH

An environment variable, specifies the location of the workspace.

Default: a directory named go in the home directory. Must be at a different path than the Go installation.

In Unix: `$HOME/go`

alternative setup: `GOPATH=$HOME`

In Windows: `%USERPROFILE%\go`

usually `C:\Users\YourName\go`

`go env GOPATH` prints the effective current GOPATH

`go help GOPATH` prints more about GOPATH

Import path

Identifies a package corresponding to the location in a workspace/remote repo:

```
import (
  "net/http"
  “github.com/pisush/smth"
  . ”github.com/pisush/smth“
  short “github.com/pisush/very_long_name”
  _ “github.com/pisush/just_for_side_effect”
)
```

Import path

Identifies a package corresponding to the location in a workspace/remote repo:

```
import (
  "net/http"
  “github.com/pisush/smth"
  . ”github.com/pisush/smth“
  short “github.com/pisush/very_long_name”
  _ “github.com/pisush/just_for_side_effect”
)
```

```
 httpRequest, err := http.NewRequest("POST",
 path, requestBody)
```

Import path

Identifies a package corresponding to the location in a workspace/remote repo:

```
import (
  "net/http"
  “github.com/pisush/smth"
  . ”github.com/pisush/smth“
  short “github.com/pisush/very_long_name”
  _ “github.com/pisush/just_for_side_effect”
)
```

```
a := smth.foo()
```

Import path

Identifies a package corresponding to the location in a workspace/remote repo:

```
import (
  "net/http"
  “github.com/pisush/smth"
  . "github.com/pisush/smth“
  short “github.com/pisush/very_long_name”
  _ “github.com/pisush/just_for_side_effect”
)
```

```
a := foo()
```

Import path

Identifies a package corresponding to the location in a workspace/remote repo:

```
import (
  "net/http"
  “github.com/pisush/smth"
  . ”github.com/pisush/smth“
  short “github.com/pisush/very_long_name”
  _ “github.com/pisush/just_for_side_effect”
)
```

```
a := short.foo()
```

Import path

Identifies a package corresponding to the location in a workspace/remote repo:

```
import (
  "net/http"
  “github.com/pisush/smth"
  . ”github.com/pisush/smth“
  short “github.com/pisush/very_long_name”
  _ “github.com/pisush/just_for_side_effect”
)
```

Syntax

Hello!

main.go:

```
package main

import "fmt"

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

```
$ go run main.go
Hello, world!
```

Hello repo!

```
$ mkdir -p $GOPATH/src/github.com/user/hello
$ cd $GOPATH/src/github.com/user/hello
```

main.go:

```
package main

import "fmt"

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

```
$ go install
$ hello
Hello, world!
```

Function signatures

```
func add(x int, y int) int {
  return x + y
}
```

Function signatures

```
func add(x int, y int) int {
  return x + y
}

func add(x, y int) int {
  return x + y
}
```

Function signatures

```
func add(x int, y int) int {
  return x + y
}

func add(x, y int) int {
  return x + y
}

func swap(x, y string) (string, string) {
  return y, x
}
```

Function signatures

```
func add(x int, y int) int {
  return x + y
}

func add(x, y int) int {
  return x + y
}

func swap(x, y string) (string, string) {
  return y, x
}

func split(sum int) (x, y int) {
  x = sum/2
  y = sum - x
  return
}
```

Variable declaration

A var statement can be at the package or function level:

```
var a, b, c bool
var x, y int = 1, 2
```

Variable declaration

A var statement can be at the package or function level:

```
var a, b, c bool // Always initiated
var x, y int = 1, 2
```

Variable declaration

A var statement can be at the package or function level:

```
var a, b, c bool // Always initiated
var x, y int = 1, 2
```

If an initializer is present, the type can be omitted:

```
var i, j, k = true, 3, “cat”
```

Variable declaration

A var statement can be at the package or function level:

```
var a, b, c bool // Always initiated
var x, y int = 1, 2
```

If an initializer is present, the type can be omitted:

```
var i, j, k = true, 3, “cat”
```

The short assignment can be used *inside functions* (outside every statement must begin with a keyword):

```
func main() {
  i, j, k := true, 3, "cat"
  var f int = j
}
```

Type conversions

Type conversions are explicit:

```
func main() {
  var j int = 3
  var k float64 = i;
}
```

Type conversions

Type conversions are **explicit**:

```
func main() {
  var j int = 3
  var k float64 = i; // Haha, nope, just kidding!
  var k float64 = float64(i);
}
```

Consts

Can be char, string, bool, numeric values:

```
const Pi = 3.14

func main() {
  const Pi = 3.14
}
```

For

The only looping construct in Go:

```
for i := 0; i < 10; i++ {
  fmt.Println(i)
}
```

For

The only looping construct in Go:

```
for i := 0; i < 10; i++ {
  fmt.Println(i)
}
```

The init and post statements are optional:

```
sum := 1
for ; sum < 1000 ; {
  sum += sum
}
```

For

The only looping construct in Go:

```
for i := 0; i < 10; i++ {
  fmt.Println(i)
}
```

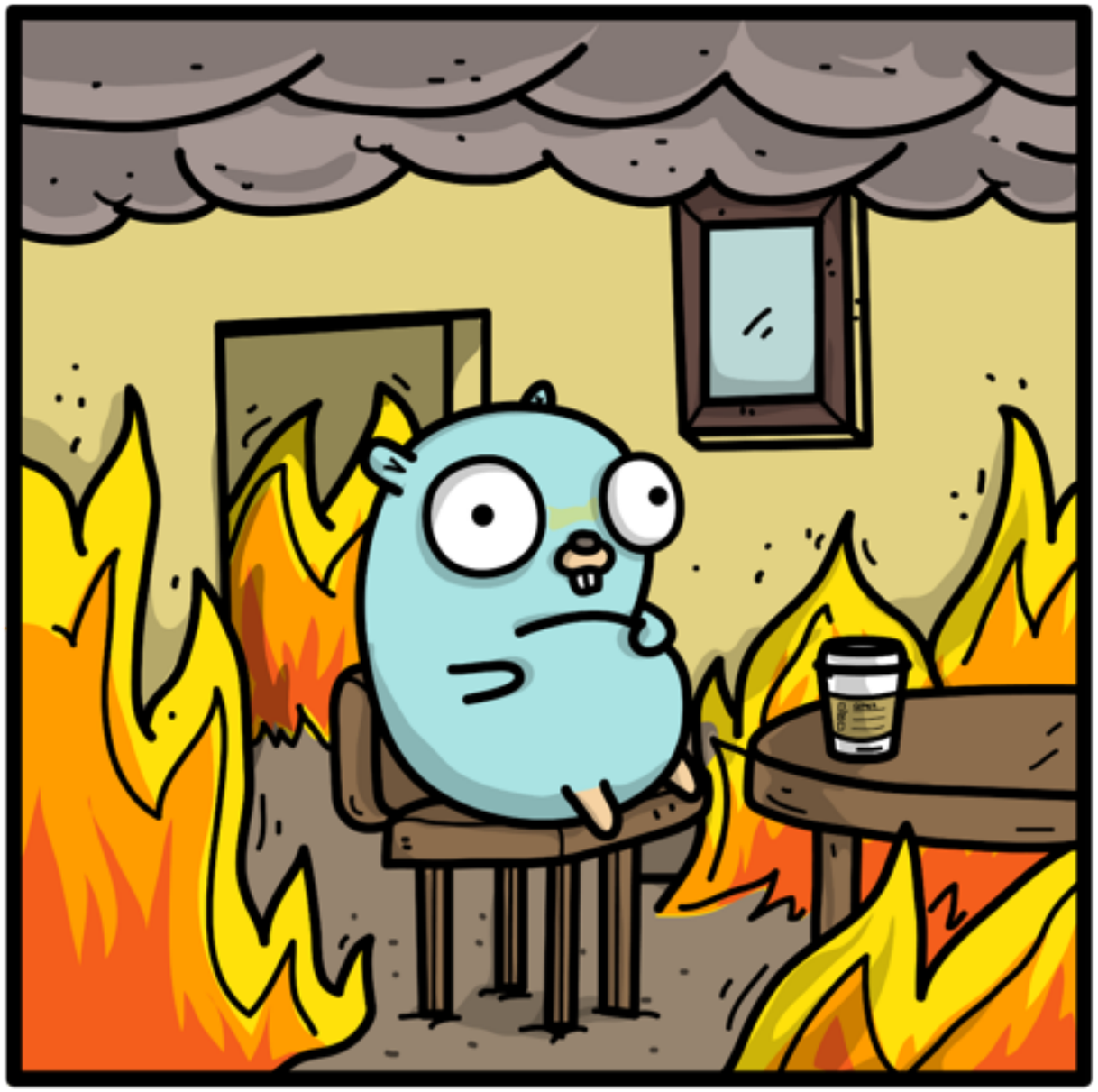
The init and post statements are optional:

```
sum := 1
for ; sum < 1000 ; {
  sum += sum
}
```

It's a *while*:

```
for sum < 1000 {
  sum += sum
}
```

for {
}




THIS IS FINE

If

```
if x < 0 {
  fmt.Println(x)
}
```

If

```
if x < 0 {
  fmt.Println(x)
}
```

Variables can be declared by an if statement, but are only in scope until its end (including all else blocks):

```
if v := a + b; v > 20 {
  fmt.Println("More than 20!")
} else if v > 10 {
  fmt.Println("More than 10!")
} else {
  fmt.Println("10 or less.")
}
fmt.Println(v) // Nope, not in scope!
```

Switch

A case body breaks automatically, unless it ends with a fall-through statement.

Switch statements evaluate cases from top to bottom.

```
switch os := runtime.GOOS; os {
  case "darwin":
    fmt.Println("OS X.")
  case "linux":
    fmt.Println("Linux.")
  default:
    fmt.Printf("%s.", os)
}
```

Switch

Switch without a condition is a clean way to write a long if-then-else:

```
switch {
  case t.Hour() > 9:
    fmt.Println("hi")
  case t.Hour() < 17 && t.Minute() > 30:
    fmt.Println("bye")
  default:
    fmt.Println(“coffee?")
}
```

Defer

Defers the execution of a function.

The arguments are evaluated immediately, and the function is executed only when the surrounding function returns:

```
func main() {
 for i := 0; i < 2; i++ {
    defer fmt.Println(i)
  }
  fmt.Println("kthxbye")
}
```

Defer

Defers the execution of a function. The arguments are evaluated immediately, and the function is executed only when the surrounding function returns:

```
func main() {
 for i := 0; i < 2; i++ {
    defer fmt.Println(i)
  }
  fmt.Println("kthxbye")
}
```

Defers are stacked as LIFO:

```
$
```

Defer

Defers the execution of a function. The arguments are evaluated immediately, and the function is executed only when the surrounding function returns:

```
func main() {
 for i := 0; i < 2; i++ {
    defer fmt.Println(i)
  }
  fmt.Println("kthxbye")
}
```

Defers are stacked as LIFO:

```
$ kthxbye
2
1
0
```

Pointers

```
func main() {
  i := 42
  p := &i
  *p = 21
  fmt.Println(i)
}
```

```
$
```

Pointers

```
func main() {
  i := 42
  p := &i
  *p = 21
  fmt.Println(i)
}
```

```
$ 21
```

Pointers

```
func main() {
  i := 42
  p := &i
  *p = 21
  fmt.Println(i)
}
```

```
21
```

No pointer arithmetic though. (Yay!)

Structs

```
type Entity struct {
    ID     string
    Name   string
}
```

Structs

```
type Entity struct {
    ID     string
    Name   string
}
```

Mapping JSON fields:

```
type jsonEntity struct {
    ID       string    `json:"category_id"`
    Name     string    `json:"category_name"`
    password int       `json:”pass, omitempty”`
}
```

Structs

```
type Entity struct {
    ID     string
    Name   string
}
```

Mapping JSON fields:

```
type jsonEntity struct {
    ID       string    `json:"category_id"`
    Name     string    `json:"category_name"`
    password int       `json:"pass, omitempty"`
}
```

Struct literals and accessing fields:

```
entity := Entity{“id”, “name”}
entity.ID = “new_id”
```

Arrays and Slices

Arrays cannot be resized: length is part of the type. Slices are sized dynamically.

```
var myArray [2]string
var mySlice []int

myOtherArray := [6]int{2, 3, 5, 7, 11, 13}
myOtherSlice := myOtherArray[:2]

for index, value := range myOtherSlice {
    fmt.Printf(value)
}
```

Slices are references to a section of an underlying array. Changing elements in a slice modifies the corresponding elements of its underlying array.

Maps

Simple mapping of keys to values. Maps are instantiated with **make()**:

```
var emotions map[string]float64

emotions = make(map[string]float64)
emotions["joy"] = 0.84
emotions["anger"] = 0.03

emotion, ok := emotions[“anger"] // 0.03, true
delete(emotions,”anger”)
emotion, ok = emotions[“anger"]  // 0, false
```

Creating map literals:

```
var emotions map[string]float64{
    "joy": 0.84
    "anger": 0.03
}
```

Methods

Go does not have classes, but there are methods on types:

```
type Connection struct {
    URL      string
    Version  string
    Username string
    Password string
}

func (c Connection) getData(text string) Response
{
   ...
}

Data := c.getData(text)
```

Interfaces

An interface type is a set of method signatures:

```
type Connection interface {
    IsOpen() bool
}
```

Interfaces are implicit:

```
func (sc TwitterConn) IsOpen() bool {
  return "I implement the Connection interface!"
}
```

And can be required instead of types:

```
func fetchData(c Connection) string { ... }
```

Error handling

Functions often return an error value.

Your code should handle errors by testing whether the error equals **nil**:

```
resp, err := client.Do(req)
    if err != nil {
        fmt.Printf("Got an error: %s", err)
    }
fmt.Printf("Correct response with length %d",
resp.ContentLength)
```

Adventure Time

Open Source

label:"good first issue"

P requests Issues Marketplace Explore

Browse activity | Discover repositories

Saved replies keyboard short
Now saved replies have keyboard shortcuts to make them even easier

View new

Languages

JavaScript	14,032
Python	8,996
Java	4,238
C++	3,423
PHP	3,276
Go	2,517
C	1,823

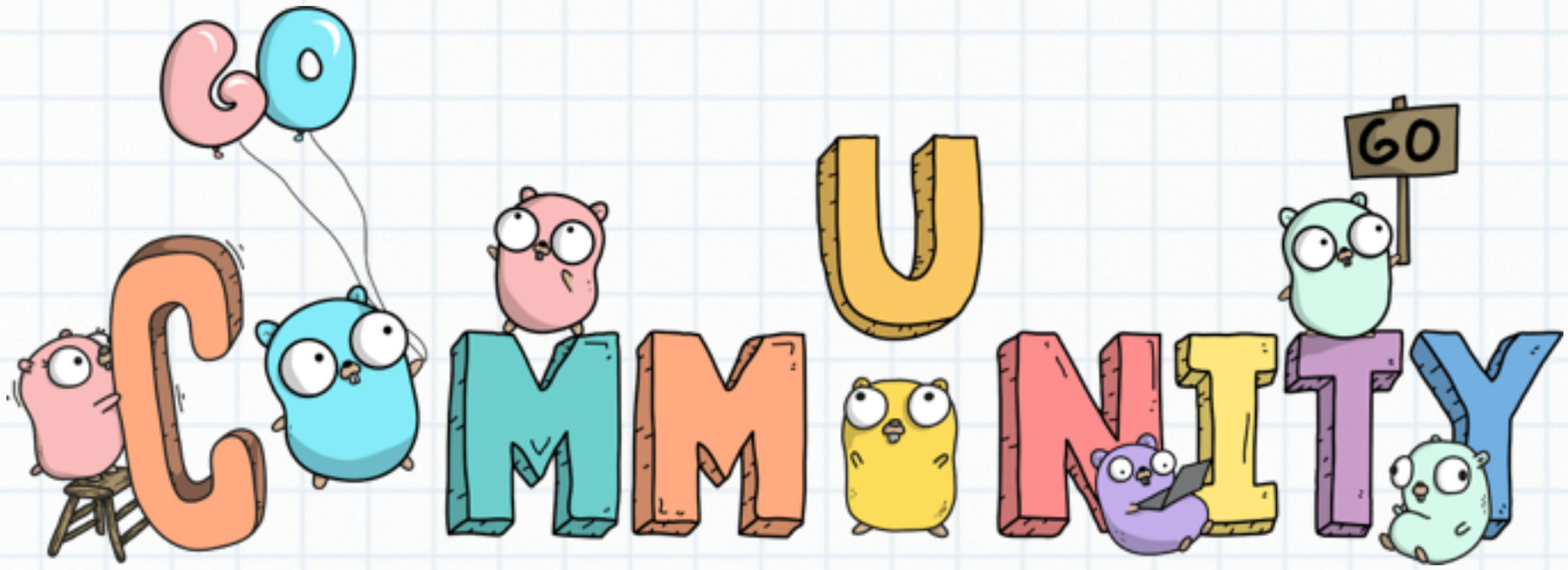
References

A Tour of Go
https://tour.golang.org

Go by Example
https://gobyexample.com

Learn X in Y Minutes
https://learnxinyminutes.com/docs/go

Effective Go
https://golang.org/doc/effective_go.html



Krakow Go user group ?

https://github.com/golang/go/wiki/GoUserGroups

golang-pl - Poland.
Gophers Katowice - Katowice, Poland.
Golang Warsaw - Warsaw, Poland.
G.L.U.G. Wroclaw - Wroclaw, Poland
Golang User Group Trójmiasto - Gdańsk/Gdynia/Sopot, Poland

Gophers Slack!

Join us: https://invite.slack.golangbridge.org/

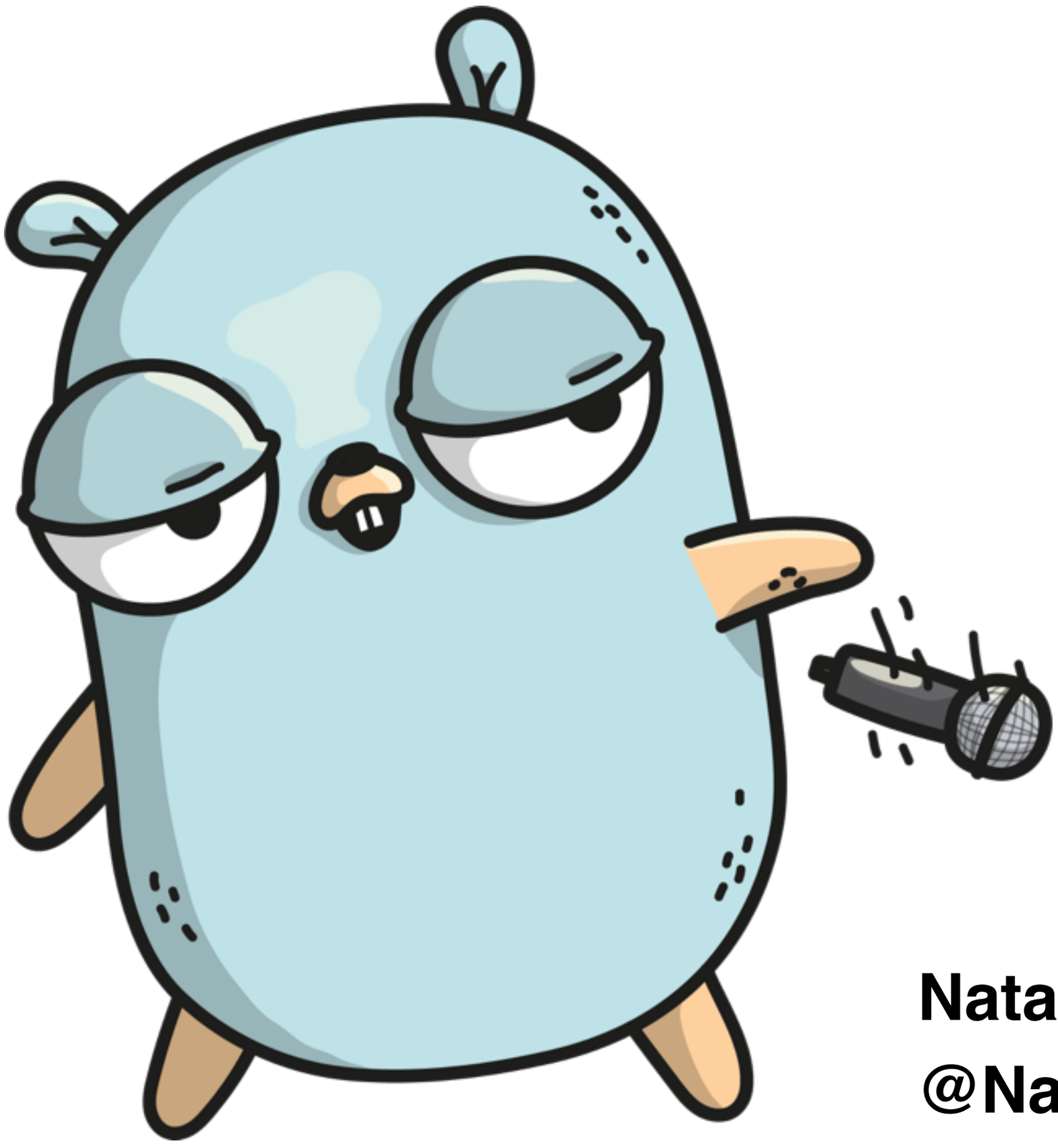
Wrap up

Krakow Go user group ?
https://github.com/golang/go/wiki/GoUserGroups

golang-pl - Poland.
Gophers Katowice - Katowice, Poland.
Golang Warsaw - Warsaw, Poland.
G.L.U.G. Wroclaw - Wroclaw, Poland
Golang User Group Trójmiasto - Gdańsk/Gdynia/Sopot, Poland

There are lots of projects looking for love
https://github.com/golang/go/wiki/Projects

And some cool conferences to attend!
https://github.com/golang/go/wiki/Conferences



Natalie Pistunovich

@NataliePis