Kyiv September Go Meetup, Sep 28 2018

Go2 drafts: errors and generics

Ivan Danyliuk

Status.im

@idanyliuk

Go2 announcement

- Announced at GopherCon 2017
- Towards Go 2 blog post

Russ Cox: Towards Go 2

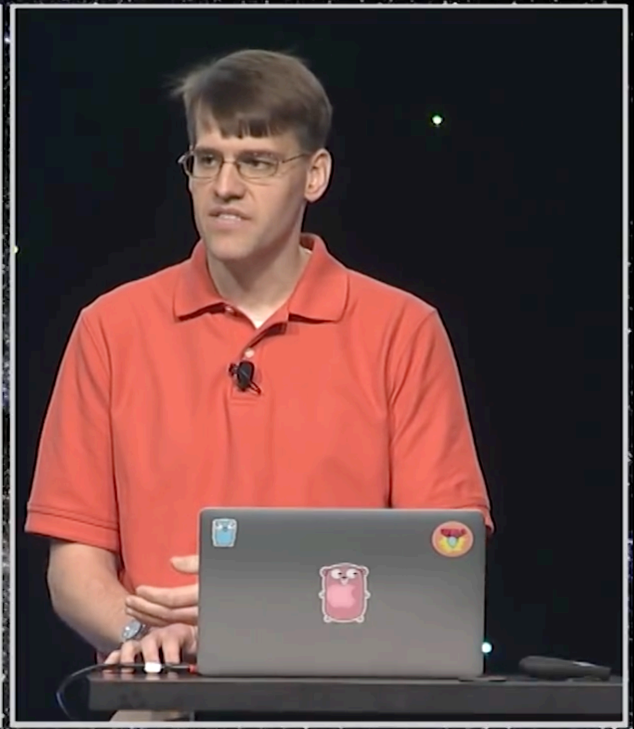
1 Use Go. (Accumulate experience.)

2 Identify and explain a problem.

3 Propose and discuss a solution.

4 Implement, evaluate, refine solution.

5 Ship production implementation.



https://www.youtube.com/watch?v=0Zbh_vmAKvk

Go2 announcement

- At GopherCon 2018 announced first drafts
- Error handling, errors wrapping and generics

1 Use Go. (Accumulate experience.)

2 Identify and explain a problem.

← *error handling & generics*

3 Propose and discuss a solution.

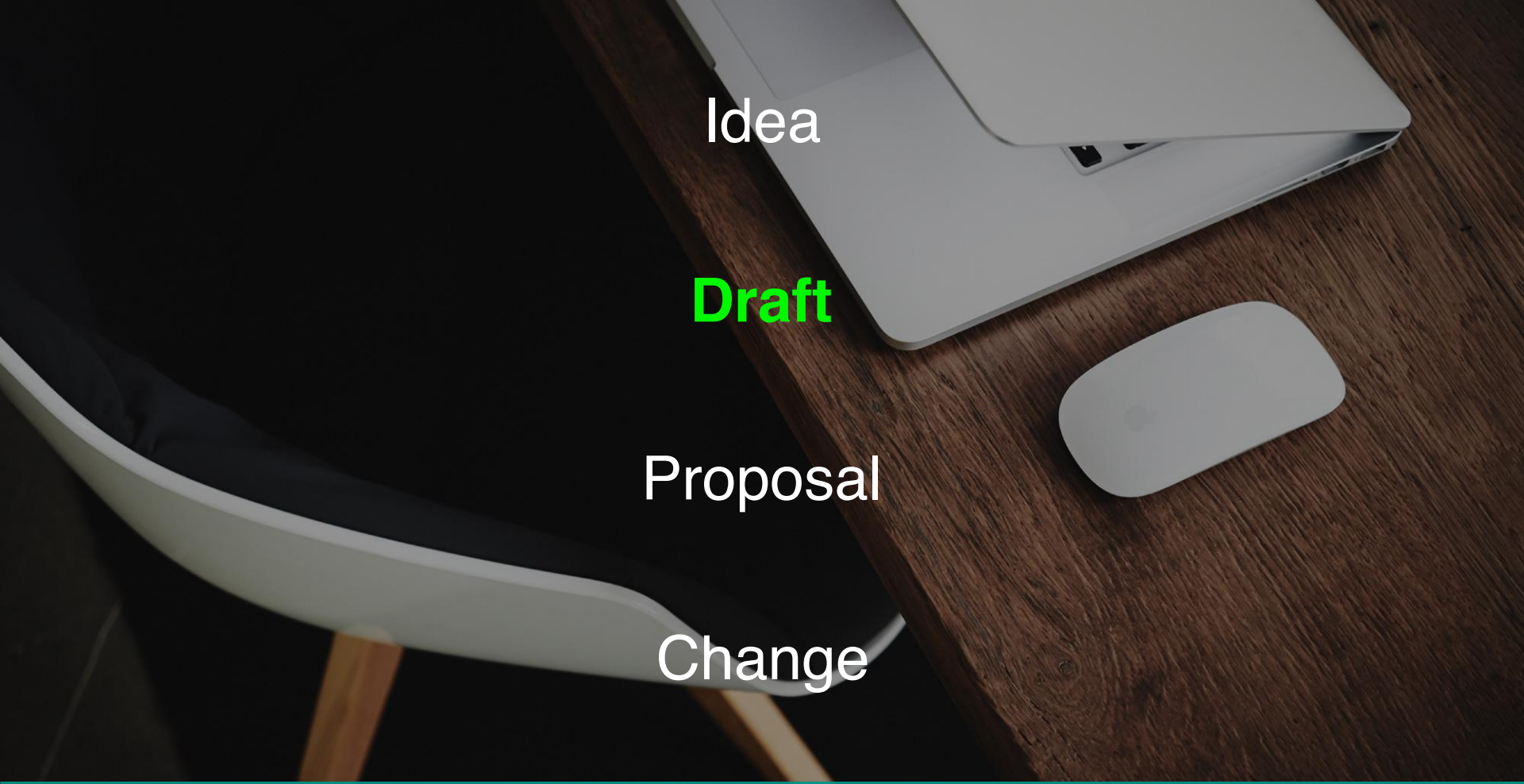
4 Implement, evaluate, refine solution.

← *modules*

5 Ship production implementation.

https://www.youtube.com/watch?v=6wIP3rO6On8

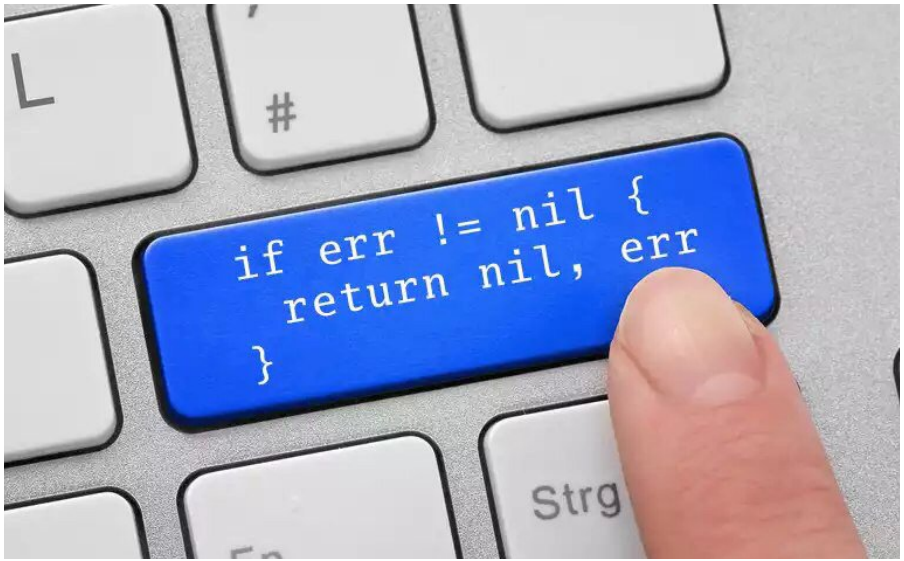
Draft is not even a proposal


Idea
Draft
Proposal
Change

Error Handling Draft

The Problem

- "Too much error *checking*"
- "Not enough error *handling*"



The Problem

```
func CopyFile(src, dst string) throws error {
        r := os.Open(src)
        defer r.Close()

        w := os.Create(dst)
        io.Copy(w, r)
        w.Close()
}
```

The Problem

```
func CopyFile(src, dst string) throws error {
        r := os.Open(src)
        defer r.Close()

        w := os.Create(dst)
        io.Copy(w, r)
        w.Close()
}
```

- This code is:
  - elegant

The Problem

```
func CopyFile(src, dst string) throws error {
        r := os.Open(src)
        defer r.Close()

        w := os.Create(dst)
        io.Copy(w, r)
        w.Close()
}
```

- This code is:
  - elegant
  - clean

The Problem

```
func CopyFile(src, dst string) throws error {
        r := os.Open(src)
        defer r.Close()

        w := os.Create(dst)
        io.Copy(w, r)
        w.Close()
}
```

- This code is:
  - elegant
  - clean
  - wrong

The Problem

```
func CopyFile(src, dst string) throws error {
        r := os.Open(src)
        defer r.Close()

        w := os.Create(dst)
        io.Copy(w, r)
        w.Close()
}
```

- This code is:
  - elegant
  - clean
  - wrong
- If w.Copy() or w.Close() fails, file is not removed

The Problem

```
func CopyFile(src, dst string) error {
        r, err := os.Open(src)
        if err != nil {
                return err
        }
        defer r.Close()
        w, err := os.Create(dst)
        if err != nil {
                return err
        }
        defer w.Close()
        if _, err := io.Copy(w, r); err != nil {
                return err
        }
        if err := w.Close(); err != nil {
                return err
        }
}
```

- Current approach:

The Problem

```
func CopyFile(src, dst string) error {
        r, err := os.Open(src)
        if err != nil {
                return err
        }
        defer r.Close()
        w, err := os.Create(dst)
        if err != nil {
                return err
        }
        defer w.Close()
        if _, err := io.Copy(w, r); err != nil {
                return err
        }
        if err := w.Close(); err != nil {
                return err
        }
}
```

- Current approach:
  - not clean

The Problem

```
func CopyFile(src, dst string) error {
        r, err := os.Open(src)
        if err != nil {
                return err
        }
        defer r.Close()
        w, err := os.Create(dst)
        if err != nil {
                return err
        }
        defer w.Close()
        if _, err := io.Copy(w, r); err != nil {
                return err
        }
        if err := w.Close(); err != nil {
                return err
        }
}
```

- Current approach:
  - not clean
  - not elegant

The Problem

```
func CopyFile(src, dst string) error {
        r, err := os.Open(src)
        if err != nil {
                return err
        }
        defer r.Close()
        w, err := os.Create(dst)
        if err != nil {
                return err
        }
        defer w.Close()
        if _, err := io.Copy(w, r); err != nil {
                return err
        }
        if err := w.Close(); err != nil {
                return err
        }
}
```

- Current approach:
  - not clean
  - not elegant
  - also wrong

The Problem

```
func CopyFile(src, dst string) error {
        r, err := os.Open(src)
        if err != nil {
                return err
        }
        defer r.Close()
        w, err := os.Create(dst)
        if err != nil {
                return err
        }
        defer w.Close()
        if _, err := io.Copy(w, r); err != nil {
                return err
        }
        if err := w.Close(); err != nil {
                return err
        }
}
```

- Current approach:
  - not clean
  - not elegant
  - also wrong
- The incorrect error check is more visible though

The Problem

```
func CopyFile(src, dst string) error {
        r, err := os.Open(src)
        if err != nil {
                return fmt.Errorf("copy %s %s: %v", src, dst, err)
        }
        defer r.Close()

        w, err := os.Create(dst)
        if err != nil {
                return fmt.Errorf("copy %s %s: %v", src, dst, err)
        }

        if _, err := io.Copy(w, r); err != nil {
                w.Close()
                os.Remove(dst)
                return fmt.Errorf("copy %s %s: %v", src, dst, err)
        }

        if err := w.Close(); err != nil {
                os.Remove(dst)
                return fmt.Errorf("copy %s %s: %v", src, dst, err)
        }
}
```

- Correct code

Goals

- More lightweight error checks
- Reduce amount of text for error handling
- Keep explicit checks
- Keep existing code working

Two syntax forms:

- check
- handle

Handle/check

```
func CopyFile(src, dst string) error {
    handle err {
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }

    r := check os.Open(src)
    defer r.Close()

    w := check os.Create(dst)
    handle err {
        w.Close()
        os.Remove(dst) // (only if a check fails)
    }

    check io.Copy(w, r)
    check w.Close()
    return nil
}
```

Handle/check

```
func CopyFile(src, dst string) error {
    handle err {
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }

    r := check os.Open(src)
    defer r.Close()

    w := check os.Create(dst)
    handle err {
        w.Close()
        os.Remove(dst) // (only if a check fails)
    }

    check io.Copy(w, r)
    check w.Close()
    return nil
}
```

- **handle** block specifies error handler code

Handle/check

```
func CopyFile(src, dst string) error {
    handle err {
        return fmt.Errorf("copy %s %s: %v", src, dst, err)
    }

    r := check os.Open(src)
    defer r.Close()

    w := check os.Create(dst)
    handle err {
        w.Close()
        os.Remove(dst) // (only if a check fails)
    }

    check io.Copy(w, r)
    check w.Close()
    return nil
}
```

- **handle** block specifices error handler code
- executed if **check** fails

Handle/check

```
func printSum(a, b string) error
{
  x, err := strconv.Atoi(a)
  if err != nil {
    return err
  }
  y, err := strconv.Atoi(b)
  if err != nil {
    return err
  }
  fmt.Println("result:", x + y)
  return nil
}
```

Handle/check

```
func printSum(a, b string) error
{
  x, err := strconv.Atoi(a)
  if err != nil {
    return err
  }
  y, err := strconv.Atoi(b)
  if err != nil {
    return err
  }
  fmt.Println("result:", x + y)
  return nil
}
```

```
func printSum(a, b string) error
{
  handle err {
    return err
  }
  x := check strconv.Atoi(a)
  y := check strconv.Atoi(b)
  fmt.Println("result:", x + y)
  return nil
}
```

Check

Check

```
v1, ..., vN := check <expr>
```

Check

```
v1, ..., vN := check <expr>

is equivalent to

v1, ..., vN, vErr := <expr>
if vErr != nil {
      <error result> = handlerChain(vn)
      return
}
```

Check

```
foo(check <expr>)
```

Check

```
foo(check <expr>)

is equivalent to

v1, ..., vN, vErr := <expr>
if vErr != nil {
       <error result> = handlerChain(vn)
       return
}
foo(v1, ..., vN)
```

Check

```
func printSum(a, b string) error {
	handle err { return err }
	fmt.Println("result:", check strconv.Atoi(x) +
check strconv.Atoi(y))
	return nil
}
```

Handle

Handle

```
handle err {
   // error handling code
}
```

- Takes an argument of type **error**
- Executed in lexical scope in **reverse order**
- Executes a **return** statement

Handle

```
func process(user string, files chan string) (n int, err error) {
    handle err { return 0, fmt.Errorf("process: %v", err)  }
    for i := 0; i < 3; i++ {
        handle err { err = fmt.Errorf("attempt %d: %v", i, err) }
        handle err { err = moreWrapping(err) }

        check do(something())
    }
    check do(somethingElse())
}
```

Handle

```
func process(user string, files chan string) (n int, err error) {
    handle err { return 0, fmt.Errorf("process: %v", err)  }
    for i := 0; i < 3; i++ {
        handle err { err = fmt.Errorf("attempt %d: %v", i, err) }
        handle err { err = moreWrapping(err) }

        check do(something())
    }
    check do(somethingElse())
}
```

A

Handle

```
func process(user string, files chan string) (n int, err error) {
    handle err { return 0, fmt.Errorf("process: %v", err)  }     A
    for i := 0; i < 3; i++ {
        handle err { err = fmt.Errorf("attempt %d: %v", i, err) }B
        handle err { err = moreWrapping(err) }

        check do(something())
    }
    check do(somethingElse())
}
```

Handle

```
func process(user string, files chan string) (n int, err error) {
    handle err { return 0, fmt.Errorf("process: %v", err)  }      A
    for i := 0; i < 3; i++ {
        handle err { err = fmt.Errorf("attempt %d: %v", i, err) } B
        handle err { err = moreWrapping(err) }                    C

        check do(something())
    }
    check do(somethingElse())
}
```

Handle

```
func process(user string, files chan string) (n int, err error) {
    handle err { return 0, fmt.Errorf("process: %v", err)  }      A
    for i := 0; i < 3; i++ {
        handle err { err = fmt.Errorf("attempt %d: %v", i, err) } B
        handle err { err = moreWrapping(err) }                    C

        check do(something())  // check 1: handler chain C, B, A
    }
    check do(somethingElse())
}
```

Handle

```
func process(user string, files chan string) (n int, err error) {
    handle err { return 0, fmt.Errorf("process: %v", err)  }      A
    for i := 0; i < 3; i++ {
        handle err { err = fmt.Errorf("attempt %d: %v", i, err) } B
        handle err { err = moreWrapping(err) }                    C

        check do(something())  // check 1: handler chain C, B, A
    }
    check do(somethingElse())  // check 2: handler chain A
}
```

Default handler

- All functions with last return value error have **default handler**
- It returns current values (or zero values)

Handle

```
func printSum(a, b string) error {
  handle err {
    return err
  }
  x := check strconv.Atoi(a)
  y := check strconv.Atoi(b)
  fmt.Println("result:", x + y)
  return nil
}
```

Handle

```
func printSum(a, b string) error {
  x := check strconv.Atoi(a)
  y := check strconv.Atoi(b)
  fmt.Println("result:", x + y)
  return nil
}
```


Errors are values

Errors are values

```
type Error struct {
        Func string
        User string
        Path string
        Err  error
}

func (e *Error) Error() string

func ProcessFiles(user string, files chan string) error {
        e := Error{ Func: "ProcessFile", User: user}
        handle err { e.Err = err; return &e } // handler A
        u := check OpenUserInfo(user)         // check 1
        defer u.Close()
        for file := range files {
                handle err { e.Path = file }       // handler B
                check process(check os.Open(file)) // check 2
        }
        ...
}
```

Drawbacks

Drawbacks

- Doesn't play well with defer/panic
- Can't check inside handlers
- Introduces flow break (like break/continue)
- Appear to be similar to exceptions (but it's not)
- Adds new thing to learn in Go

Error Values Draft

The Problem

- Hard to inspect errors
- Hard to reports deeply nested errors

The Problem

- Four ways to test errors:
  - check equality with vars **(if err == io.EOF)**
  - type switch **(if _, ok := err.(MyError))**
  - special functions **(os.IsNotExist())**
  - string search **(strings.Contains(err.Error(), "wtf"))**

The Problem

- Most common error reporting patterns:

```
if err != nil {
  return fmt.Errorf("write users database: %v", err)
}
```

```
if err != nil {
      return &WriteError{Database: "users", Err: err}
}
```

The Problem

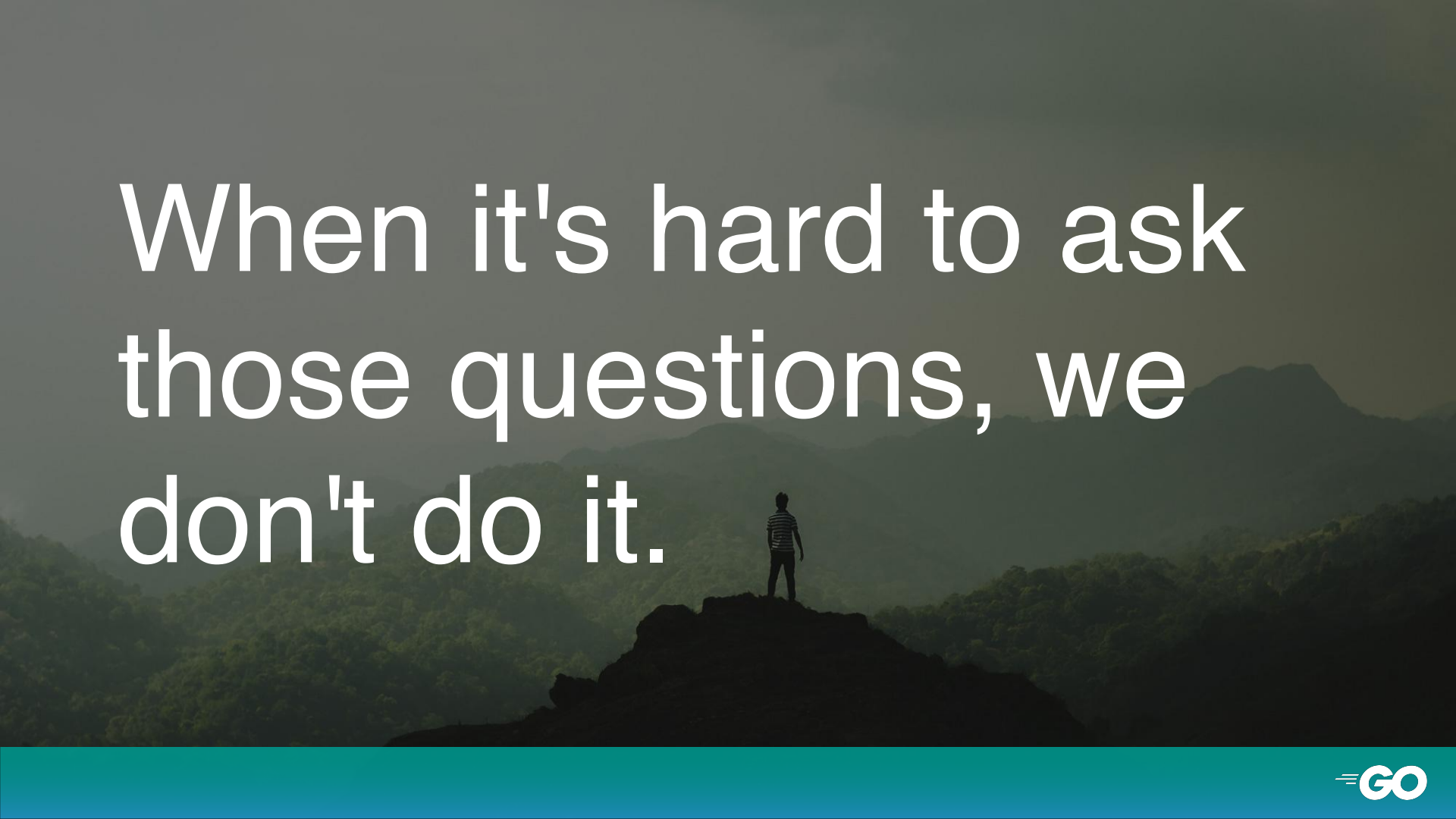
- Example error from database write:

```
write users database: call myserver.Method: \
    dial myserver:3333: open /etc/resolv.conf: permission denied
```

- A WriteError, which provides "write users database: " and wraps
- an RPCError, which provides "call myserver.Method: " and wraps
- a net.OpError, which provides "dial myserver:3333: " and wraps
- an os.PathError, which provides "open /etc/resolv.conf: " and wraps
- syscall.EPERM, which provides "permission denied"

The Problem

- You might want to ask questions in code:
  - Is it an RPCError?
  - Is it a net.OpError?
  - Does it satisfy the net.Error interface?
  - Is it an os.PathError?
  - Is it a permission error?


When it's hard to ask
those questions, we
don't do it.

Goals

- Make error inspection easier
- Add ability to print more detailed error info
- Existing errors should work
- Keep error creation cheap (errors are not exceptions)

Error Inspection

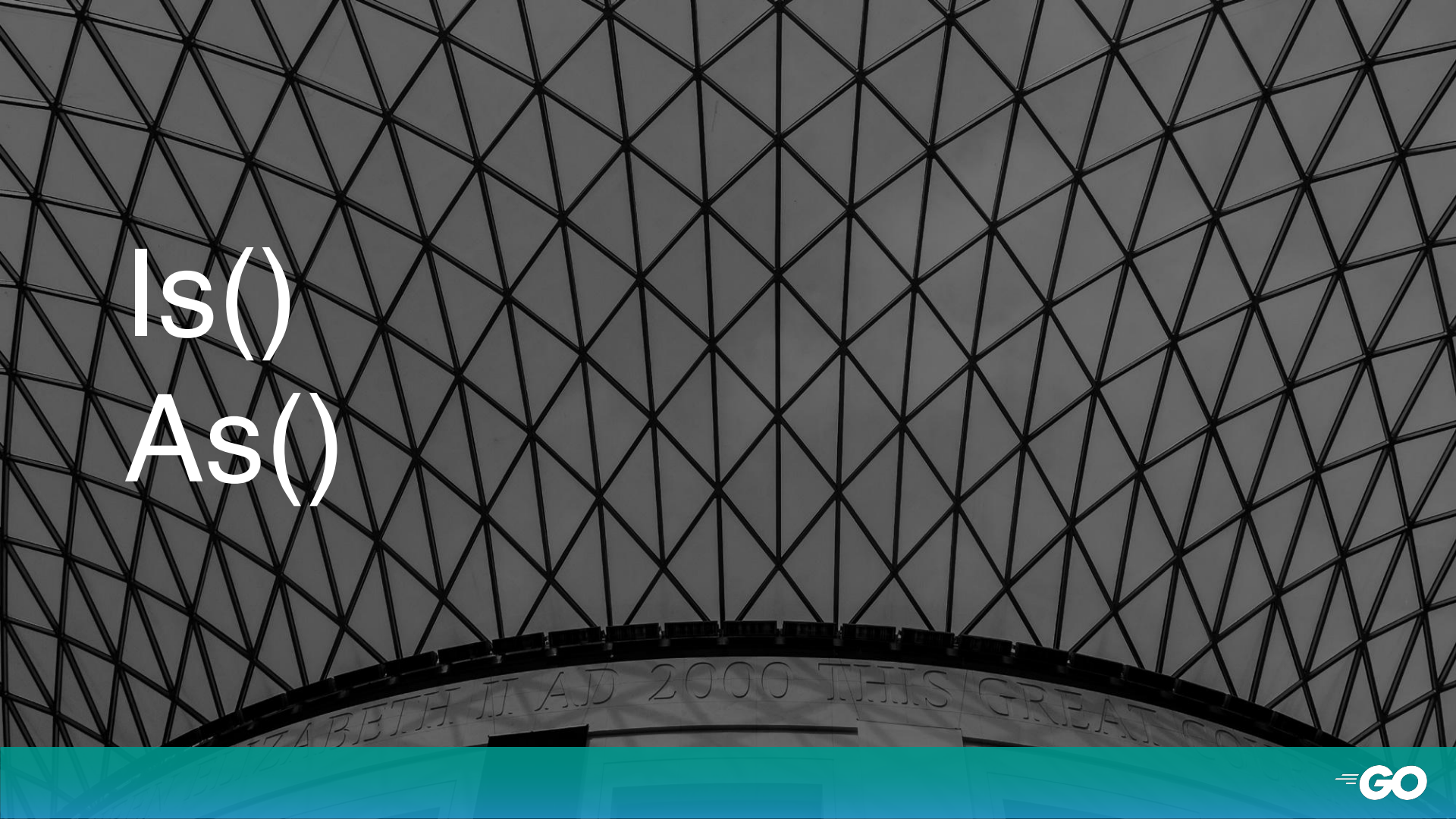
For programs

Errors Inspection

```
package errors

// A Wrapper is an error implementation
// wrapping context around another error.
type Wrapper interface {
  // Unwrap returns the next error in the error chain.
  // If there is no next error, Unwrap returns nil.
  Unwrap() error
}
```

- New Wrapper interface in errors package
- Any error type can implement this interface
- error interface remains unchanged


Is()
As()

Errors Inspection

```
// instead of err == io.ErrUnexpectedEOF
if errors.Is(err, io.ErrUnexpectedEOF) { ... }
```

- Equality check then replaced by errors.Is

Errors Inspection

```
func Is(err, target error) bool {
  for {
    if err == target {
        return true
    }
    wrapper, ok := err.(Wrapper)
    if !ok {
      return false
    }
    err = wrapper.Unwrap()
    if err == nil {
      return false
    }
  }
}
```

- It essentially loops over nested errors

Errors Inspection

```
// instead of pe, ok := err.(*os.PathError)

pe, ok := errors.As(*os.PathError)(err)
If ok {
    ... pe.Path ...
}
```

- Type assertion then replaced by errors.As

Errors Inspection

```
// instead of pe, ok := err.(*os.PathError)

pe, ok := errors.As(*os.PathError)(err)
If ok {
    ... pe.Path ...
}
```

- Type assertion then replaced by errors.As
- Uses new contract draft design

Errors Inspection

```
// instead of pe, ok := err.(*os.PathError)

pe, ok := errors.As(*os.PathError)(err)
If ok {
    ... pe.Path ...
}

// without generics
var pe *os.PathError
if errors.AsValue(&pe, err) { ... pe.Path ... }
```

- Type assertion then replaced by errors.As
- Uses new contract draft design
- Without generics can be done as well

Errors Inspection

```
func As(type E)(err error) (e E, ok bool) {
  for {
    if e, ok := err.(E); ok {
        return e, true
    }
    wrapper, ok := err.(Wrapper)
    if !ok {
      return e, false
    }
    err = wrapper.Unwrap()
    if err == nil {
      return e, false
    }
  }
}
```

- It essentially loops over nested errors

Design considerations

- Easier to inspect errors for programs, not humans
- Unifies third-party error helpers packages
- Doesn't handle well case of multiple error checks

Error Printing

For humans

Problem

- Humans need readable error representation
- Sometimes we need stacktraces
- Or file name and line number info
- Or we need i18n for errors

Errors Printing

```
package errors

// A Formatter formats error messages.
type Formatter interface {
  // Format is implemented by errors to print a single error message.
  // It should return the next error in the error chain, if any.
  Format(p Printer) (next error)
}

// Printer is implemented by fmt.
type Printer interface {
  Print(args ...interface{})
  Printf(format string, args ...interface{})
  // Detail reports whether error detail is requested.
  Detail() bool
}
```

- Two interfaces - **Formatter** and **Printer**
- **Formatter** can be implemented by errors
- **Printer** designed to allow i18n
- fmt or golang.org/x/text/message.

Errors Printing

```
import "golang.org/x/text/message"

p := message.NewPrinter(language.Dutch)
p.Printf("Error: %v", err)
```

- Localization

Errors Printing

```
type myAddrError struct {
  address string
  detail  string
  err     error
}

func (e *myAddrError) Error() string {
  return fmt.Sprint(e) // delegate to Format
}

func (e *myAddrError) Format(p errors.Printer) error {
  p.Printf("address %s", e.address)
  if p.Detail() {
    p.Print(e.detail)
  }
  return e.err
}
```

- Example

Errors Printing

```
foo: bar(nameserver 139): baz flopped

// with detailed output

foo:
    file.go:123 main.main+0x123
--- bar(nameserver 139):
    some detail only text
    file.go:456
--- baz flopped:
    file.go:789
```

- Proposed output

Errors Printing

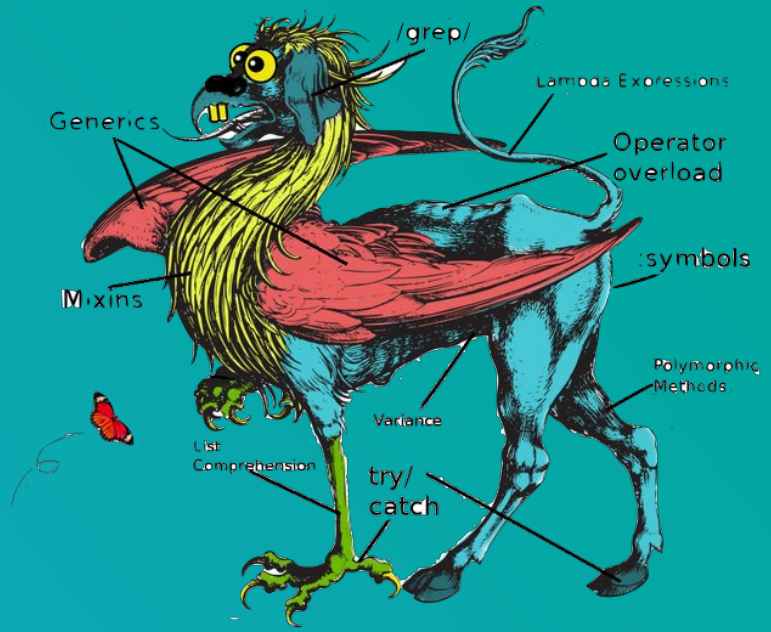
- Use "%+v" to print error in detailed format
- Existing verbs of fmt.Printf:
  - %s: err.Error() as a string
  - %q: err.Error() as a quoted string
  - %+q: err.Error() as an ASCII-only quoted string
  - %v: err.Error() as a string
  - %#v: err as a Go value, in Go syntax

Errors Printing

- Use "%+v" to print error in detailed format
- Existing verbs of fmt.Printf:
  - %s: err.Error() as a string
  - %q: err.Error() as a quoted string
  - %+q: err.Error() as an ASCII-only quoted string
  - %v: err.Error() as a string
  - %#v: err as a Go value, in Go syntax
  - %+v: err in multi-line format

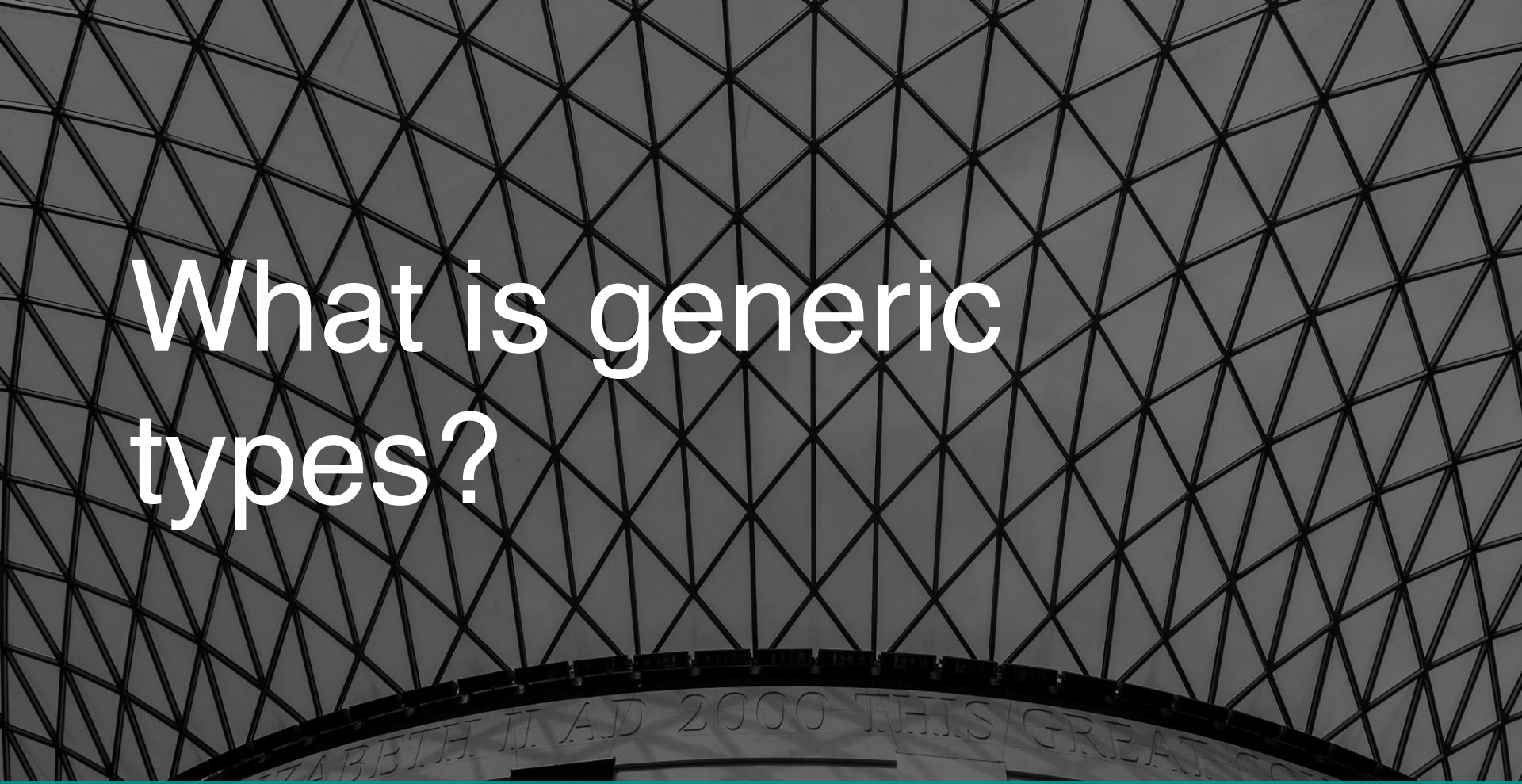
Generics Draft

Generics Draft



Questions

- What is generics?
- Why many people think they're crucial?
- Why Go succeeded over 10 years w/o them?
- Why I've seen so many generics abuse (in other languages)?



What is generic types?

What is generics

- Recommended read - Summary of Go Generics Discussion
- Generics is an umbrella term for a few things:
  - Generic Data Structures (trees, graphs, sets)
  - Generic Algorithms (sort, min/max, FFT, etc)
  - Other esoteric use-cases (language extensions, etc)

What is generics

- Go has built-in generics (maps, slices, new, make, etc)
- Users cannot create own generic types and algorithms

What is generics

- Three approaches:
  - leave them out (C, Go)
  - compile-time specialization or macro expansion (C++)
  - boxing (Java approach)

+ bunch of esoteric approaches like witness tables (Swift)

What is generics

- Three approaches:
  - leave them out (C, Go)
  - compile-time specialization or macro expansion (C++)
  - boxing (Java approach)

Slow compilation

+ bunch of esoteric approaches like witness tables (Swift)

What is generics

- Three approaches:
  - leave them out (C, Go)
  - compile-time specialization or macro expansion (C++) Slow compilation
  - boxing (Java approach) Slow execution

+ bunch of esoteric approaches like witness tables (Swift)

What is generics

- Three approaches:
  - leave them out (C, Go)
  - compile-time specialization or macro expansion (C++)
  - boxing (Java approach)

Slow compilation

Complicated code

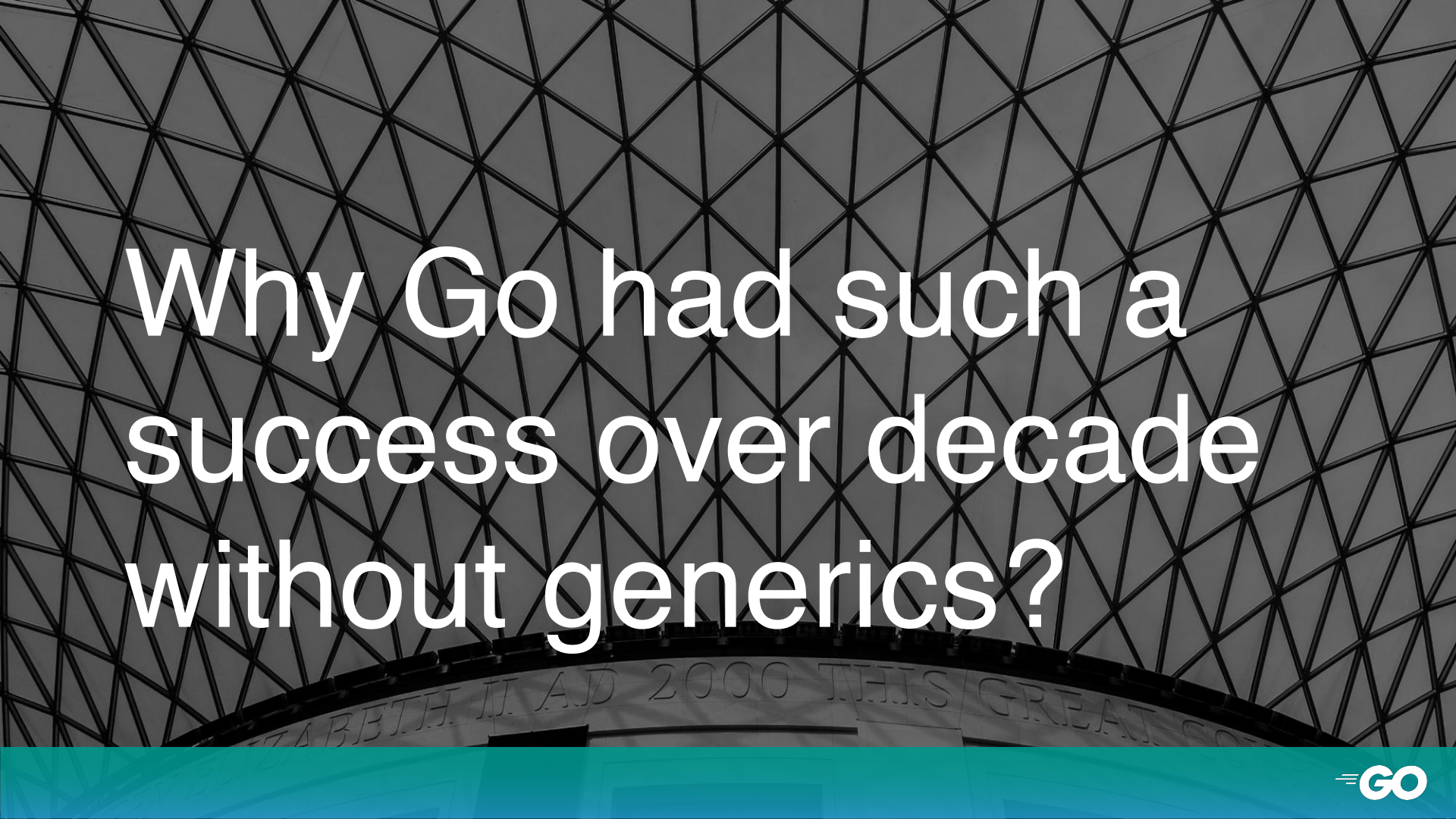
Slow execution

+ bunch of esoteric approaches like witness tables (Swift)


Why people think
they're crucial?

Why some people think generics is must-have

- In some fields, it's really a must (compilers, math software, algorithms designers)
- Allow to reuse the code for many types
- 99% answers I got for this question:
  - "Let's imagine I want to write the code that works with any type"


Why Go had such a success over decade without generics?

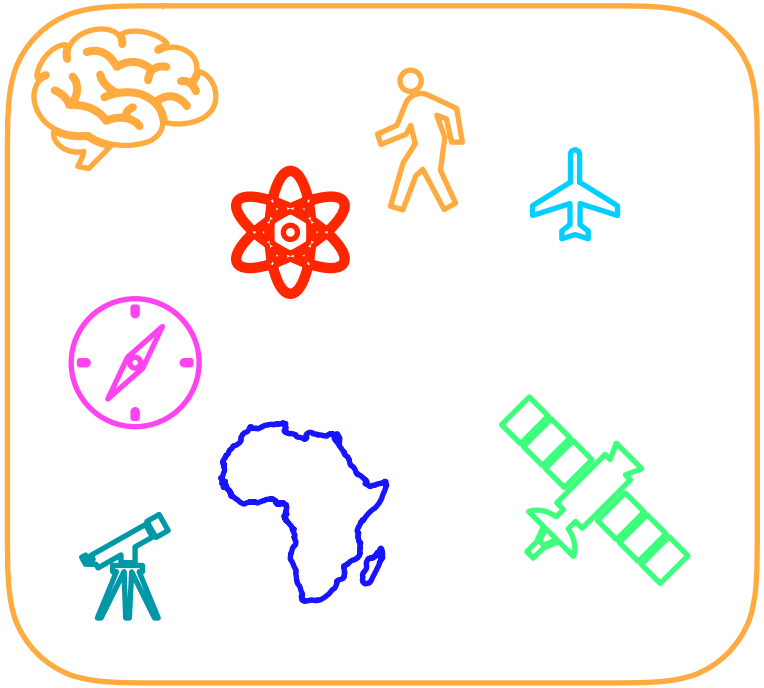
Why Go succeeded without user generics?

Your brain

Code

Why Go succeeded without user generics?

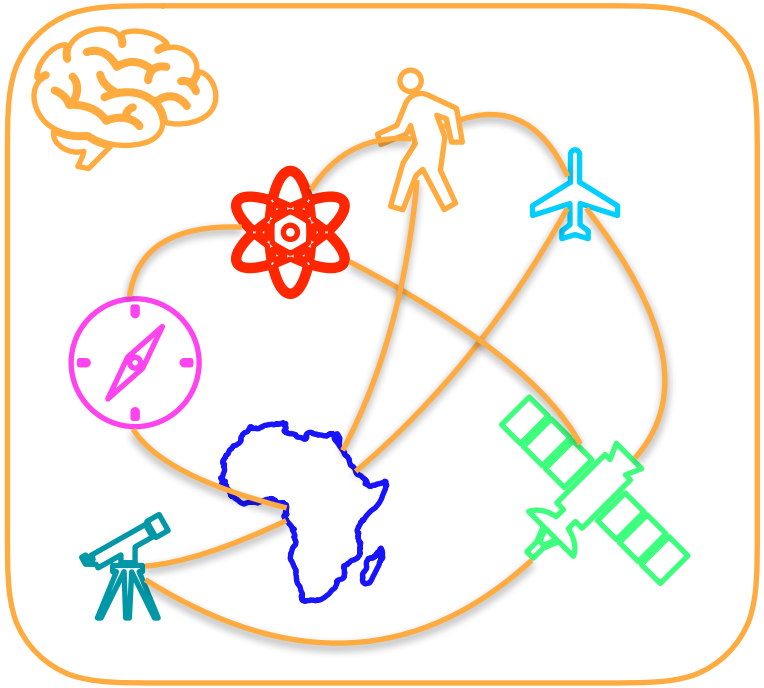
Your brain



Code

Why Go succeeded without user generics?

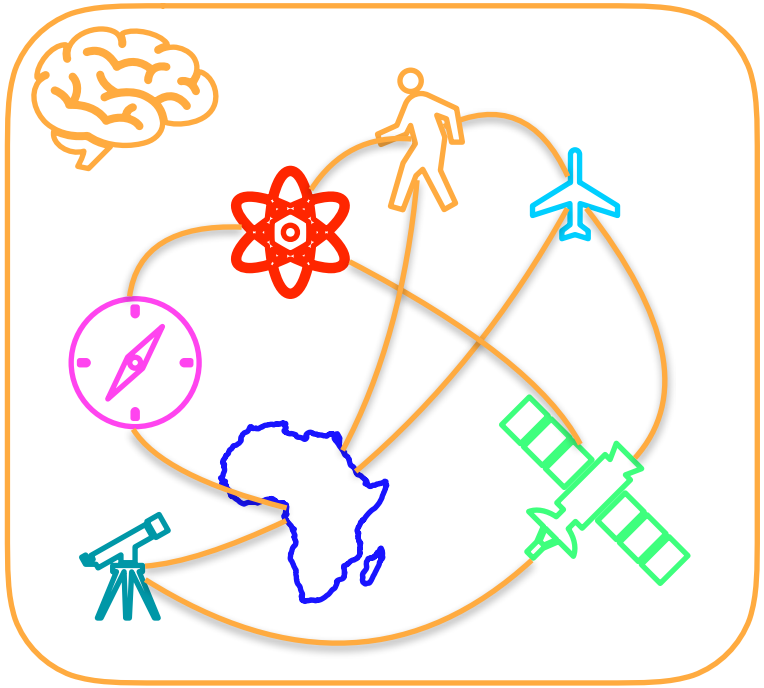
Your brain



Code

Why Go succeeded without user generics?

Your brain



Code

```
type Atom struct { ... }
type Human struct { ... }
type Plane struct { ... }
type Continent struct { ... }
type Telescope struct { ... }
type Satellite struct { ... }
type Compass struct { ... }
```

Why Go succeeded without user generics?

- Programming is a translation of mental model to the code
- The task is to make this process reversable
- By reading code you should be able to restore the mental model of the problem domain

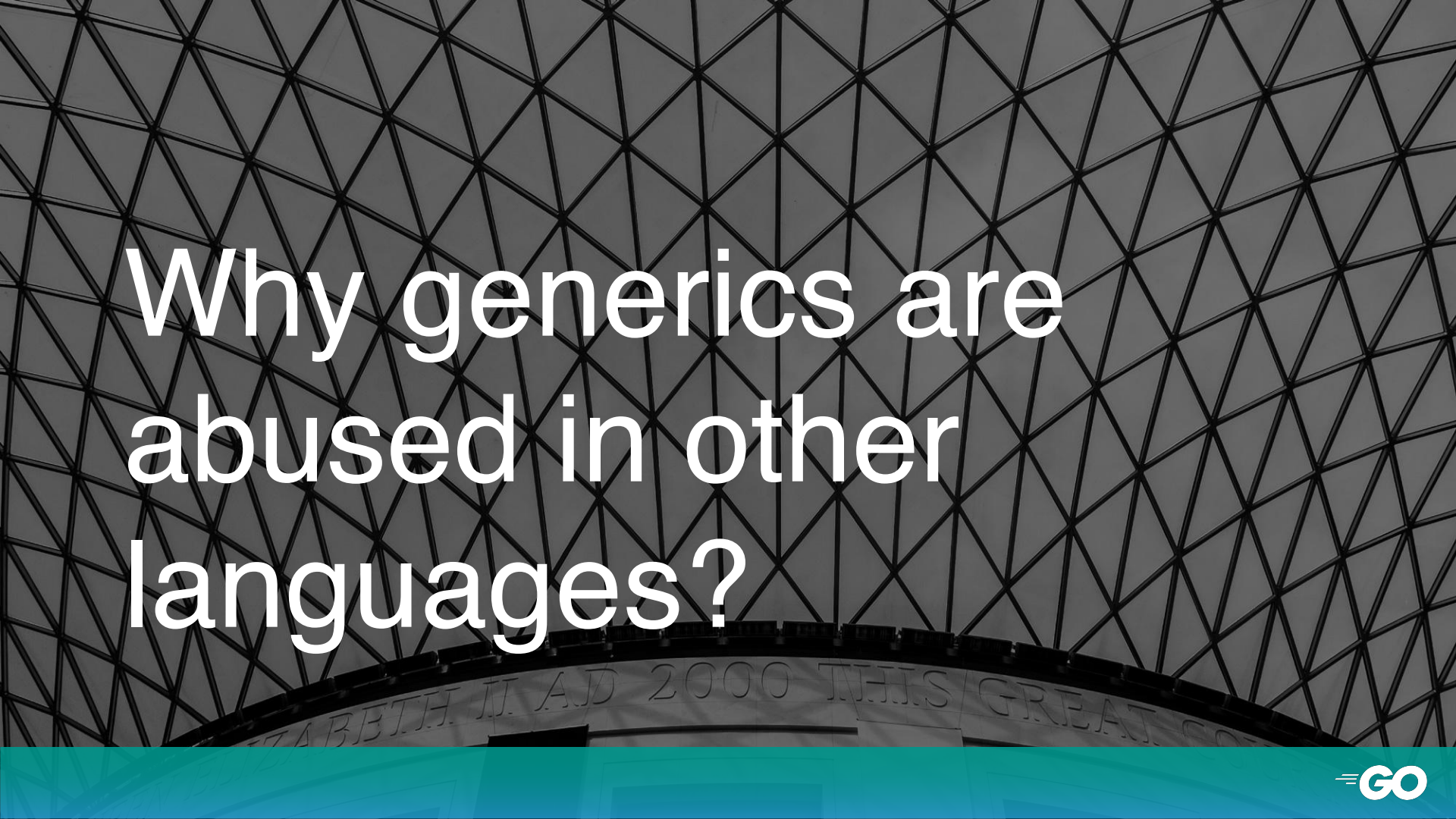
Why some people think generics is must-have

- In practice you work mostly with concrete types:
  - protocols, commands, users, events, tickets, files, coordinates, documents, etc. etc.
- It's 10% where you really need "with any type" code

Why some people think generics is must-have

- Go has 4 ways to implement "generic"-ish things:
  - copy-paste (sometimes you just need two types, really)
  - interfaces (they provide type generalization)
  - reflect (maps are implemented using reflect)
  - code generation

It's not "native generics support", but it covers 10%


Why generics are abused in other languages?

Why generics often abused?

- Often positioned as "superior" way to write programs
- I.e. "generic" types is better than "concrete", (because they can be reused)
- The question is how to prevent this with Go?

Contracts Design

Contracts

Contracts

```
contract stringer(x T) {
  var s string = x.String()
}
```

- New syntax construct
- Contract code specifies requirement for type
- Evaluated on compile time for each invocation of this type

Contracts

```
contract stringer(x T) {
  var s string = x.String()
}

func Stringify(type T stringer)(s []T) (ret []string) {
  for _, v := range s {
    ret = append(ret, v.String()) // now valid
  }
  return ret
}
```

- New syntax construct
- Contract code specifies requirement for type
- Evaluated on compile time for each invocation of this type

Contracts

```
contract convertible(_ To, f From) {
    To(f)
}

func FormatUnsigned(type T convertible(uint64,
T))(v T) string {
    return strconv.FormatUint(uint64(v), 10)
}

s := FormatUnsigned(rune)('a')
```

- New syntax construct
- Contract code specifies requirement for type
- Evaluated on compile time for each invocation of this type

Contracts

```
contract viaStrings(t To, f From) {
  var x string = f.String()
  t.Set(string("")) // could also use t.Set(x)
}

func SetViaStrings(type To, From viaStrings)(s
[]From) []To {
  r := make([]To, len(s))
  for i, v := range s {
    r[i].Set(v.String())
  }
  return r
}
```

- New syntax construct
- Contract code specifies requirement for type
- Evaluated on compile time for each invocation of this type

Contracts

```
contract PrintStringer(x X) {
        stringer(X)
        x.Print()
}
```

- contracts also can be embedded

Contracts

```
contract add1K(x T) {
        x = 1000
        x + x
}

func Add1K(type T add1K)(s []T) {
        for i, v := range s {
                s[i] = v + 1000
        }
}
```

- adds 1000 to each element of the slice

Contracts

- In a way, contracts are superset of interfaces
- Interfaces restrict methods type implements, contracts specify/restrict pretty much everything

Parametrized types

Parametrized types

```
type Vector(type Element) []Element
```

- Type now can be Thing(T)
- Type can take arguments

Parametrized types

```
type Vector(type Element) []Element

var v Vector(int)

func (v *Vector(Element)) Push(x Element) { *v =
append(*v, x) }
```

- Type now can be Thing(T)
- Type can take arguments

Parametrized types

```
type Lockable(type T) struct {
        T
        mu sync.Mutex
}

func (l *Lockable(T)) Get() T {
        l.mu.Lock()
        defer l.mu.Unlock()
        return l.T
}
```

- Type now can be Thing(T)
- Type can take arguments

Parametrized types

```
package pair

type Pair(type carT, cdrT) struct {
        f1 carT
        f2 cdrT
}
```

- Type now can be Thing(T)
- Type can take arguments

Conclusion

Thoughts

- Go team believes most people will not write generic code themselves
- Reading generic code should be much easier then writing it
- Most complicated part will be writing contract bodies

Thoughts

- No metaprogramming (code that generates code that will be executed at runtime)
- No operator methods
- No specialization (no way to write multiple versions of generics function)

Issues

- Zero value?
- Pointer vs value methods
- Lots of silly parentheses
  - F(int, float64)(x, y)(s)

Thank you