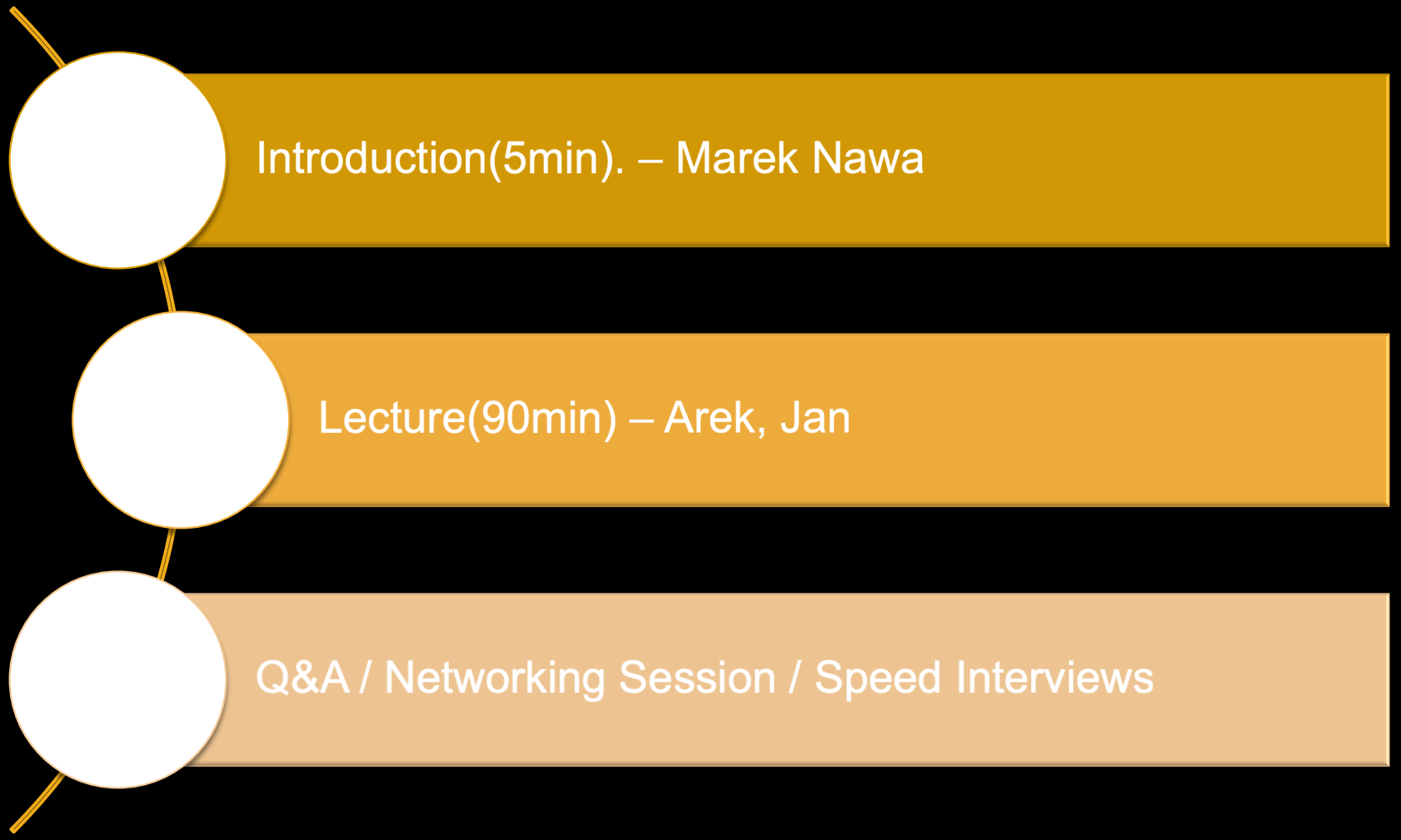
Jak to robią goffery, czyli wprowadzenie do języka Go

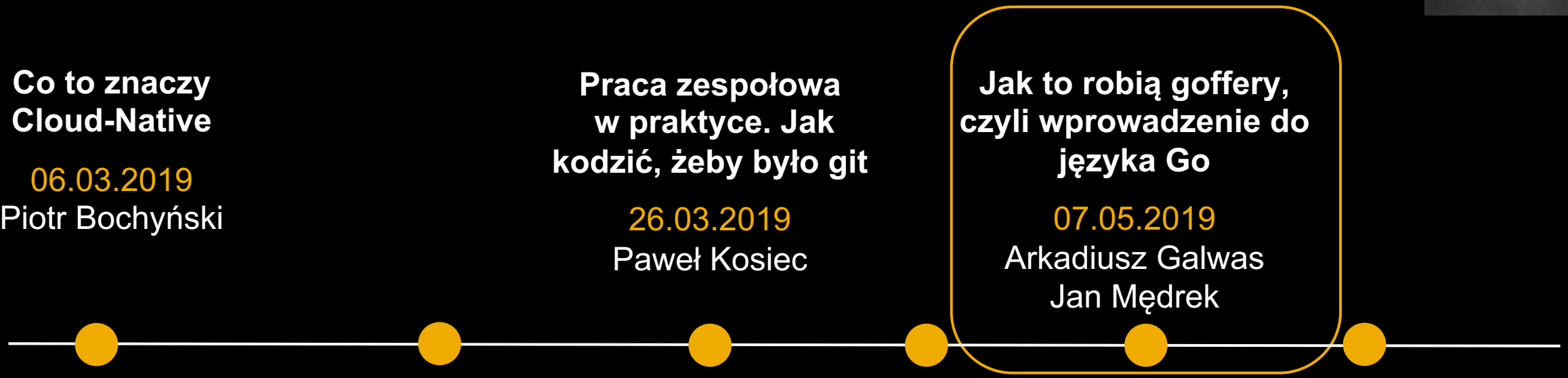
Arkadiusz Galwas, SAP
Jan Mędrek, SAP
May 7, 2019

PUBLIC

Agenda



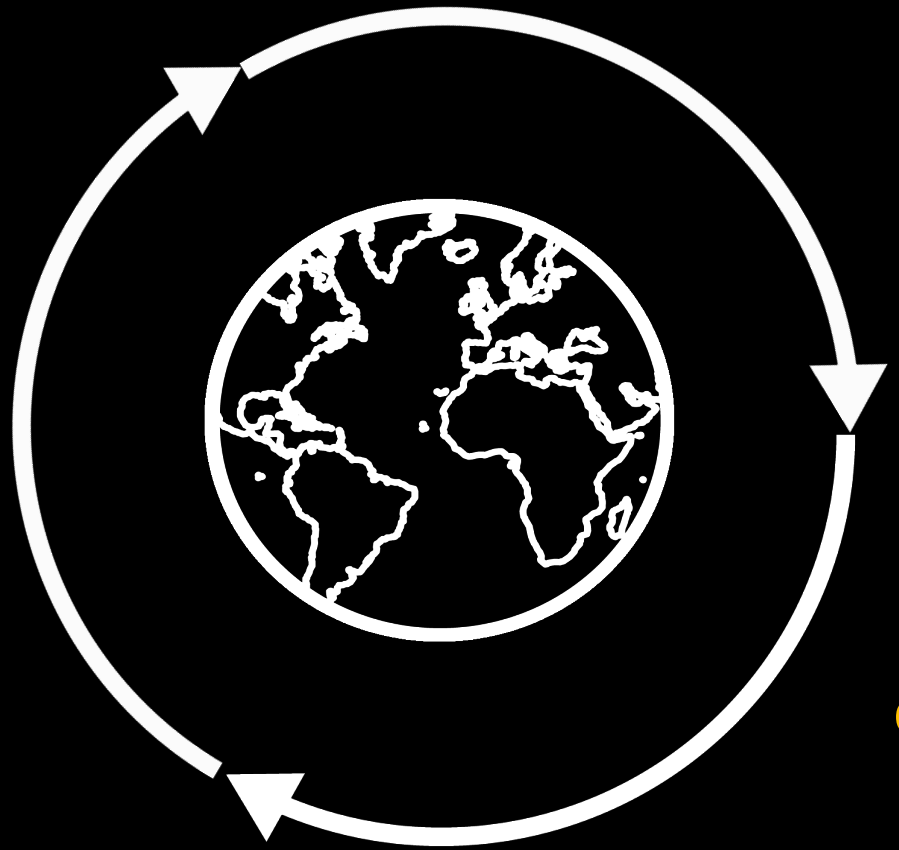
https://hackyourcareer.github.io



SAP Labs Poland

HAYO CKUR CAREER

Top ecommerce, marketing, billing



> 400 employees

Development: Go, Java, JavaScript, Cloud Native solutions

One of the 20 **innovation SAP development centers**

Who are we looking for?

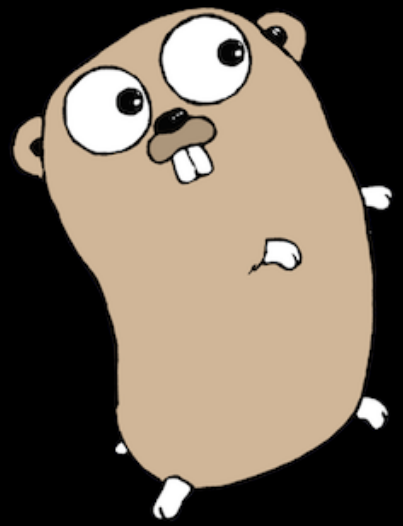
- Java, Go, React Developers
  - Quality Engineers
    - Support Engineers
      - Scrum Masters
        - Demo Solutions Specialists

https://jobs.sap.com/
Location - Gliwice

Kahoot!

HAYO CKUR CAREER

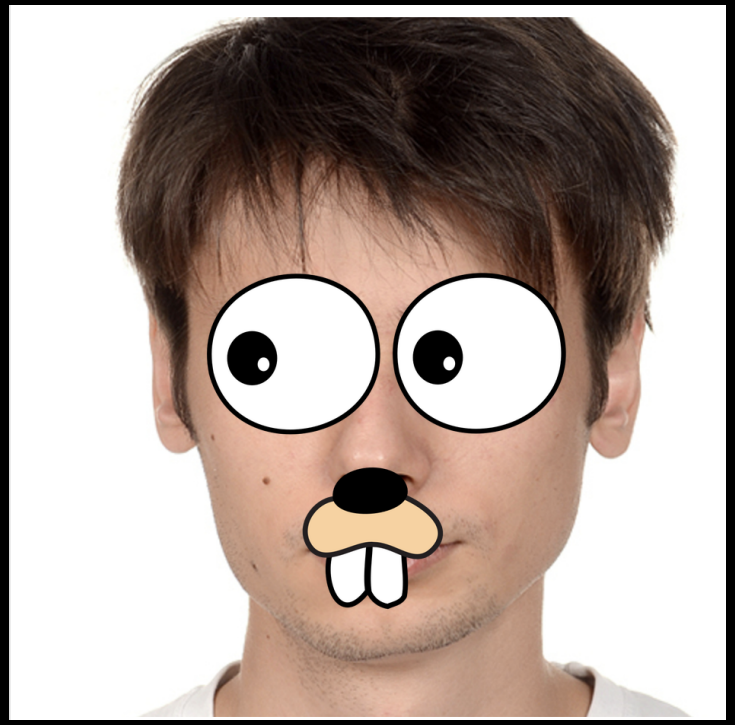
Jak to robią goffery, czyli wprowadzenie do języka Go



HAYO CKUR CAREER

Arkadiusz Galwas

Jan Mędrek



- What is Go?
- Gophers habitat
- Gopherised workbench
- Go minimal!
- Testing in Go
- Designing Programs in Go

What is Go?

What is Go?

- Statically typed
- Compiled
- Opinionated
- Suited for modern needs

“Go’s purpose is therefore not to do research into programming language design;
it is to improve the working environment for its designers and their coworkers.
Go is more about software engineering than programming language research.
Or to rephrase, it is about language design in the service of software engineering.”

—Rob Pike, Go’s creator

What is Go?

Go

Golang

What’s up with the gophers?

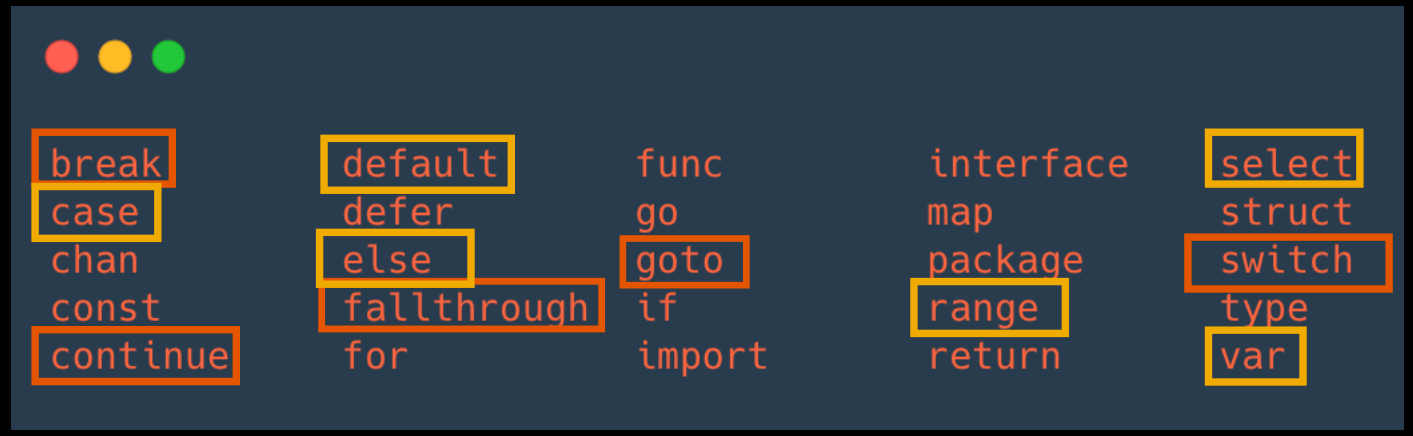
What is Go?

- Go was born out of frustration
- Ancestry:
  - C
  - Pascal / Oberon / Modula
  - Newsqueak, Limbo

Design focus

- Improve coding process
- Easily scalable
- Reduction of complexity level
- Tooling that is easy to use
- Easy to learn

Keywords



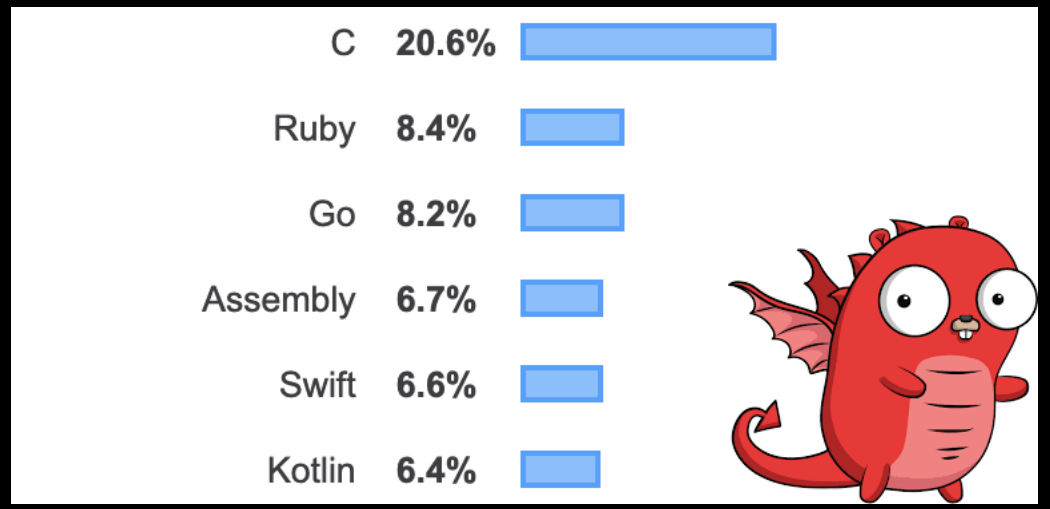
Simplicity

```go
package main

import "fmt"

func add(a int, b int) int {
    return a + b
}

func main() {
    var res int
    res = add(1, 2)

    res2 := add(1,2)

    fmt.Println(res, res2)
}
```

```go
package main

import "fmt"

func main() {

    i := 1
    for i <= 3 {
        fmt.Println(i)
        i = i + 1
    }

    for j := 7; j <= 9; j++ {
        fmt.Println(j)
    }

    for {
        fmt.Println("loop")
        break
    }
}
```

Where Go is being used

- Google
- Docker
- Allegro
- OLX
- SAP



Gophers habitat

GOPATH

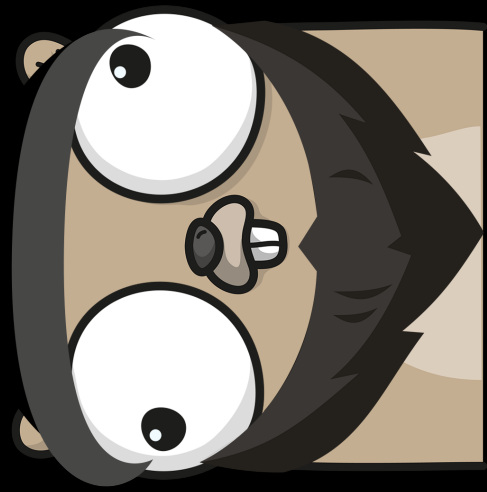
- Forces to define root directory
- Single workspace
- Imports are relative to `GOPATH/src`

Runtime

- No VM
- Runtime is a library
  - Garbage collection
  - Concurrency
  - Stack management
  - Other critical features

Garbage collector

- No explicit memory-freeing operation
- Algorithm from decades ago (tuned up)
- GC knob



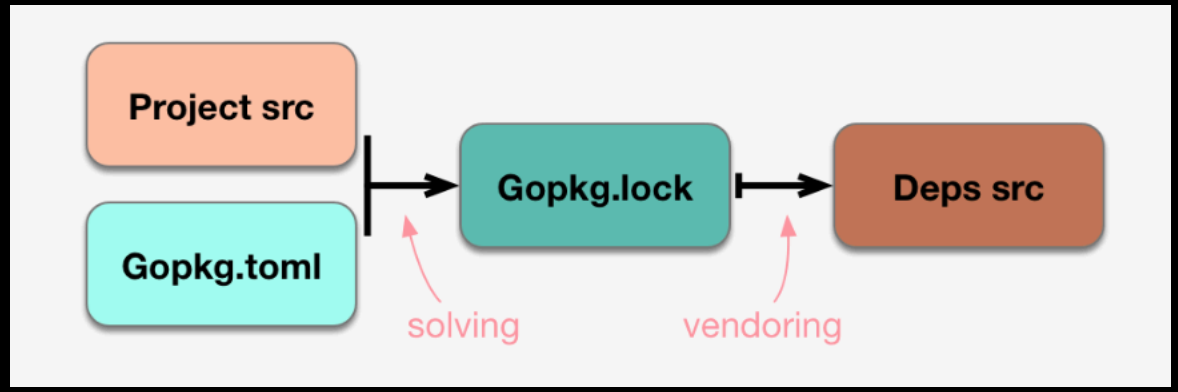
Dependency management

- Projects organised in packages
- Convention-driven exports
- Import only once
- Vendoring

Gopherised workbench

Dep

- Community-driven
- Four-state system
  - Source code
  - Manifest file
  - Lock file
  - Dependencies
- Super-simple usage



GoDoc

- Documentation generator
- Parses source code
- Concept:
  - Docstring
  - Javadoc

GoDoc

```
// Package sort provides primitives for sorting slices and user-defined
// collections.
package sort

// Float64s sorts a slice of float64s in increasing order
// (not-a-number values are treated as less than other values).
func Float64s(a []float64) { Sort(Float64Slice(a)) }
```

https://godoc.org/sort

Formatting, linting, …

gofmt

- Code formatting
- 70% of all Go packages is formatted with gofmt
- Uncontroversial

golint

- Code styling
- Makes **suggestions**

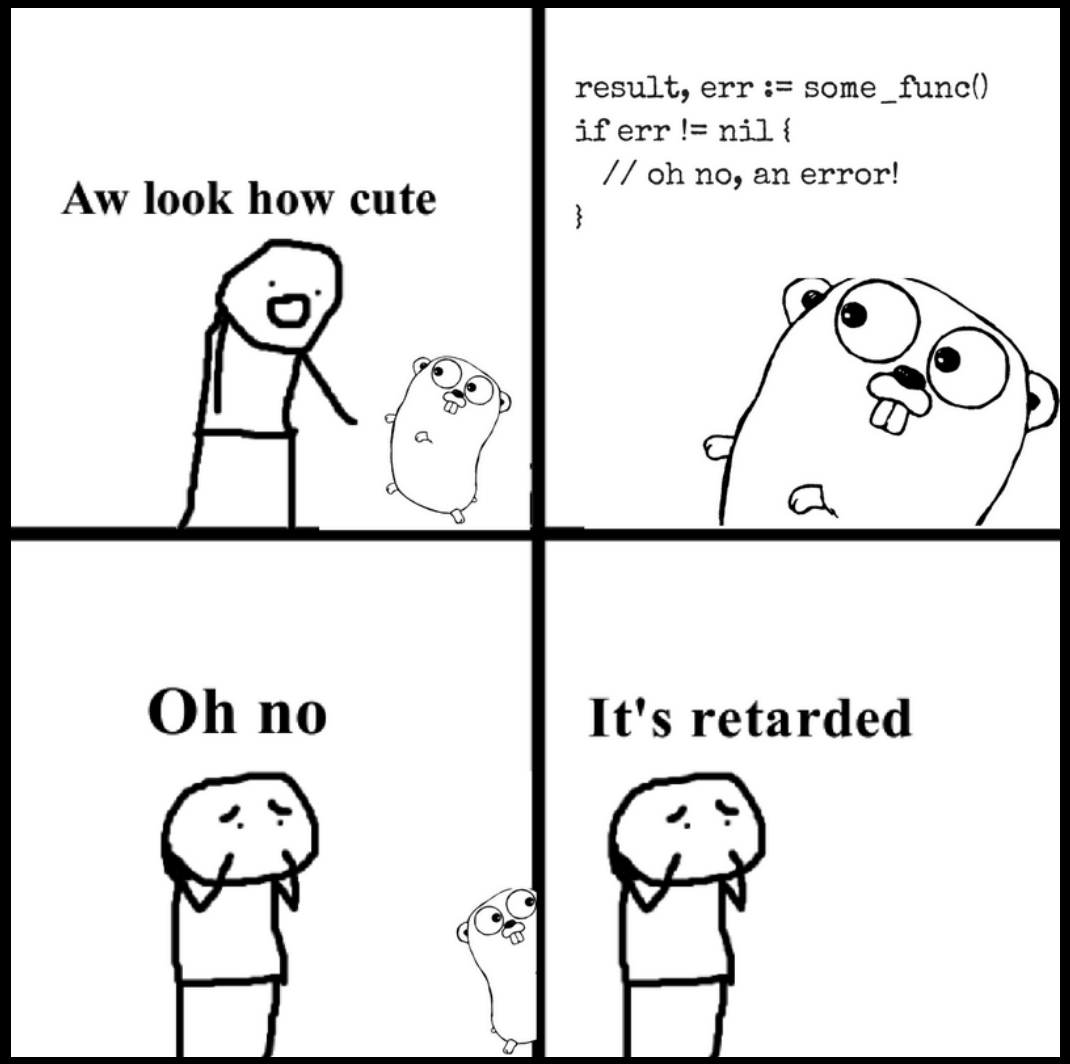
Why would I bother?

Go minimal!

Go minimal!

- Limited set of features
- Pragmatic design
- No inheritance

Errors in Go



Errors in Go

- Error as an interface type
- Multi-value returns
- Built-in functions (for extreme cases)

Errors in Go

```go
func (th *tokenHandler) CreateToken(w http.ResponseWriter, r *http.Request) {
    clientContextService, err := th.connectorClientExtractor(r.Context())
    if err != nil {
        logger.Error(err)
        httphelpers.RespondWithError(w, err)
        return
    }

    ...

    token, err := th.tokenManager.Save(clientContextService.ClientContext())
    if err != nil {
        logger.Error(err)
        httphelpers.RespondWithError(w, err)
        return
    }

    ...
}
```

Errors in Go

- No exceptions
- User – friendly
- Extendable

Generics and generators

“Generics may well be added at some point. We don't feel an urgency for them, although we understand some programmers do.”

- Left our for the simplicity
- go-generate
  - generics-like functionality

Testing in Go

Testing package

- Supports automated testing of packages
- Used with „go test” command
- Built-in coverage
- Benchmarking
- Conventions

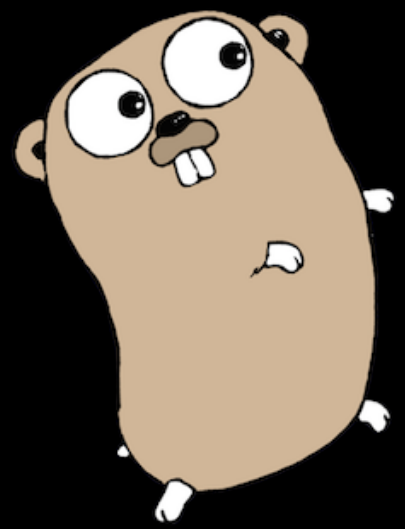
Testing package

```
func TestOptions_ParseDuration(t *testing.T) {
    t.Run("should parse proper duration string", func(t *testing.T) {
        //given
        durationString := "30h"

        //when
        res, err := parseDuration(durationString)

        //then
        assert.NoError(t, err)
        assert.Equal(t, time.Duration(30)*time.Hour, res)
    })

    ...
}
```

Race detector

- Races can cause failures
- Ran via „-race” flag
- Runtime lib watches for unsynchronised accesses to variables

Designing Go programs



Designing Go programs

- Object Oriented Programming
- Functional Programming
- Summary

Object Oriented Programming

Object Oriented Programming

Object-oriented programming (OOP) is a programming paradigm based on the concept of objects, which can contain data, in the form of fields (often known as attributes), and code, in the form of procedures (often known as methods).(…) In OOP, computer programs are designed by making them out of objects that interact with one another.

https://en.wikipedia.org/wiki/Object-oriented_programming

Object Oriented Programming

Key mechanism:

- Encapsulation
- Relationships:
  - Inheritance
  - Composition
- Polymorphism

Object Oriented Programming in Go

Is Go an object-oriented language?

Yes and no. Although Go has types and methods and allows an object-oriented style of programming, there is no type hierarchy. The concept of “interface” in Go provides a different approach that we believe is easy to use and in some ways more general. (…)

Also, the lack of a type hierarchy makes “objects” in Go feel much more lightweight than in languages such as C++ or Java.

https://golang.org/doc/faq#Is_Go_an_object-oriented_language

Object Oriented Programming in Go – structs and method receivers

```go
type Person struct {
    Name    string
    Surname string
}

func (person Person) SayHello() {
    fmt.Printf(format:"Hello I'm %s %s \n", person.Name, person.Surname)
}

func main() {
    empty := Person{}
    empty.SayHello() // Hello I'm

    ag := Person{Name:"Arek", Surname:"Galwas"}
    ag.SayHello() // Hello I'm Arek Galwas
}
```

Object Oriented Programming in Go – structs and method receivers

Method receivers can be used with any type!

```
type MyInt int

func (i MyInt) IsEven() bool {
    return i % 2 == 0
}

func main() {
    var number MyInt = 10

    fmt.Printf(format:"%d is even = %t \n", number, number.IsEven())
}
```

Object Oriented Programming in Go – encapsulation

An identifier may be exported to permit access to it from another package. An identifier is exported if both:

1. *the first character of the identifier's name is a Unicode upper case letter (Unicode class "Lu"); and*
2. *the identifier is declared in the package block or it is a field name or method name.*

All other identifiers are not exported.

https://golang.org/ref/spec#Exported_identifiers

Object Oriented Programming in Go – composition by type embedding

```go
type Engine struct {
    Capacity int
    Type     string
}

type Car struct {
    Engine
    Brand  string
    Model  string
}

func main() {
    car := Car{Engine: Engine{Capacity: 1995, Type: "Diesel"}, Brand: "BMW", Model: "G20"}

    // Longer syntax for accessing embedded fields
    fmt.Printf(format: "Brand: %s, Engine capacity:%d \n", car.Brand, car.Engine.Capacity)

    // Syntactic sugar making use of type promotion
    fmt.Printf(format: "Brand: %s, Engine capacity:%d \n", car.Brand, car.Capacity)

    // Type embedding is NOT inheritance
    var engine Engine = car
}
```

Object Oriented Programming in Go – composition by type embedding and method receivers

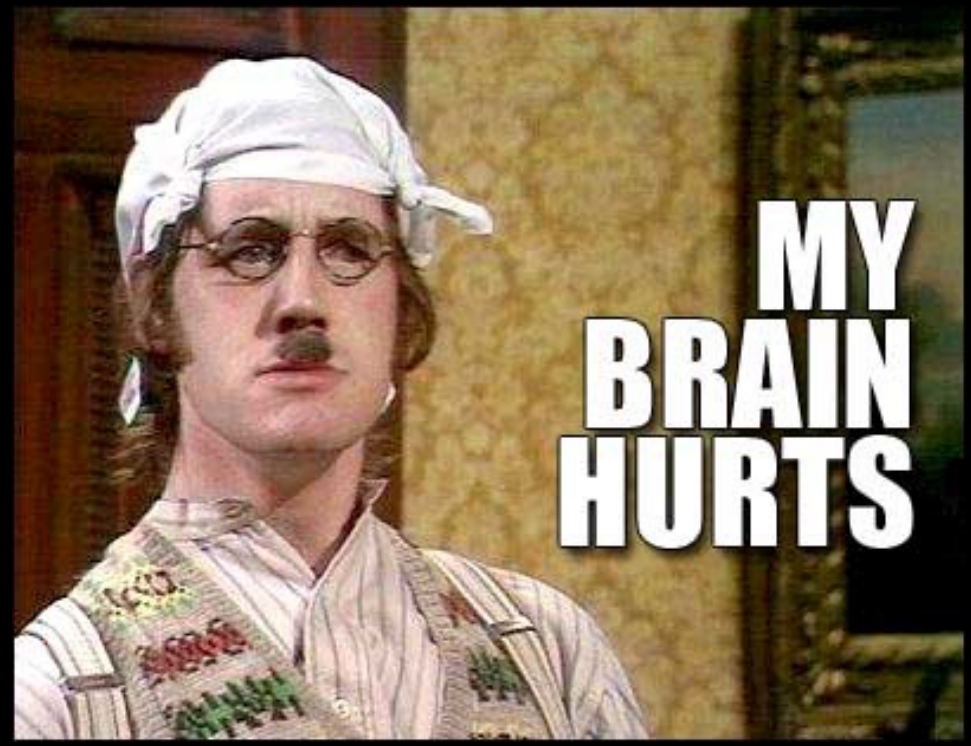
```go
type Engine struct {...}

type Car struct {
    Engine
    Brand  string
    Model  string
}

func (e Engine) IsEcoFriendly() bool{
    return e.Type != "Diesel" && e.Type != "Petrol"
}

func (c Car) IsPremium() bool {
    return c.Brand == "Rolls-Royce" || c.Brand == "Ferrari"
}

func main() {
    car := Car{Engine: Engine{Capacity: 1995, Type: "Diesel"}, Brand: "BMW", Model: "G20"}

    fmt.Sprintf( format: "Is premium: %t, is eco-friendly:%t \n", car.IsPremium(), car.IsEcoFriendly())
}
```

Object Oriented Programming in Go – polymorphism

Interface is the basic tool for writing polymorphic code.

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

Important note: there is ***NO*** *implements* keyword.



Object Oriented Programming in Go – polymorphism

If there is no such thing like *implements keyword*, how do I (and the compiler) know that struct implements an interface?

Interfaces Duck Typing is the answer!

If it walks like a duck and it quacks like a duck, then it must be a duck.

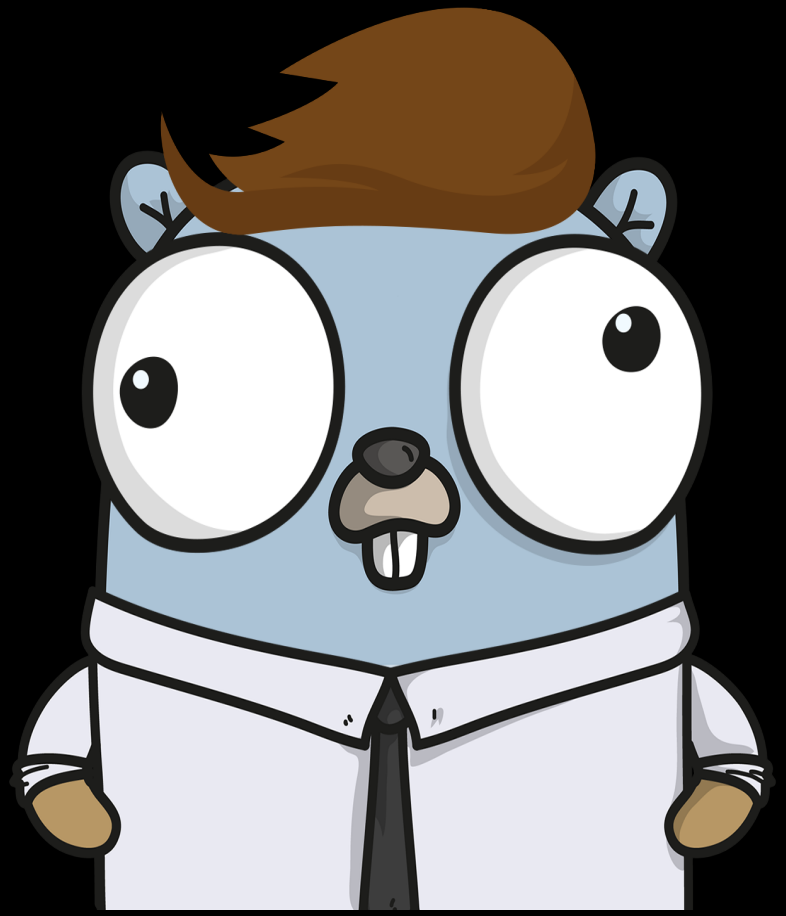
Object Oriented Programming in Go – polymorphism and composition

Interfaces can be composed.

```
type ReadWriteCloser interface {
    Reader
    Writer
    Closer
}
```

Object Oriented Programming in Go

Let’s see some real-life code!



Functional Programming

Functional Programming

In computer science, functional programming is a programming paradigm (…) that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions or declarations[1] instead of statements.

Functional code is idempotent, the output value of a function depends only on the arguments that are passed to the function, so calling a function f twice with the same value for an argument x produces the same result f(x) each time;

https://en.wikipedia.org/wiki/Functional_programming

Functional Programming

Key mechanisms:

- First Class Functions
- Closures
- Recursion and internal iteration

Functional languages usually have sophisticated type system and support generics.

Functional Programming in Go

Go supports the following:

- First Class Functions
- Closures
- Anonymous functions

Functional Programming in Go – First Class Functions

Function is a first class citizen.

```
type BinaryFunc func (a int, b int) int
type UnaryFunc func (a int) int

func main() {
    var add BinaryFunc = func (a int, b int) int{
        return a + b
    }

    var neg UnaryFunc = func(a int) int {
        return -1 * a
    }

    res := neg(add( a: 1,  b: 2))
    fmt.Printf( format: "neg(add(1,2))=%d \n", res) // neg(add(1,2)=-3)
}
```

Functional Programming in Go – First Class Functions

```go
type BinaryFunc func (a int, b int) int
type UnaryFunc func (a int) int

// It is a Higher Order Function
func combine(f BinaryFunc, g UnaryFunc) BinaryFunc{
    return func(a int, b int) int{
        return g(f(a,b))
    }
}

func main() {
    add := func (a int, b int) int{
        return a + b
    }
    neg := func(a int) int {
        return -1 * a
    }
    combined := combine(add, neg)

    fmt.Printf( format: "neg(add(1,2))=%d \n", combined( a: 1, b: 2)) // neg(add(1,2)=-3)
}
```

Functional Programming in Go – First Class Functions and method receivers

```go
// Defined in net/http/server.go
type HandlerFunc func(ResponseWriter, *Request)

// ServeHTTP calls f(w, r).
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    f(w, r)
}
```

```go
func main() {
    handler := http.HandlerFunc(func (w http.ResponseWriter, r *http.Request){
        w.WriteHeader(http.StatusOK)
    })

    http.ListenAndServe( addr: ":8080", handler)
}
```

Functional Programming in Go

Anonymous functions are useful with Higher Order Functions

```
func main() {
    s := strings.Map(func (r rune) rune {
        return unicode.ToUpper(r)
    }, s:"hello Golang")

    fmt.Printf( format:"upper cased: %s \n", s) // upper cased: HELLO GOLANG
}
```

Functional Programming in Go

Let’s see some functional Go!

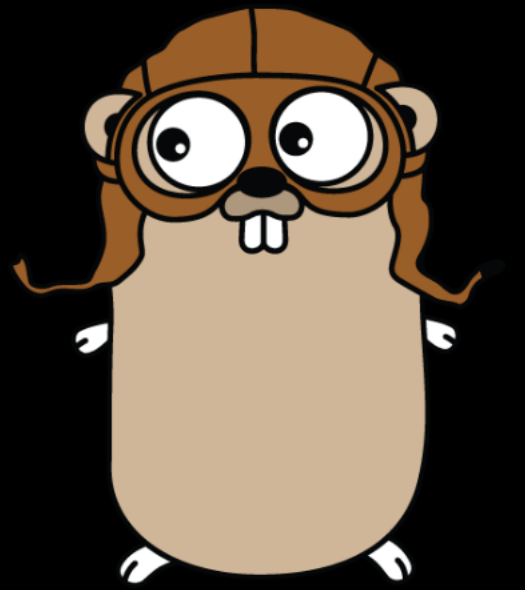
$$f(x)$$

Summary

- Go is a multiparadigm programming language so that I can pick up the right tool to the job.
- The lack of inheritance is a nice feature.
- Interfaces give great deal of flexibility and allow loose coupling.
- Functions can be used in different ways: as OOP style methods or FP style pure functions.
- I find Higher Order Function pattern very useful and easy to implement in Go.

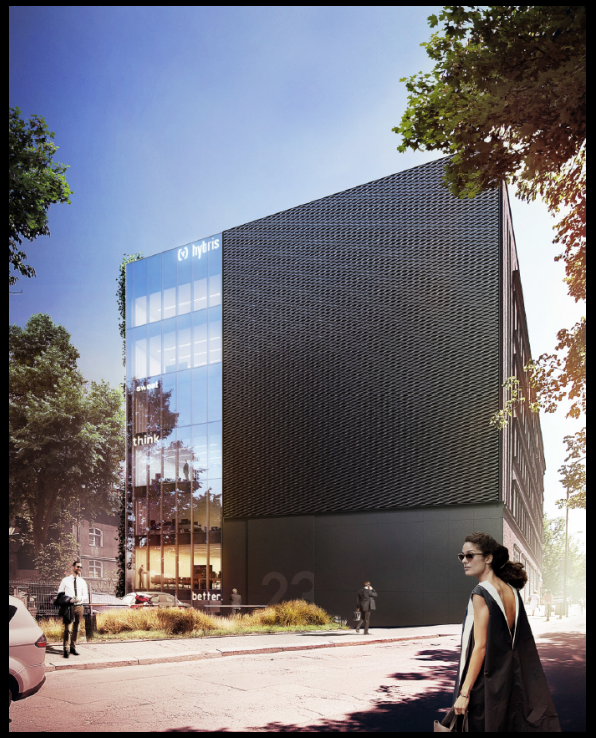
Questions?

Thank you!



Opcja 1

- Networking Session



Opcja 2

- Q&A with presenters

Opcja 3

- Speed Interviews

Thank you.

THE BEST RUN SAP