An Introduction to Go

Why and how to write good Go code

@francesc

Francesc Campoy

VP of Product & Developer Relations

source{d}

Previously:

- Developer Advocate at Google
  - Go team
  - Google Cloud Platform

twitter.com/francesc | github.com/campoy

```
// HTMLEscape appends to dst the JSON-encoded src with <, >, &, U+2028 a
// characters inside string literals changed to \u003c, \u003e, \u0026,
// so that the JSON will be safe to embed inside HTML <script> tags.
// For historical reasons, web browsers don't honor standard HTML
// escaping within <script> tags, so an alternative JSON encoding must
// be used.
func HTMLEscape(dst *bytes.Buffer, src []byte) {
    // The characters can only appear in stri    literals,
    // so just scan the string one byte at a time.
    start := 0
    for i, c := range src {
        if c == '<' || c == '>' || c == '&'
            if start < i {
                dst.Write(src[start:i])
            }
            dst.WriteString(`\u00`)
            dst.WriteByte(hex[c>>4])
            dst.WriteByte(hex[c&0xF])
            start = i + 1
        }
        // Convert U+2028 and U+2029 (E2 80 A8 a
        if c == 0xE2 && i+2 < len(src) && src[i+
            if start < i {
                dst.Write(src[start:i])
            }
            dst.WriteString(`\u202`)
            dst.WriteByte(hex[src[i+2]&0xF])
            start = i + 3
        }
    }
    if start < len(src) {
        dst.Write(src[start:])
    }
}
```

```
func (e *MarshalerError) Error() string {
    return "json: error calling MarshalJSON for type " + e.Type.String() + ": " + e.Err
}

var hex = "0123456789abcdef"

// An encodeState encodes JSON into a bytes.Buffer.
type encodeState struct {
    bytes.Buffer    umulated output
    cratch          4]byte
```

just for

JustForFunc: Programming in Go

20,990 subscribers • 58 videos

Series of talk recordings and screencasts mainly about Go and the Google

The Go Programming Language

22,831 subscribers • 55 videos

Videos about working with the **Go Programming** Language.

Agenda

Day 1

- Go basics
- Type System

Day 2

- Concurrency

Day 3

- Performance Analysis
- Tooling
- Advanced Topics
- Q&A

Day 1

Agenda

- Go basics
- Go’s Type System
- Go’s Standard Library Overview
- Q&A

What is Go?

An open source (BSD licensed) project:

- Language specification,
- Small runtime (garbage collector, scheduler, etc),
- Two compilers (gc and gccgo),
- A standard library,
- Tools (build, fetch, test, document, profile, format),
- Documentation.

Language specs and std library are backwards compatible in Go 1.x.

Go 1.x

Released in March 2012

A specification of the language and libraries supported for years.

The guarantee: code written for Go 1.0 will build and run with Go 1.x.

Best thing we ever did.

What is Go about?

Go is about composition.

Composition of:

- Types:
  - The type system allows bottom-up design.
- Processes:
  - The concurrency principles of Go make process composition straight-forward.
- Large scale systems:
  - The packaging and access control system and Go tooling all help on this.

Hello, CERN!

```go
package main

import "fmt"

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

Hello, CERN!

```go
package main

import "fmt"

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

Packages

All Go code lives in packages.

Packages contain **type, function, variable, and constant** declarations.

Packages can be very small (package errors has just one declaration) or very large (package net/http has >100 declarations).

Case determines visibility:

Foo is exported, foo is not.

Hello, CERN!

```go
package main

import "fmt"

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

Hello, CERN!

```
package main

import "fmt"

func main() {
    fmt.println("Hello, CERN")
}
```

```
prog.go:4:5: cannot refer to unexported name fmt.println
prog.go:4:5: undefined: fmt.println
```

More packages

Some packages are part of the standard library:

- “fmt”: formatting and printing
- “encoding/json”: JSON encoding and decoding

golang.org/pkg for the whole list

Convention: package names match the last element of the import path.

```
import “fmt”            →   fmt.Println
import “math/rand”      →   rand.Intn
```

More packages

All packages are identified by their import path

- “github.com/golang/example/stringutil”
- “golang.org/x/net”

You can use godoc.org to find them and see their documentation.

```
$ go get github.com/golang/example/hello
$ ls $GOPATH/src/github.com/golang/example/hello
hello.go
$ $GOPATH/bin/hello
Hello, Go examples!
```

Understanding GOPATH

A Go workspace resides under a single directory: GOPATH.

```
$ go env GOPATH
```

- defaults to $HOME/go
- will maybe disappear soon (Go modules)

Three subdirectories:

- src: Go source code, your project but also all its dependencies.
- bin: Binaries resulting from compilation.
- pkg: A cache for compiled packages

Hello, CERN!

```
package main

import (
    "fmt"
    "github.com/golang/example/stringutil"
)

func main() {
    msg := stringutil.Reverse("Hello, CERN")
    fmt.Println(msg)
}
```

Further workspace topics

Dependency management:

- vendor directories
- dep / Go modules

Workspace management:

- internal directories
- The `go list` tool

More info: github.com/campoy/go-tooling-workshop

Type System

Go Type System

Go is statically typed:

```
var s string = “hello”
s = 2.0
```

```
cannot use 2 (type float64) as type string in assignment
```

But it doesn’t feel like it:

```
s := “hello”
```

More types with less typing.

Variable declaration

Declaration with name and type

```
var number int

var one, two int
```

Declaration with name, type, and value

```
var number int = 1

var one, two int = 1, 2
```

Variable declaration

Short variable declaration with name and value

```
number := 1

one, two := 1, 2
```

Default values:

integer literals:	42	int
float literals:	3.14	float64
string literal:	“hi”	string
bool literal:	true	bool

abstract types

---

concrete types

abstract types

concrete types

concrete types in Go

- they describe a memory layout

- behavior attached to data through methods

The predefined types

Numerical:

int, int8, int16, int32 (rune), int64
uint, uint8 (byte), uint16, uint32, uint64
complex64, complex128
uintptr

Others

bool, string, error

Creating new types

Arrays:

```
type arrayOfThreeInts [3]int
```

Slices:

```
type sliceOfInts []int
```

Maps:

```
type mapOfStringsToInts map[string]int
```

Creating new types

Functions:

```
type funcIntToInt func(int) int

type funcStringToIntAndError func(string) (int, error)
```

Channels:

```
type channelOfInts chan int

type readOnlyChanOfInts chan <-int

type writeOnlyChanOfInts chan int<-
```

Creating new types

Structs:

```
type Person struct {
    Name string
    AgeYears int
}
```

Pointers:

```
type pointerToPerson *Person
```

Slices and arrays

Slices are of dynamic size, arrays are not.

You probably want to use slices.

```
var s []int
fmt.Println(len(s)) // 0
s = append(s, 1)    // [1]
```

```
s := make([]int, 2)
fmt.Println(len(s)) // 2
fmt.Println(s)      // {0,0}
```

Sub-slicing

You can obtain a section of a slice with the `[:]` operator.

```
s := []int{0, 1, 2, 3, 4, 5}  // [0, 1, 2, 3, 4, 5]
t := s[1:3]                   // [1, 2]
t := u[:3]                    // [0, 1, 2]
t := s[1:]                    // [1, 2, 3, 4, 5]
t[0] = 42
fmt.Println(s)                // [0, 42, 2, 3, 4, 5]
```

Maps

Their default value is not usable other than for reading

```
m := make(map[int]string)
m[1] = “one”
delete(m, 1)
```

```
m := map[int]string{1: “one”}
fmt.Println(len(m)  // 1
fmt.Println(m[1])   // “one”
```

Functions

They can return multiple values.

```
func double(x  int) int{ return 2 *x }

func div(x, y int) (int, error)  { … }

func splitHostIP(s string) (host, ip string)   { … }

var even func(x int) bool

even := func(x int) bool { return x%2 == 0 }
```

More functions

Functions can be used as any other value.

```
func fib() func() int {
    a, b := 0, 1
    return func() int {
        a, b = b, a+b
        return a
    }
}

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

Closures

Lexical scope is great!

```
func fib() func() int {
    a, b := 0, 1
    return func() int {
        a, b = b, a+b
        return a
    }
}

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

Closures

Lexical scope is great!

```
var a, b int = 0, 1

func fib() func() int {
    return func() int {
        a, b = b, a+b
        return a
    }
}
```

Structs

Structs are simply lists of fields with a name and a type.

```
type Person struct {
    AgeYears int
    Name string
}
```

```
me := Person{35, “Francesc”}
```

```
me := Person{Age: 35}
```

```
fmt.Println(me.Name)    // Francesc
```

Methods declaration

Given the previous Person struct type:

```
func (p Person) Major() bool { return p.AgeYears >= 18 }
```

The `(p Person)` above is referred to as the receiver.

When a method needs to modify its receiver, it should receive a pointer.

```
func (p *Person) Birthday() { p.AgeYears++ }
```

Go is “pass-by-value”

In Go, all parameters are passed by value:

- The function receives a copy of the original parameter.

But, some types are “reference types”:

- Pointers
- Maps
- Channels

Note: Slices are not reference types per-se, but share backing arrays.

*gzip.Writer

*strings.Reader

[]bool

int

*os.File

Defining Methods

Methods can be declared on any named type

Methods can be also declared on non-struct types.

```
type Number int
func (n Number) Positive() bool { return n >= 0 }
```

But also:

```
type mathFunc func(float64) float64
func (f mathFunc) Map(xs []float64) []float64 { … }
```

Methods can be defined **only** on named types defined in this package.

Go does not support inheritance

Go does not support inheritance

There's good reasons for this.

Weak encapsulation due to inheritance is a great example of this.

A Runner class

```
class Runner {
    private String name;

    public Runner(String name) { this.name = name; }

    public String getName() { return this.name; }

    public void run(Task task) { task.run(); }

    public void runAll(Task[] tasks) {
        for (Task task : tasks) { run(task); }
    }
}
```

A RunCounter class

```
class RunCounter extends Runner {
    private int count;

    public RunCounter(String message) { super(message); this.count = 0; }

    @override public void run(Task task) { count++; super.run(task); }

    @override public void runAll(Task[] tasks) {
            count += tasks.length;
            super.runAll(tasks);
        }

    public int getCount() { return count; }
}
```

Let's run and count

What will this code print?

```
RunCounter runner = new RunCounter("my runner");

Task[] tasks = { new Task("one"), new Task("two"), new Task("three")};

runner.runAll(tasks);

System.out.printf("%s ran %d tasks\n",
    runner.getName(), runner.getCount());
```

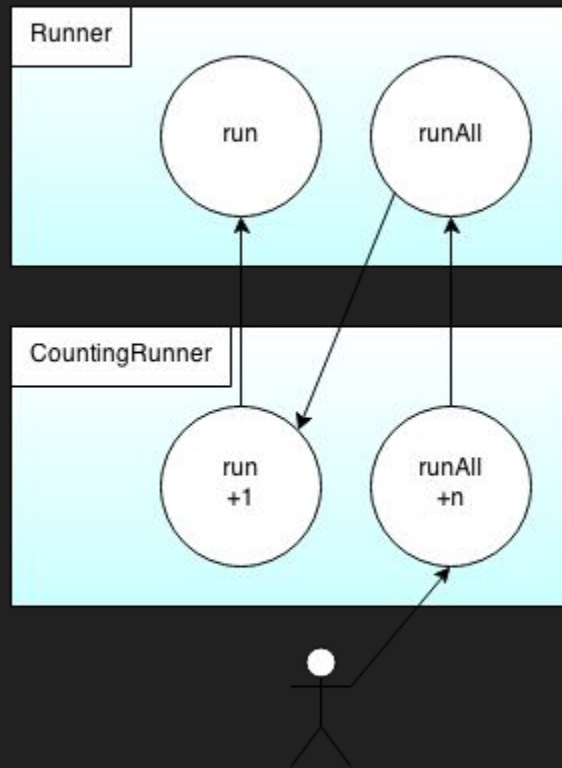
Of course, this prints

```
running one
running two
running three
my runner ran 6 tasks
```

Wait … what?

Inheritance causes:

- weak encapsulation,
- tight coupling,
- surprising bugs.



A *correct* RunCounter class

```
class RunCounter {
    private Runner runner;
    private int count;

    public RunCounter(String message) {
        this.runner = new Runner(message);
        this.count = 0;
    }

    public void run(Task task) { count++; runner.run(task); }

    public void runAll(Task[] tasks) {
        count += tasks.length;
        runner.runAll(tasks);
    }
…
```

A *correct* RunCounter class *(cont.)*

```
…

    public int getCount() {
        return count;
    }

    public String getName() {
        return runner.getName();
    }
}
```

Solution: use composition

Pros:

- The bug is gone!
- Runner is completely independent of RunCounter.
- The creation of the Runner can be delayed until (and if) needed.

Cons:

- We need to explicitly define the Runner methods on RunCounter:

```
public String getName() { return runner.getName(); }
```

- This can cause lots of repetition, and eventually bugs.

The Go way: type Runner

```
type Runner struct{ name string }

func (r *Runner) Name() string { return r.name }

func (r *Runner) Run(t Task) {
    t.Run()
}

func (r *Runner) RunAll(ts []Task) {
    for _, t := range ts {
        r.Run(t)
    }
}
```

The Go way: type RunCounter

```
type RunCounter struct { runner Runner; count  int}

func New(name string) *RunCounter {  return &RunCounter{Runner{name}, 0} }

func (r *RunCounter) Run(t Task) { r.count++; r.runner.Run(t) }

func (r *RunCounter) RunAll(ts []Task) {
    r.count += len(ts);
    r.runner.RunAll(ts)
}

func (r *RunCounter) Count() int { return r.count }

func (r *RunCounter) Name() string { return r.runner.Name() }
```

Struct embedding

Expressed in Go as unnamed fields in a struct.

It is still composition.

The fields and methods of the embedded type are exposed on the embedding type.

Similar to inheritance, but the embedded type doesn't know it's embedded, i.e. no *super*.

The Go way: type RunCounter

```
type RunCounter struct {
    Runner
    count  int
}

func New(name string) *RunCounter2 { return &RunCounter{Runner{name}, 0} }

func (r *RunCounter) Run(t Task) { r.count++;  r.Runner.Run(t) }

func (r *RunCounter) RunAll(ts []Task) {
    r.count += len(ts)
    r.Runner.RunAll(ts)
}

func (r *RunCounter) Count() int { return r.count }
```

Is struct embedding like inheritance?

No, it is better! It is composition.

- You can't reach into another type and change the way it works.
- Method dispatching is explicit.

It is more general.

- Struct embedding of interfaces.

The error type

This is the only predeclared type that is not a concrete.

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

Error handling is done with error values, not exceptions.

```
if err := doSomething(); err != nil {
    return fmt.Errorf(“couldn’t do the thing: %v”, err)
}
```

abstract types

concrete types

abstract types in Go

- they describe behavior

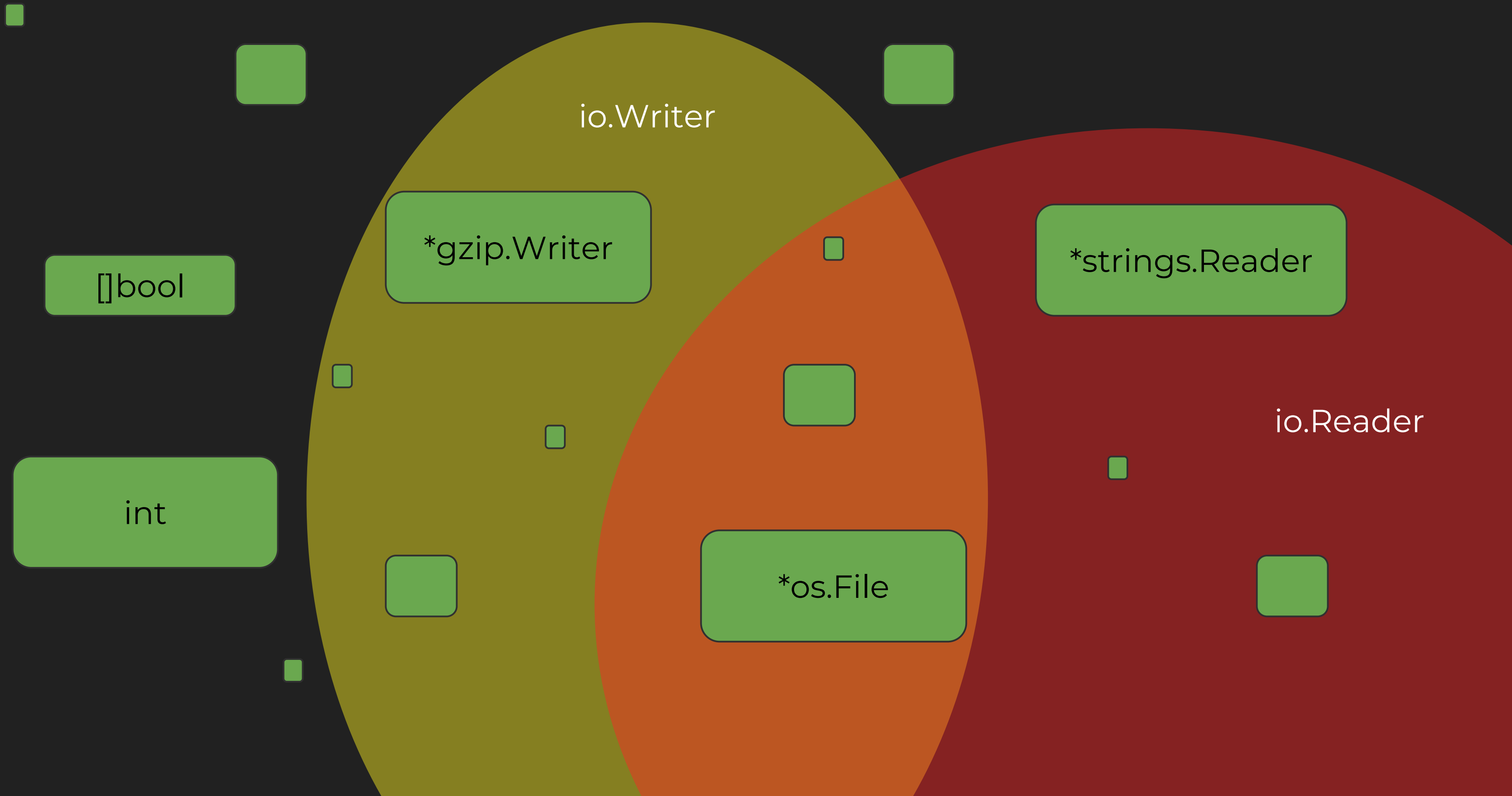
io.Reader | io.Writer | fmt.Stringer

- they define a set of methods, without specifying the receiver

```
type Positiver interface {
    Positive() bool
}
```

two interfaces

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


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

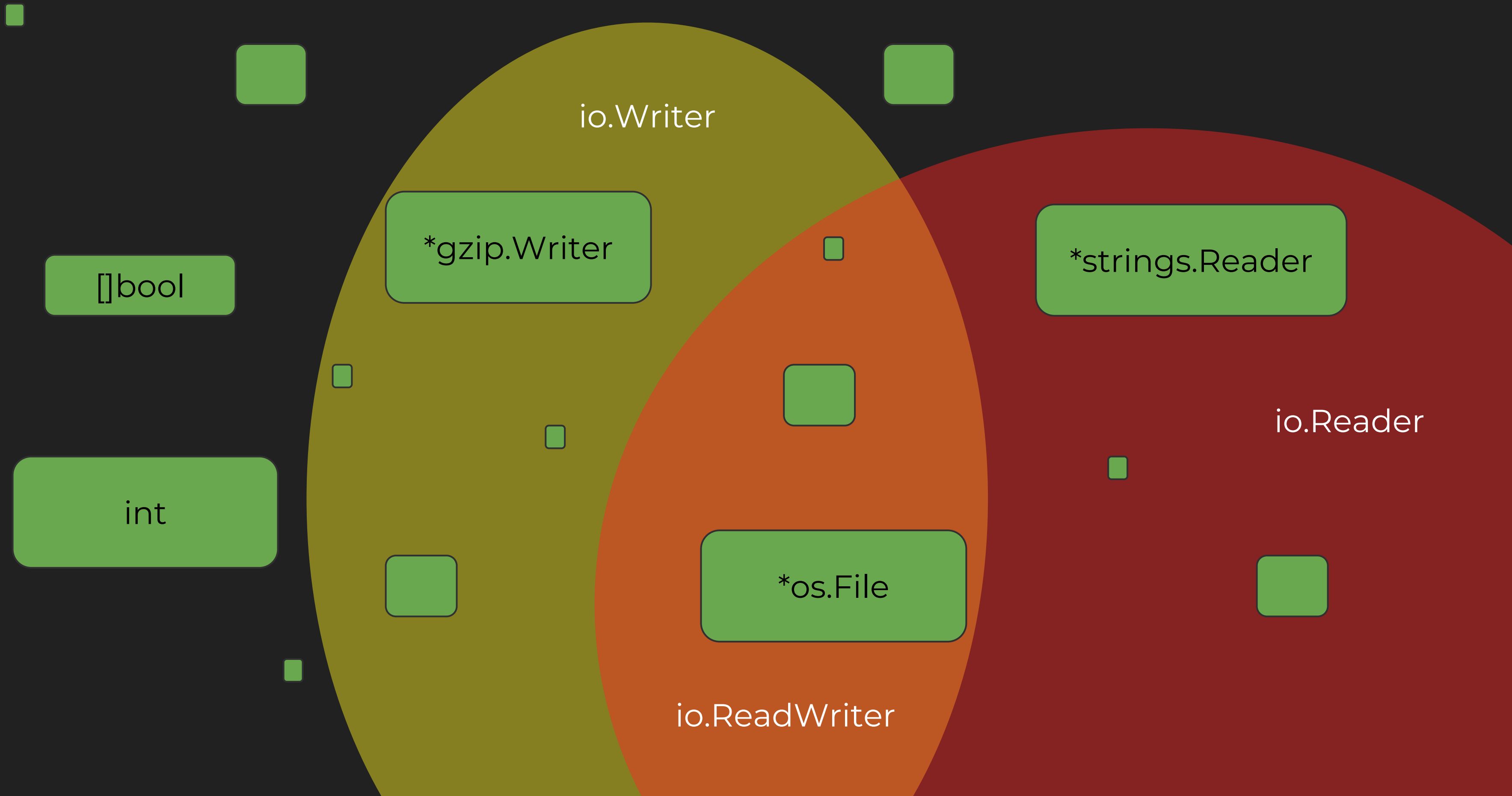

io.Writer
*gzip.Writer
[]bool
*strings.Reader
io.Reader
int
*os.File

union of interfaces

```
type ReadWriter interface {
    Read(b []byte) (int, error)
    Write(b []byte) (int, error)
}
```

union of interfaces

```go
type ReadWriter interface {
    Reader
    Writer
}
```


io.Writer
*gzip.Writer
[]bool
*strings.Reader
io.Reader
int
*os.File
io.ReadWriter

?

io.Writer

*gzip.Writer

*strings.Reader

[]bool

io.Reader

int

*os.File

io.ReadWriter

interface{}

“interface{} says nothing”

- Rob Pike in his Go Proverbs

why do we use interfaces?

why do we use interfaces?

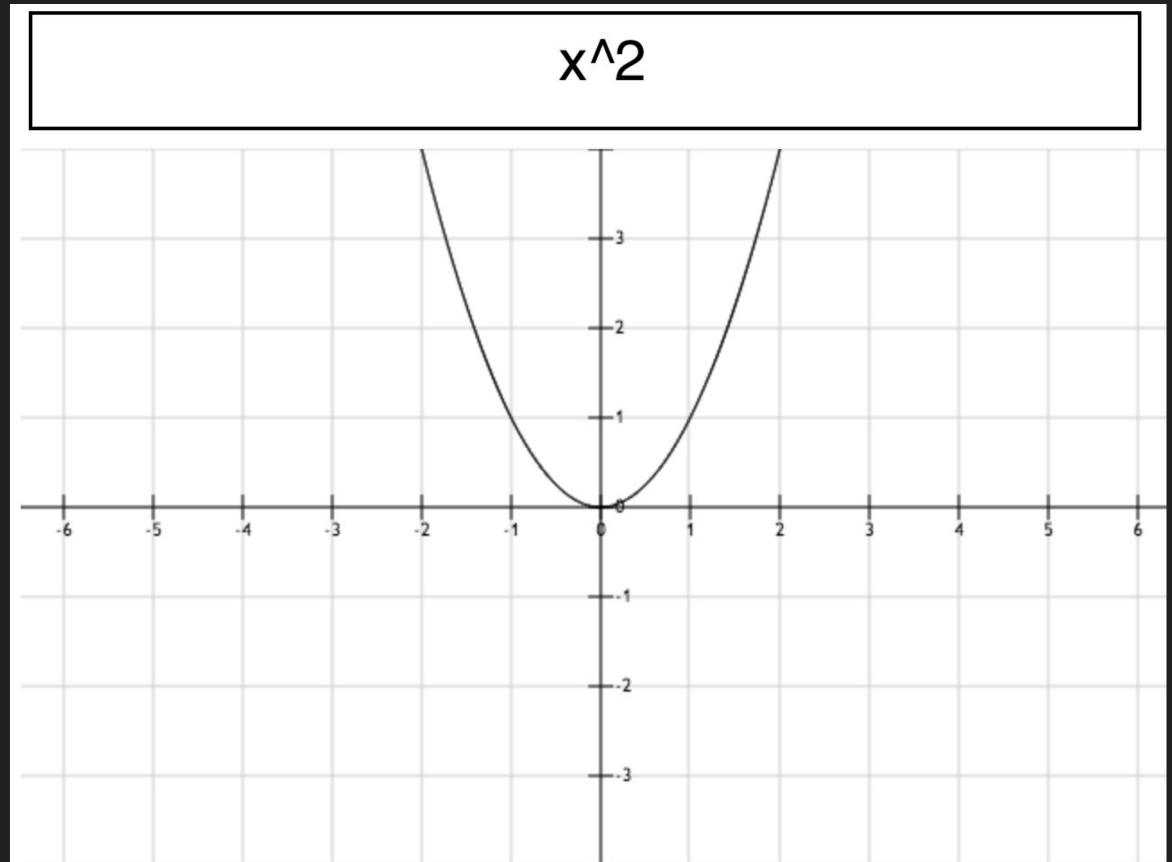
- writing generic algorithms
- hiding implementation details
- providing interception points

so … what’s new?

implicit interface satisfaction

no “implements”

funcdraw



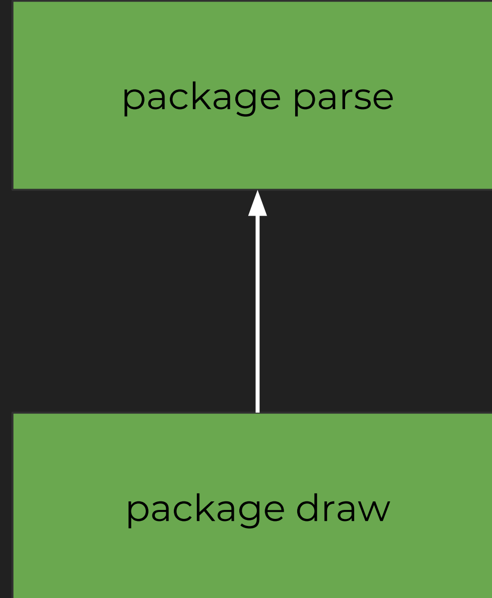
Two packages: parse and draw

```
package parse


func Parse(s string) *Func


type Func struct { … }

func (f *Func) Eval(x float64) float64
```

Two packages: parse and draw

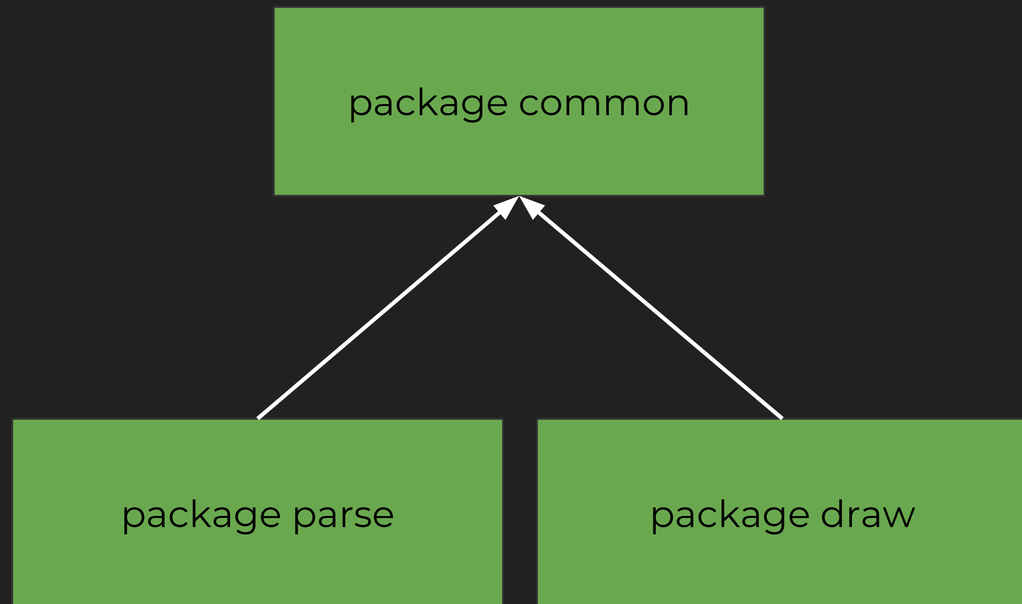
```
package draw

import “.../parse”

func Draw(f *parse.Func) image.Image {
    for x := minX; x < maxX; x += incX {
        paint(x, f.Eval(y))
    }
    …
}
```

funcdraw



funcdraw

with explicit satisfaction



funcdraw

with implicit satisfaction

package parse

package draw

Two packages: parse and draw

```
package draw

import “.../parse”

func Draw(f *parse.Func) image.Image {
    for x := minX; x < maxX; x += incX {
        paint(x, f.Eval(y))
    }
    …
}
```

Two packages: parse and draw

```
package draw

type Evaler interface { Eval(float64) float64 }

func Draw(e Evaler) image.Image {
    for x := minX; x < maxX; x += incX {
        paint(x, e.Eval(y))
    }
    …
}
```

interfaces can break dependencies

define interfaces where you use them

the super power of Go interfaces

type assertions

type assertions from interface to concrete type

```
func do(v interface{}) {
    i := v.(int)          // will panic if v is not int
    i, ok := v.(int)      // will return false
}
```

type assertions from interface to concrete type

```
func do(v interface{}) {

    switch v.(type) {

    case int:
        fmt.Println("got int %d", v)

    default:


    }
}
```

type assertions from interface to concrete type

```
func do(v interface{}) {

    switch t := v.(type) {

    case int:       // t is of type int
        fmt.Println("got int %d", t)

    default:        // t is of type interface{}
        fmt.Println("not sure what type")

    }
}
```

type assertions from interface to interface

```
func do(v interface{}) {
    s := v.(fmt.Stringer)       // might panic
    s, ok := v.(fmt.Stringer)  // might return false
}
```

type assertions from interface to interface

```
func do(v interface{}) {

    switch v.(type) {

    case fmt.Stringer:
        fmt.Println(“got Stringer %v”, v)

    default:


    }
}
```

type assertions from interface to interface

```
func do(v interface{}) {
    select s := v.(type) {
    case fmt.Stringer:        // s is of type fmt.Stringer
        fmt.Println(s.String())
    default:                  // s is of type interface{}
        fmt.Println(“not sure what type”)
    }
}
```

type assertions as extension mechanism

Many packages check whether a type satisfies an interface:

- fmt.Stringer : *implement* `String() string`
- json.Marshaler : *implement* `MarshalJSON() ([]byte, error)`
- json.Unmarshaler : *implement* `UnmarshalJSON([]byte) error`
- ...

and adapt their behavior accordingly.

Tip: Always look for exported interfaces in the standard library.

use type assertions to extend behaviors

Day 2

Concurrency FTW!

Agenda

- Live Coding
- ...
- Q&A

Live Coding Time!

Code

github.com/campoy/chat

- Includes Markov chain powered bot, which I skipped during live coding session.
- Feel free to send questions about it!

References:

Original talk by Andrew Gerrand: slides

Concurrency is not parallelism: blog

Go Concurrency Patterns: slides

Advanced Concurrency Patterns: blog

I came for the easy concurrency, I stayed for the easy composition: talk

Day 3

Agenda

- Debugging
- Testing and Benchmarks
- pprof & Flame Graphs
- Q&A

Debugging

- github.com/go-delve/delve
- Linux, macOS, Windows
- Written in Go, supports for goroutines
- Debugger backend and multiple frontends (CLI, VSCode, ...)

DELVE

A Debugger for the Go Programming Language

Debugging Live Demo!

code

Testing

Code sample:

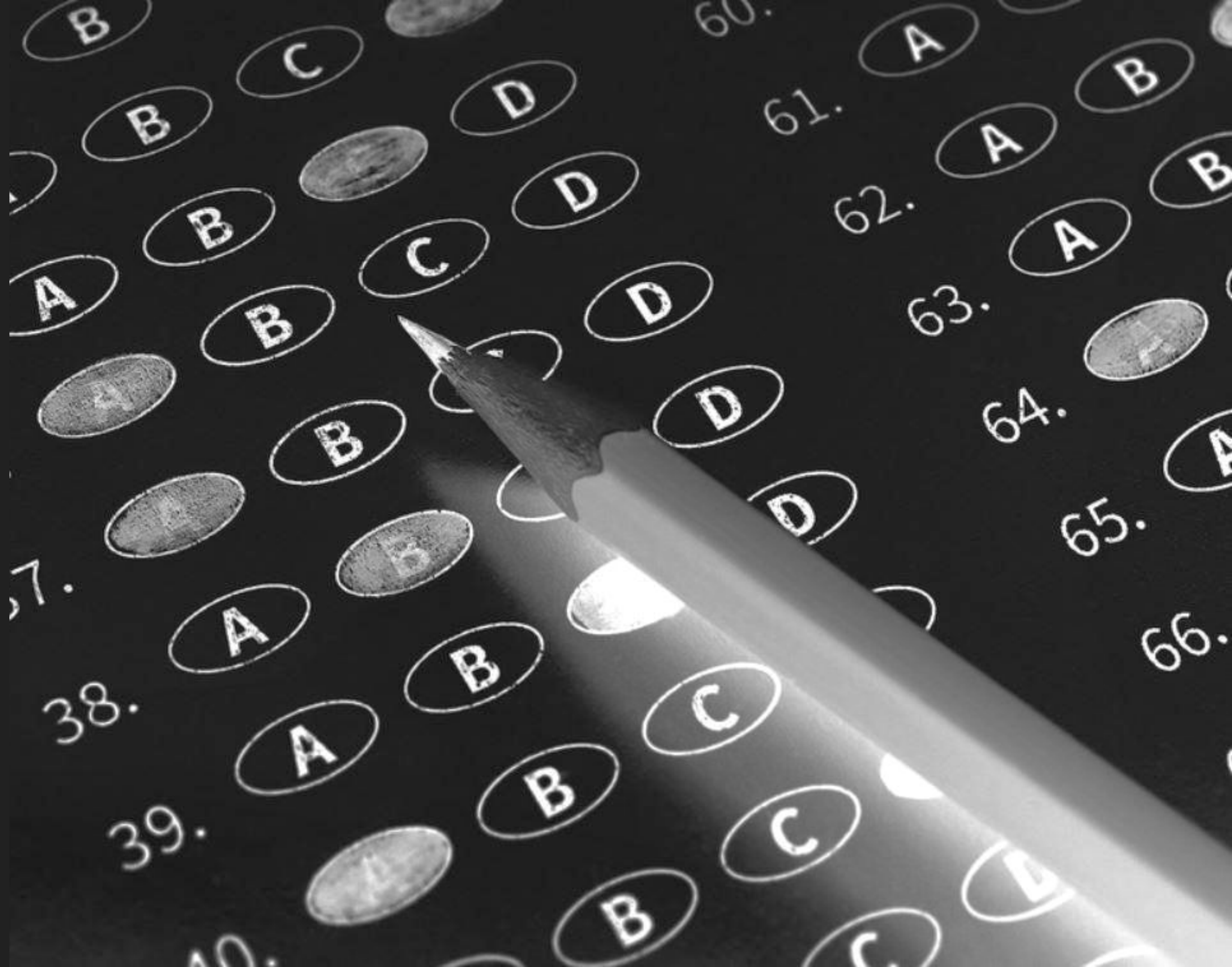
```
import “testing”

func TestFoo(t *testing.T) { … }
```

$ go test

Marking failure: t.Error, t.Errorf, t.Fatal, t.Fatalf

Table Driven Testing - Subtests: t.Run

Testing Live Demo!

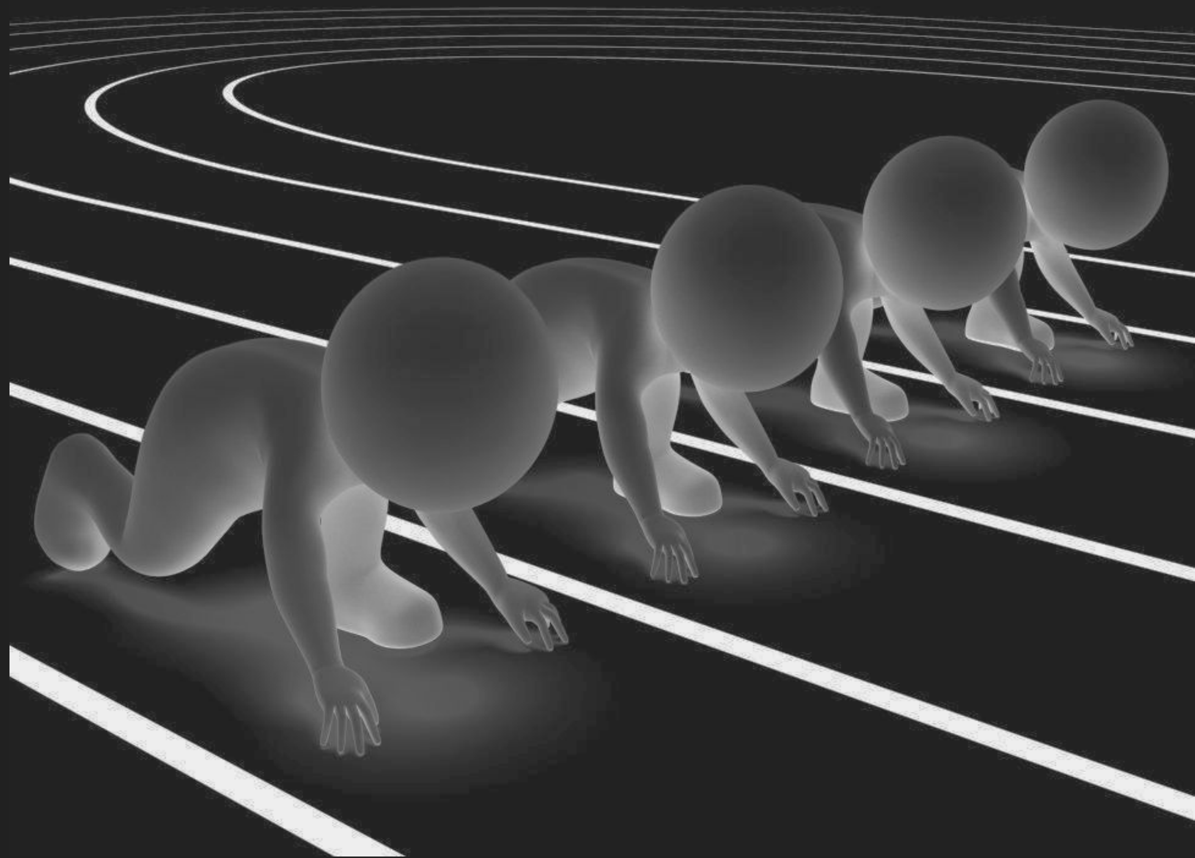


Benchmarks

Code sample:

```
import “testing’

func BenchmarkFoo(b *testing.B) {
    for i := 0; i < b.N; i++ {
        // do some stuff
    }
}
```

```
$ go test -bench=.
```



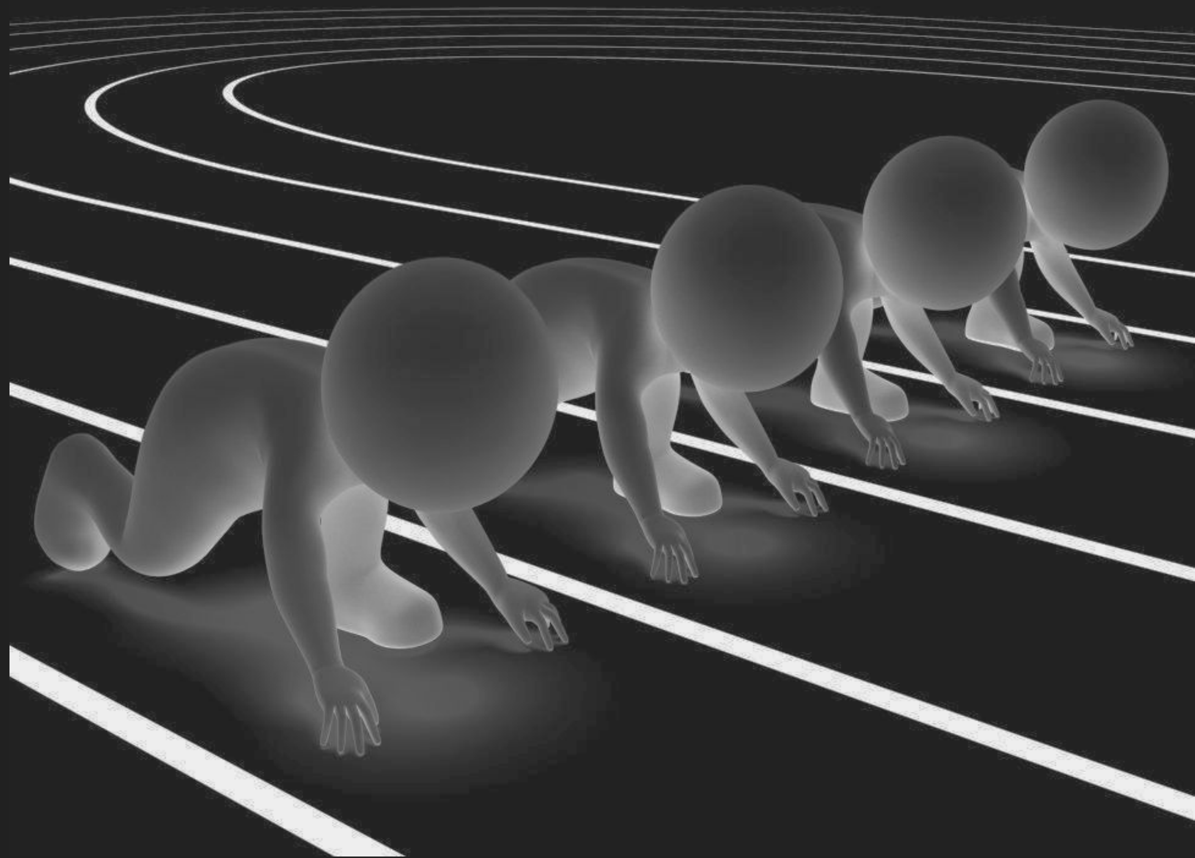
Benchmarking Live Demo!

pprof

- go get github.com/google/pprof

```
$ go test -bench=.    -cpuprofile=cpu.pb.gz
                      -memprofile=mem.pb.gz

$ pprof -http=:$PORT profile.pb.gz
```

Checks what the program is up *very* regularly, then provides statistics.



Benchmarking Live Demo!

pprof for web servers

```
import _ net/http/pprof
```

Web servers ... and anything else!

$ pprof -seconds 5 http://localhost:8080/debug/pprof/profile

Notes:

- Requires traffic (github.com/tsliwowicz/go-wrk)
- No overhead when off, small overhead when profiling.

Benchmarking Live Demo!

References

Go Tooling in Action: video

Go Tooling Workshop: github.com/campoy/go-tooling-workshop

- Compilation, cgo, advanced build modes.
- Code coverage.
- Runtime Tracer.
- Much more!

justforfunc #22: Using the Go Execution Tracer: video

Questions and Answers

Go for scientific computation?

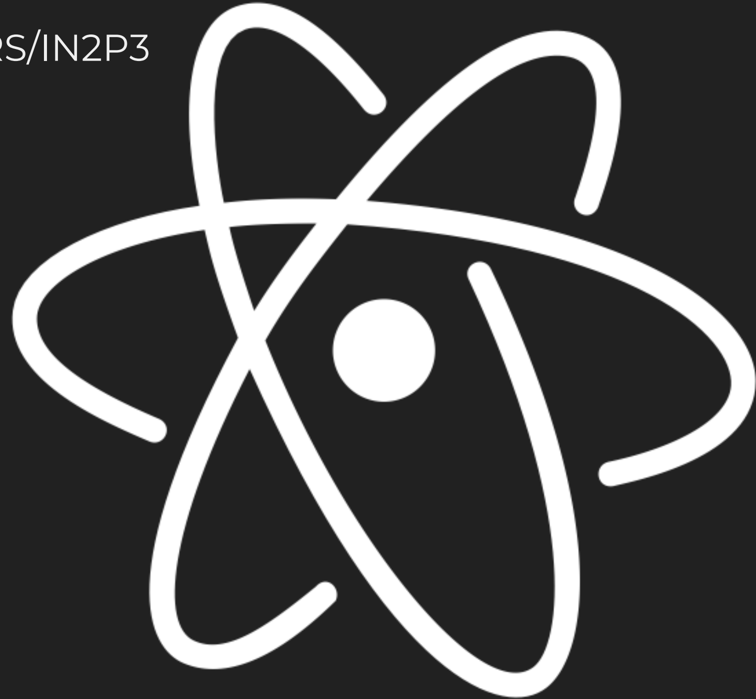
- gonum.org/v1/gonum
  - Similar to numpy
  - I love using it, really fast!

- knire-n/gota
  - Similar to Pandas
  - Never used it, but I heard good things

Go for scientific computation?

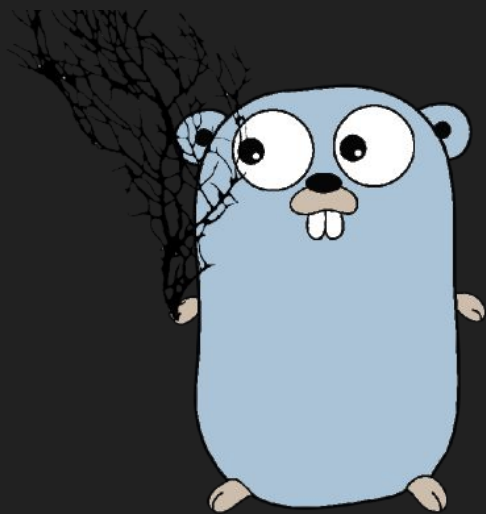
- go-hep.org/x/hep
  - High Energy Physicis
  - By Sebastien Binet – Research engineer @CNRS/IN2P3

So … Gophers @ CERN?



Go for Machine Learning?

- github.com/tensorflow/tensorflow/tensorflow/go
  - Rumour says, if you say it out loud Jeff Dean will appear.
  - Bindings for Go, only for serving – training not supported yet.
- gorgonia.org/gorgonia
  - Similar to Theano or Tensorflow, but in Go
  - I love the package, but the docs need some love.



Configuring Go programs?

- github.com/kelseyhightower/envconfig
  - Straight forward but pretty powerful.
- github.com/spf13/viper
  - Very complete and many people use it.
- github.com/spf13/cobra
  - Great to define CLIs, works with viper
- Reading json, csv, xml, ...
  - encoding/json, encoding/csv, encoding/xml
  - Find more on godoc.org

Go vs Python

Go for Pythonistas: [talk]

Pros:

- Speed
- Statically typed
- Less *magic*

Cons:

- Less *magic*
- Rigidity of type system (Tensorflow)

Now it’s your time!

Thanks!

Thanks, @francesc

campoy@golang.org