Will contracts replace interfaces?

@francesc

Agenda

- Interfaces
- Generics draft:
  - Type Parameters
  - Contracts
- Interfaces vs Contracts

what is an interface?

"In object-oriented programming, a protocol or interface is a common means for unrelated objects to communicate with each other"

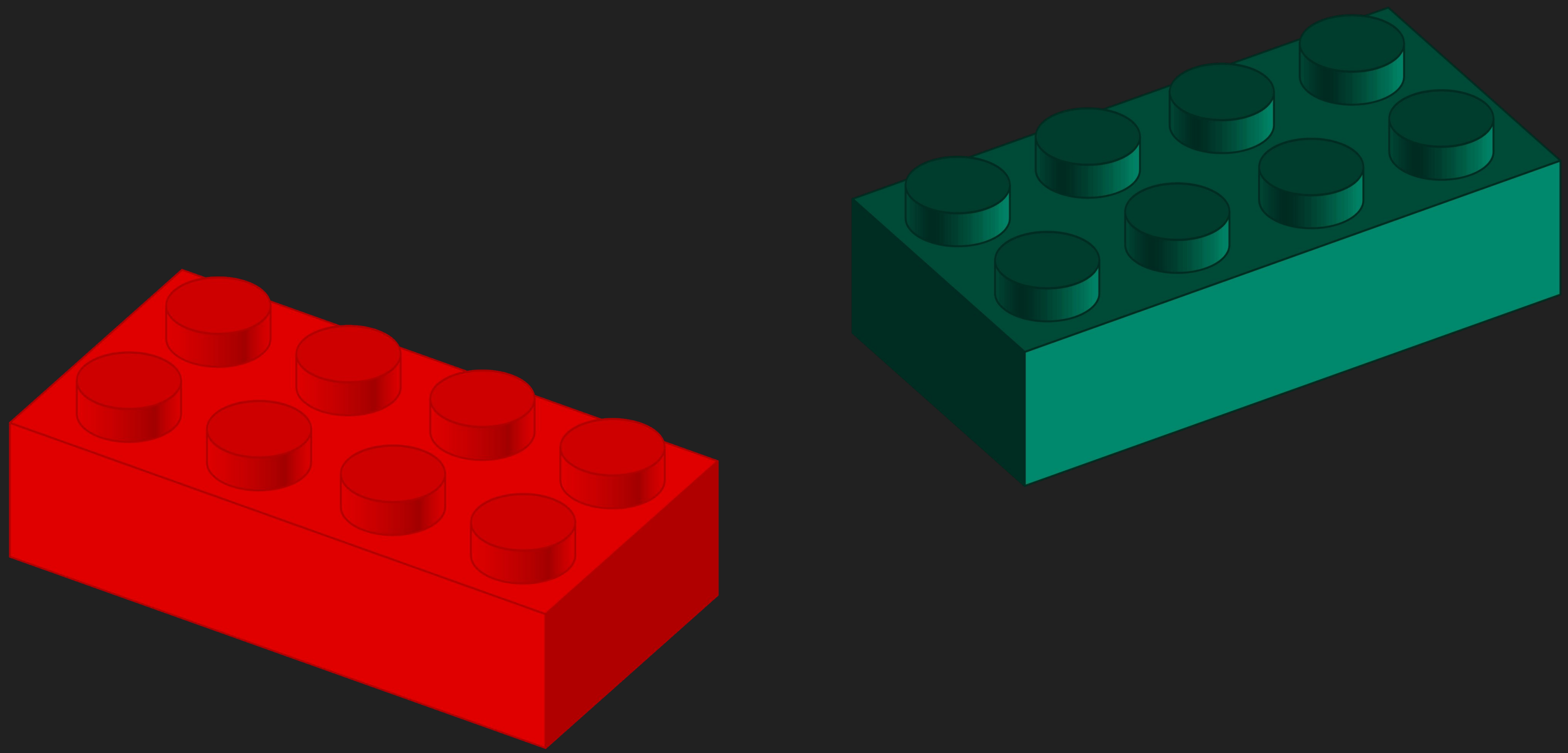
\- wikipedia

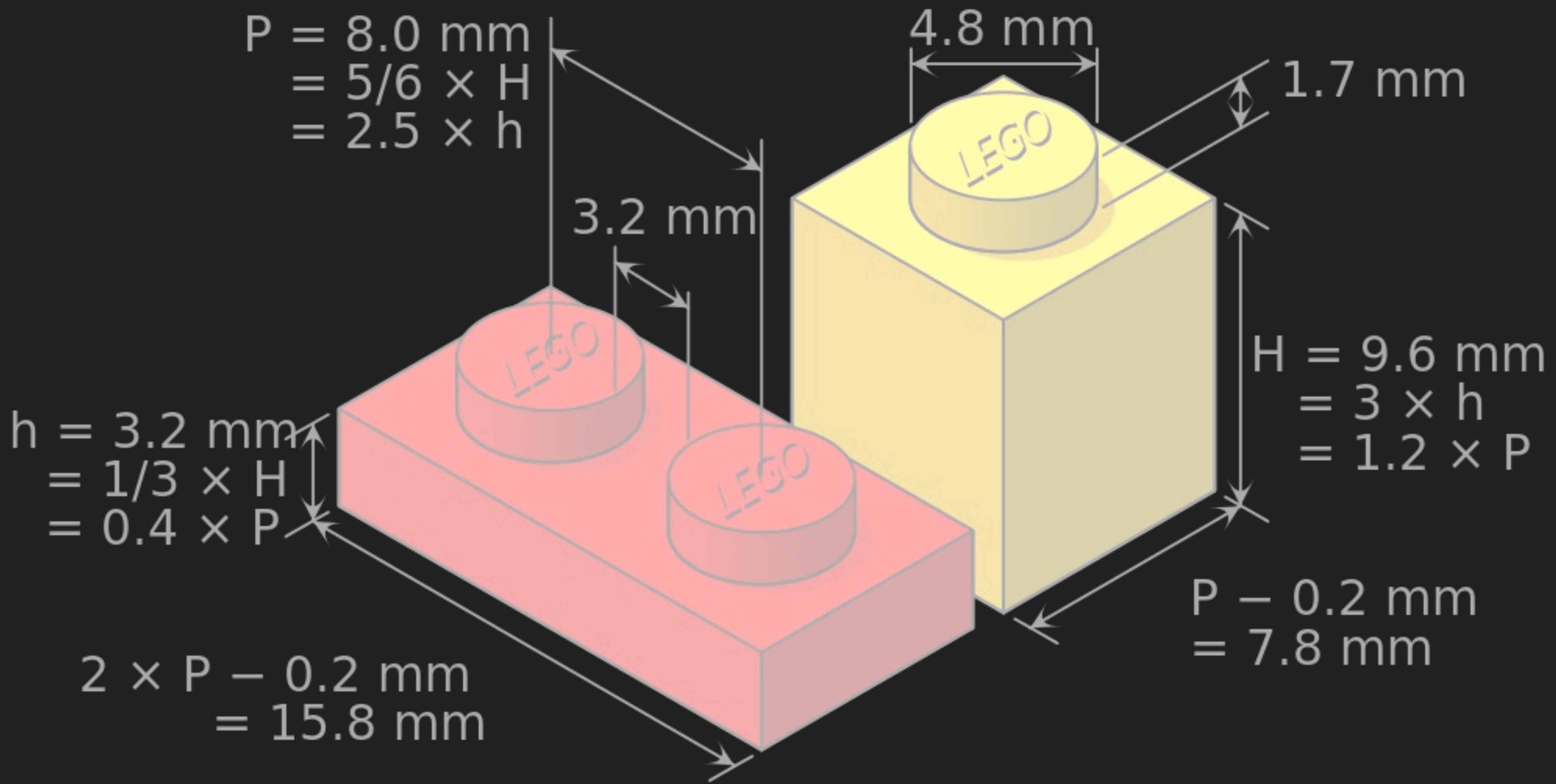
"In object-oriented programming, a protocol or interface is a common means for unrelated objects to communicate with each other"

- wikipedia

"In object-oriented programming, a protocol or interface is a common means for unrelated objects to communicate with each other"

- wikipedia





P = 8.0 mm
= 5/6 × H
= 2.5 × h
4.8 mm
1.7 mm
3.2 mm
LEGO
LEGO
LEGO
H = 9.6 mm
= 3 × h
= 1.2 × P
h = 3.2 mm
= 1/3 × H
= 0.4 × P
P − 0.2 mm
= 7.8 mm
2 × P − 0.2 mm
= 15.8 mm


MSC
MSC TOMOKO
PANAMA
© Michael "Mike" L. Baird bairdphotos.com

what is a Go interface?

abstract types

concrete types

concrete types in Go

- they describe a memory layout

- behavior attached to data through methods

```
type Number int

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


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

abstract types in Go

- they describe behavior

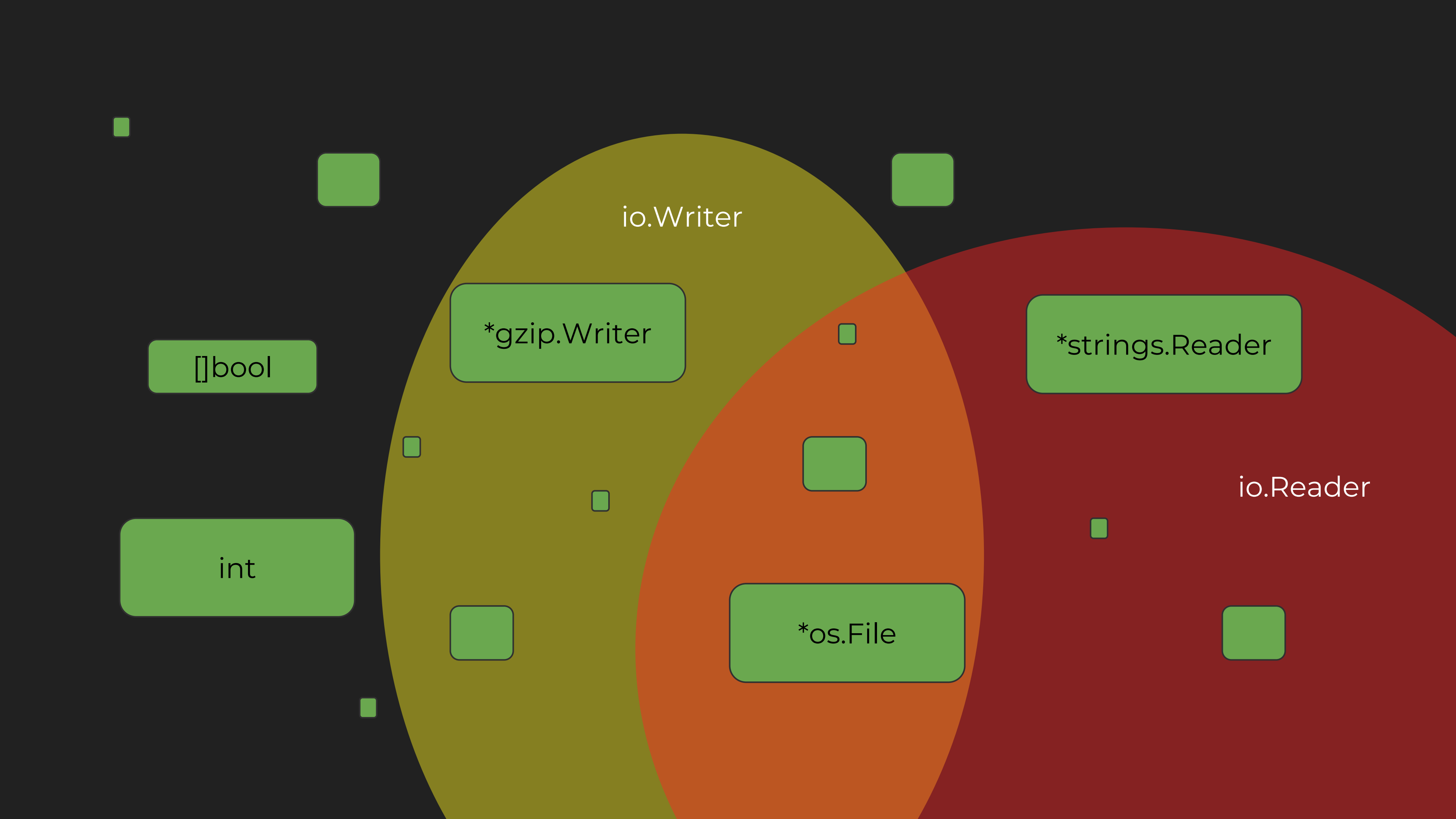
io.Reader | io.Writer | fmt.Stringer

- 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)
}
```

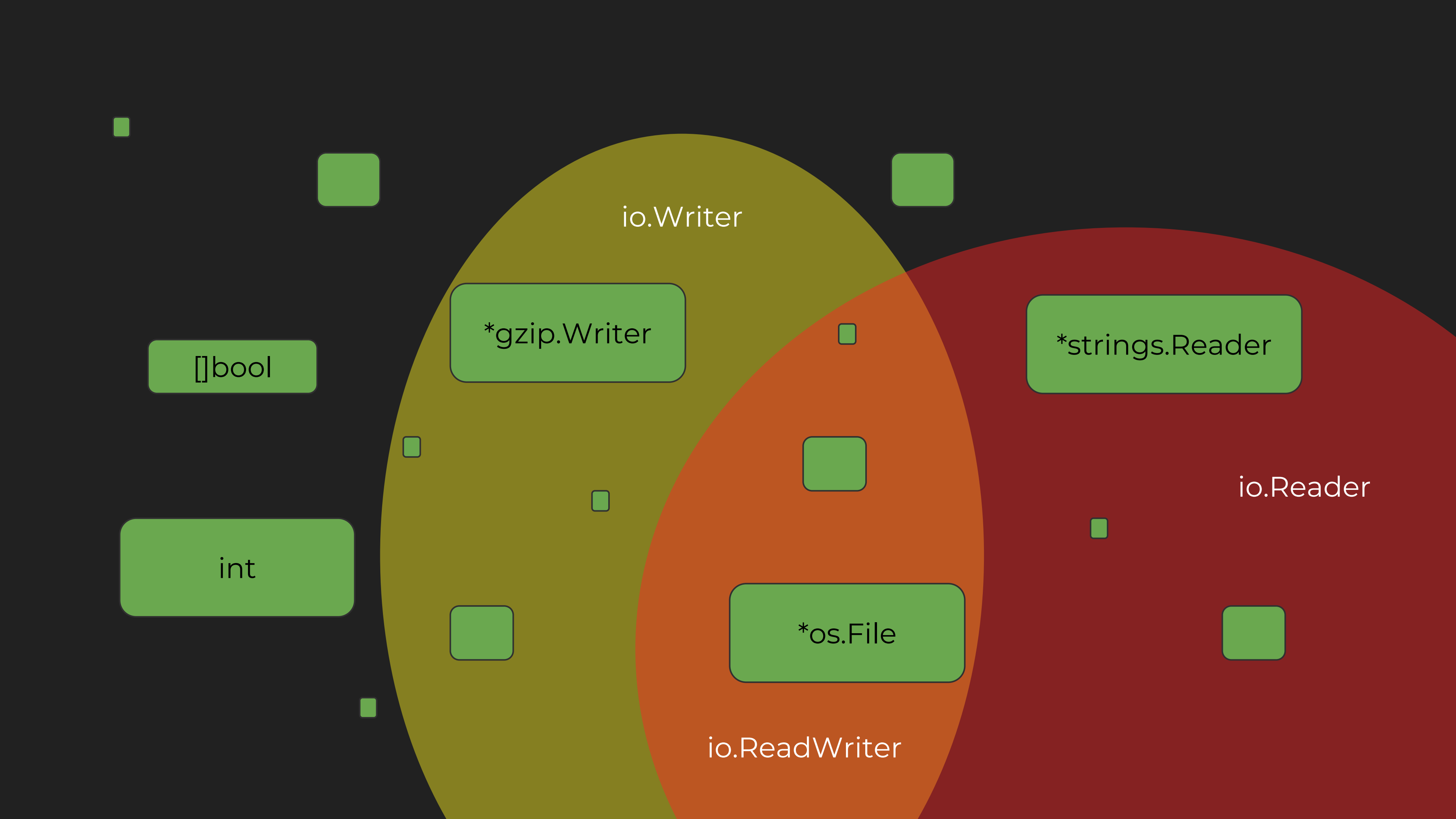

io.Writer
*gzip.Writer
[]bool
*strings.Reader
io.Reader
int
*os.File

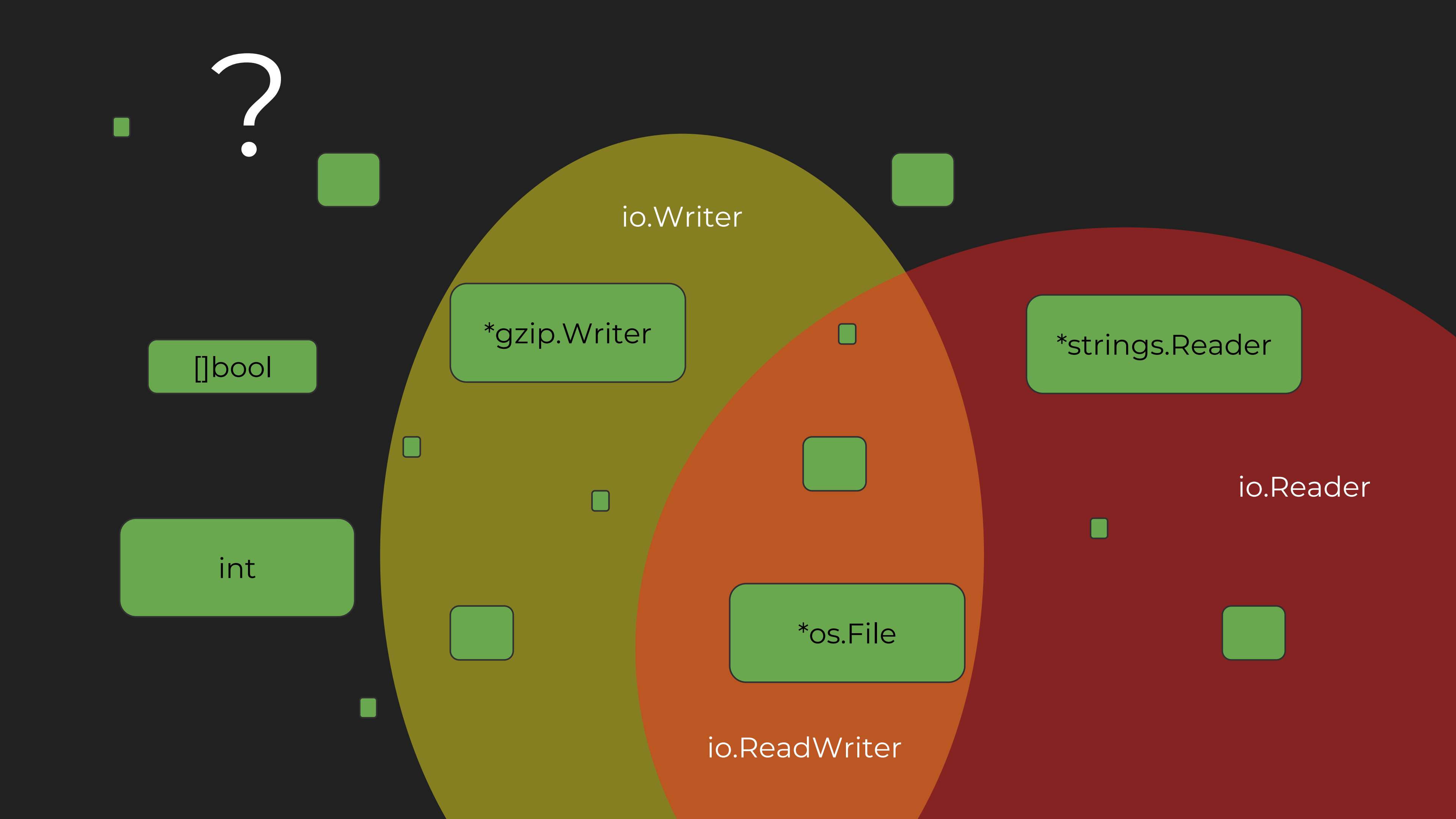
union of interfaces

```go
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


?
[]bool
int
io.Writer
*gzip.Writer
*strings.Reader
io.Reader
*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

what function do you prefer?

a)
```go
func WriteTo(f *os.File) error
```

b)
```go
func WriteTo(w io.ReadWriteCloser) error
```

c)
```go
func WriteTo(w io.Writer) error
```

d)
```go
func WriteTo(w interface{}) error
```

a) `func WriteTo(f *os.File) error`

Cons:

- how would you test it?
- what if you want to write to memory?

Pros:

- ?

d) `func WriteTo(w interface{}) error`

Cons:

- how do you even write to interface{}?
- probably requires runtime checks

Pros:

- you can write really bad code

```
b)      func WriteTo(w io.ReadWriteCloser) error
c)      func WriteTo(w io.Writer) error
```

Which ones does WriteTo really need?

- Write
- Read
- Close

"The bigger the interface, the weaker the abstraction"

- Rob Pike in his Go Proverbs

“Be conservative in what you do, be liberal in what you accept from others”

- Robustness Principle

“Be conservative in what you send, be liberal in what you accept”

- Robustness Principle

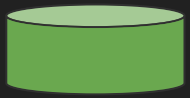
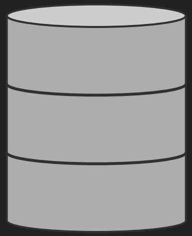
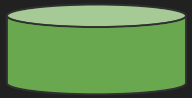
Abstract Data Types

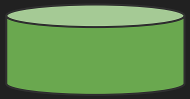
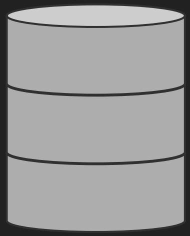
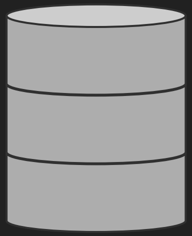
Abstract Data Types

Mathematical model for data types

Defined by its behavior in terms of:

- possible values,
- possible operations on data of this type,
- and the behavior of these operations

top(push(x , s ))= x 

pop(push(x  , s )) = s 

```
empty(new())

not empty(push(S, X))
```

Example: stack ADT

Axioms:

top(push(S, X)) = X

pop(push(S, X)) = S

empty(new())

!empty(push(S, X))

a Stack interface

```
type Stack interface {
    Push(v interface{}) Stack
    Top() interface{}
    Pop() Stack
    Empty() bool
}
```

algorithms on Stack

```
func Size(s Stack) int {
    if s.Empty() {
        return 0
    }
    return Size(s.Pop()) + 1
}
```

a sortable interface

```
type Interface interface {
    Less(i, j int) bool
    Swap(i, j int)
    Len() int
}
```

algorithms on sortable

```
func Sort(s Interface)

func Stable(s Interface)

func IsSorted(s Interface) bool
```

remember Reader and Writer?

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


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

algorithms on Reader and Writer

```
func Fprintln(w Writer, ar ...interface{}) (int, error)

func Fscan(r Reader, a ...interface{}) (int, error)

func Copy(w Writer, r Reader) (int, error)
```

is this enough?

type **Reader**

Reader is the interface that wraps the basic Read method.

Read reads up to len(p) bytes into p. It returns the number of bytes read (0 <= n <= len(p)) and any error encountered. Even if Read returns n < len(p), it may use all of p as scratch space during the call. If some data is available but not len(p) bytes, Read conventionally returns what is available instead of waiting for more.

When Read encounters an error or end-of-file condition after successfully reading n > 0 bytes, it returns the number of bytes read. It may return the (non-nil) error from the same call or return the error (and n == 0) from a subsequent call. An instance of this general case is that a Reader returning a non-zero number of bytes at the end of the input stream may return either err == EOF or err == nil. The next Read should return 0, EOF.

Callers should always process the n > 0 bytes returned before considering the error err. Doing so correctly handles I/O errors that happen after reading some bytes and also both of the allowed EOF behaviors.

Implementations of Read are discouraged from returning a zero byte count with a nil error, except when len(p) == 0. Callers should treat a return of 0 and nil as indicating that nothing happened; in particular it does not indicate EOF.

Implementations must not retain p.

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

write generic algorithms on interfaces

“Be conservative in what you send, be liberal in what you accept”

- Robustness Principle

what function do you prefer?

a)
```go
func New() *os.File
```

b)
```go
func New() io.ReadWriteCloser
```

c)
```go
func New() io.Writer
```

d)
```go
func New() interface{}
```

```
func New() *os.File
```

“Be conservative in what you send, be liberal in what you accept”

- Robustness Principle

"Return concrete types, receive interfaces as parameters"

- Robustness Principle applied to Go (me) unless

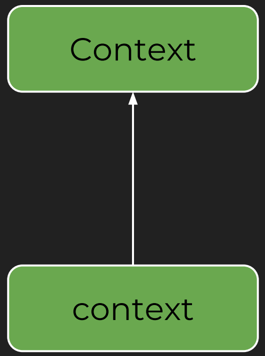
Hiding implementation details

Use interfaces to hide implementation details:

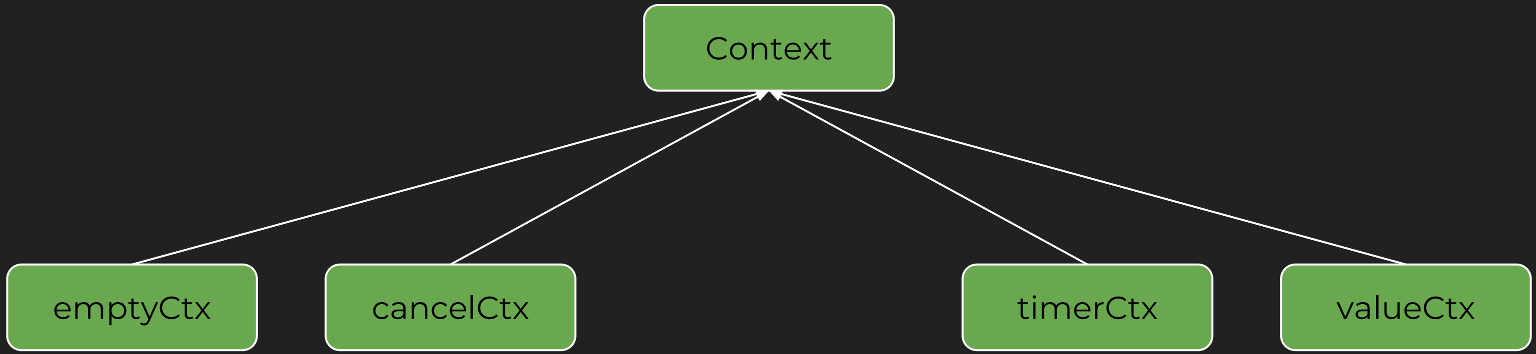
- decouple implementation from API
- easily switch between implementations / or provide multiple ones

context.Context

satisfying the Context interface



satisfying the Context interface



interfaces hide implementation details

call dispatch

f.Do()

call dispatch

Concrete types: static

- known at compilation
- very efficient
- can’t intercept

Abstract types: dynamic

- unknown at compilation
- less efficient
- easy to intercept

interfaces: dynamic dispatch of calls

```
type Client struct {
        Transport RoundTripper
        …
}


type RoundTripper interface {
        RoundTrip(*Request) (*Response, error)
}
```

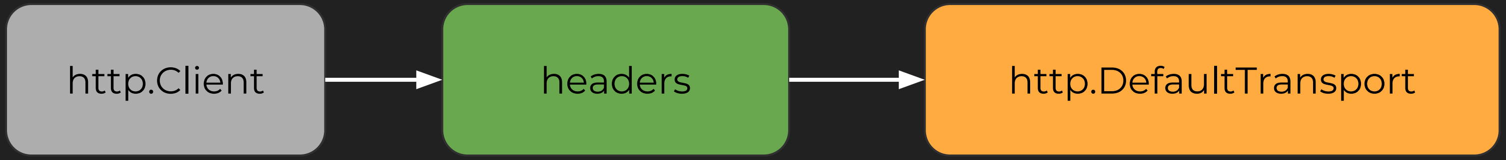

http.Client
http.DefaultTransport

interfaces: dynamic dispatch of calls

```
type headers struct {
    rt  http.RoundTripper
    v   map[string]string
}

func (h headers) RoundTrip(r *http.Request) *http.Response {
    for k, v := range h.v {
        r.Header.Set(k, v)
    }
    return h.rt.RoundTrip(r)
}
```

interfaces: dynamic dispatch of calls

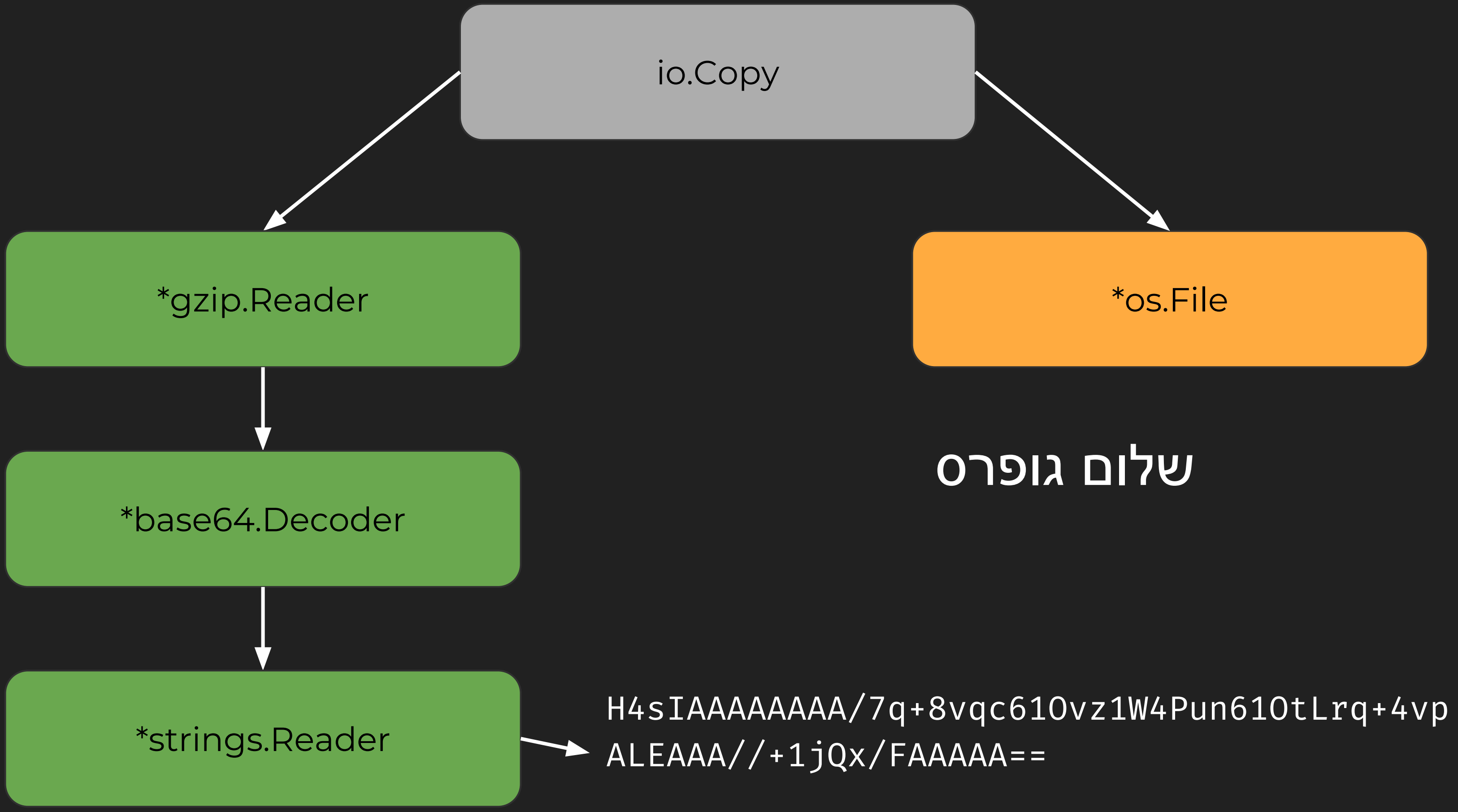
```
c := &http.Client{
    Transport: headers{
        rt:  http.DefaultTransport,
        v:   map[string]string{“foo”: “bar”},
    },
}

res, err := c.Get(“http://golang.org”)
```


http.Client
headers
http.DefaultTransport

chaining interfaces

Chaining interfaces

```
const input =
`H4sIAAAAAAAA/7q+8vqc61Ovz1W4Pun61OtLrq+4vpALEAAA//+1jQx/FAAAAA==`

var r io.Reader = strings.NewReader(input)

r = base64.NewDecoder(base64.StdEncoding, r)

r, err := gzip.NewReader(r)

if err != nil {log.Fatal(err) }

io.Copy(os.Stdout, r)
```



io.Copy
*gzip.Reader
*os.File
*base64.Decoder
שלום גופרס
*strings.Reader
H4sIAAAAAAAA/7q+8vqc61Ovz1W4Pun61OtLrq+4vp
ALEAAA//+1jQx/FAAAAA==

interfaces are interception points

why do we use interfaces?

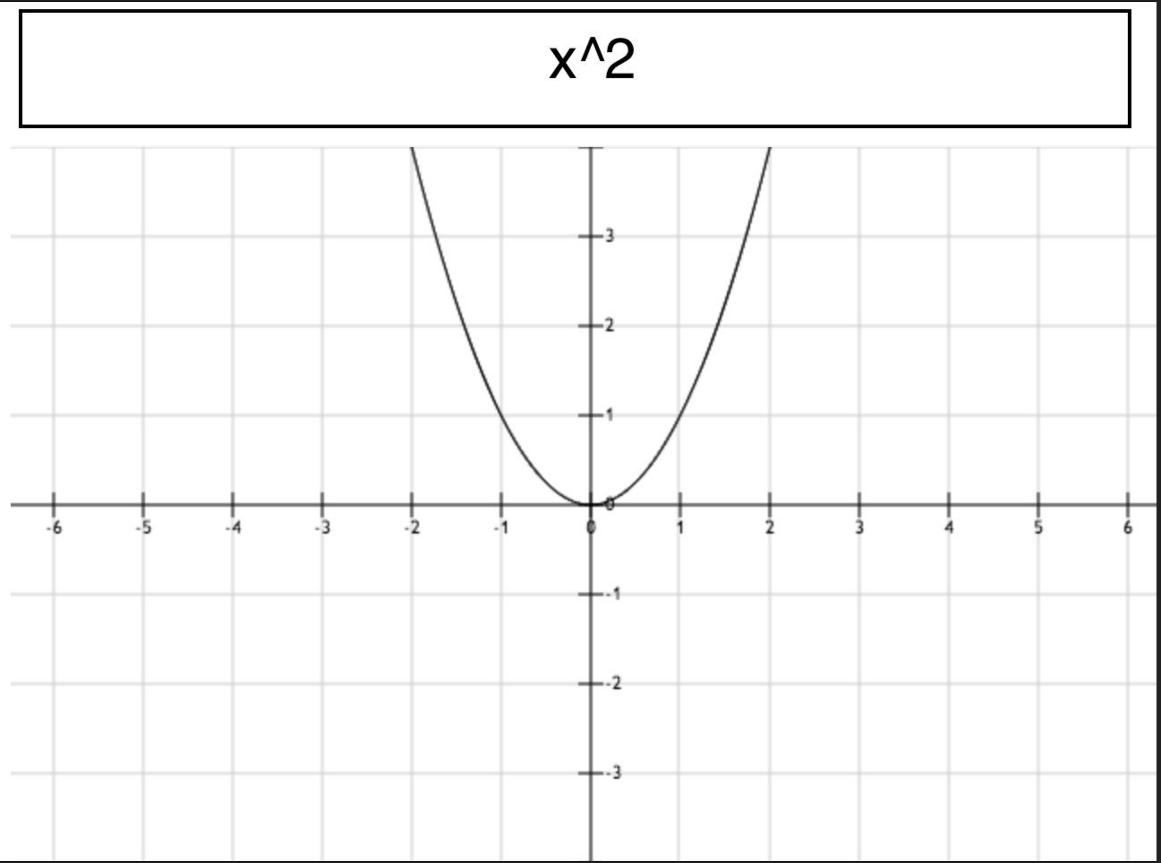
- writing generic algorithms
- hiding implementation details
- providing interception points

so … what’s new?

implicit interface satisfaction

no “implements”

funcdraw



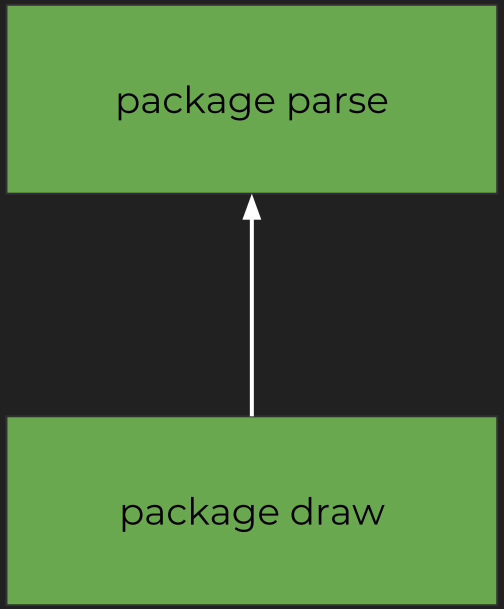
Two packages: parse and draw

```
package parse


func Parse(s string) *Func


type Func struct { … }

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

Two packages: parse and draw

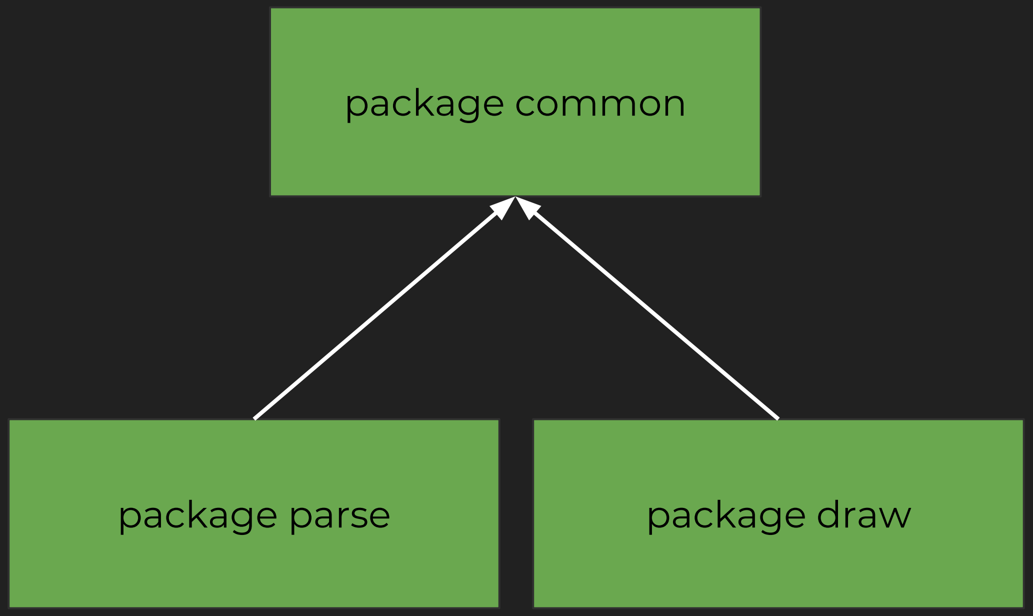
```
package draw

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

But, how do I know what satisfies what, then?

guru

a tool for answering questions about Go source code.

```
File  Edit  Options  Buffers  Tools  Index  Guru  Go  Help
}

type handler chan int

func (h handler) ServeHTTP(w http.ResponseWriter, req *http.Request) {
        w.Header().Set("Content-type", "text/plain")
        fmt.Fprintf(w, "%s: you are visitor #%d", req.URL, <-h)
}
-:---  example.go     Bot L27     (Go)
…/net/http/server.go  interface type net/http.ResponseWriter
…t/http/h2_bundle.go     is implemented by pointer type *net/http.http2responseWriter
…tp/filetransport.go     is implemented by pointer type *net/http.populateResponse
…/net/http/server.go     is implemented by pointer type *net/http.response
…/net/http/server.go     is implemented by pointer type *net/http.timeoutWriter
…van/go/src/io/io.go     implements io.Writer

Go guru finished at Fri Jul  8 12:54:33

U:%*-  *go-guru-output*    All L9     (Go guru:exit [0])
```

http://golang.org/s/using-guru

the super power of Go interfaces

type assertions

type assertions from interface to concrete type

```go
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

```go
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

Don’t just check errors, handle them gracefully

Go Proverb

Dave Cheney - GopherCon 2016

the Context interface

```
type Context interface {
        Done() <-chan struct{}
        Err() error
        Deadline() (deadline time.Time, ok bool)
        Value(key interface{}) interface{}
}

var Canceled, DeadlineExceeded error
```

errors in context

```
var Canceled = errors.New("context canceled")
```

errors in context

```
var Canceled = errors.New("context canceled")

var DeadlineExceeded error = deadlineExceededError{}
```

errors in context

```
var Canceled = errors.New("context canceled")

var DeadlineExceeded error = deadlineExceededError{}
```

errors in context

```
var Canceled = errors.New("context canceled")

var DeadlineExceeded error = deadlineExceededError{}

type deadlineExceededError struct{}

func (deadlineExceededError) Error() string   { return "..." }
func (deadlineExceededError) Timeout() bool   { return true }
func (deadlineExceededError) Temporary() bool { return true }
```

errors in context

```
var Canceled = errors.New("context canceled")

var DeadlineExceeded error = deadlineExceededError{}

type deadlineExceededError struct{}

func (deadlineExceededError) Error() string   { return "..." }
func (deadlineExceededError) Timeout() bool   { return true }
func (deadlineExceededError) Temporary() bool { return true }
```

errors in context

```
var Canceled = errors.New("context canceled")

var DeadlineExceeded error = deadlineExceededError{}

type deadlineExceededError struct{}

func (deadlineExceededError) Error() string   { return "..." }
func (deadlineExceededError) Timeout() bool   { return true }
func (deadlineExceededError) Temporary() bool { return true }
```

errors in context

```
if tmp, ok := err.(interface { Temporary() bool }); ok {

    if tmp.Temporary() {
        // retry
    } else {
        // report
    }

}
```

use type assertions to classify errors

type assertions as evolution mechanism

Adding methods to an interface breaks backwards compatibility.

```
type ResponseWriter interface {
    Header() Header
    Write([]byte) (int, error)
    WriteHeader(int)
}
```

How could you add one more method without breaking anyone's code?

type assertions as evolution mechanism

Step 1: add the method to your concrete type implementations

Step 2: define an interface containing the new method

Step 3: document it

http.Pusher

```
type Pusher interface {
        Push(target string, opts *PushOptions) error
}



func handler(w http.ResponseWriter, r *http.Request) {
    if p, ok := w.(http.Pusher); ok {
        p.Push(“style.css”, nil)
    }
}
```

use type assertions to maintain backwards compatibility

In conclusion

In conclusion

Interfaces provide:

- generic algorithms
- hidden implementation
- interception points

In conclusion

Interfaces provide:

- generic algorithms
- hidden implementation
- interception points

Implicit satisfaction:

- break dependencies

In conclusion

Interfaces provide:

- generic algorithms
- hidden implementation
- interception points

Implicit satisfaction:

- break dependencies

Type assertions:

- to extend behaviors
- to classify errors
- to maintain compatibility

what about generics?

Type Parameters

A new draft was proposed to support generic programming in Go.

It adds type parameters to:

- functions
- types

Type Parameters on functions

```
func Print(type T)(vs []T) {

    for _, v := range vs {
        fmt.Print(v)
    }

}
```

Type Parameters on types

Interfaces are not always enough for container types:

- container/heap
  - Provides only high level functions (Push, Pop, etc)
  - Requires the user to implement the Heap interface
- container/ring
  - Provides a type Element containing a interface{}
  - Requires constant type conversions, loses compile time checks.

Type Parameters on types

```
type Stack(type T) []T


func (s Stack(T)) Push(v T) Stack(T) { … }
func (s Stack(T)) Top() T            { … }
func (s Stack(T)) Pop() Stack(T)     { … }
func (s Stack(T)) Empty() bool       { … }
```

is this enough?

A generic Max function

```
func Max(type T)(vs []T) T {

    var max T
    for _, v := range vs {
        if v > max {
            max = v
        }
    }
    return max
}
```

A generic Max function

```
Max(int)([]int{1, 3, 2})                          // 3

Max(float64)([]float64{1.0, 3.0, 2.0})            // 3.0

Max(string)([]string{"a", "c", "b"})              // "c"

Max(bool)([]bool{true, false})                    // ?

Max([]byte)([][]byte{{1, 2}, {2, 1}})             // ?
```

type contracts

Type Contracts

Type contracts provides a set of constraints on types.

Sounds familiar?

They look like functions:

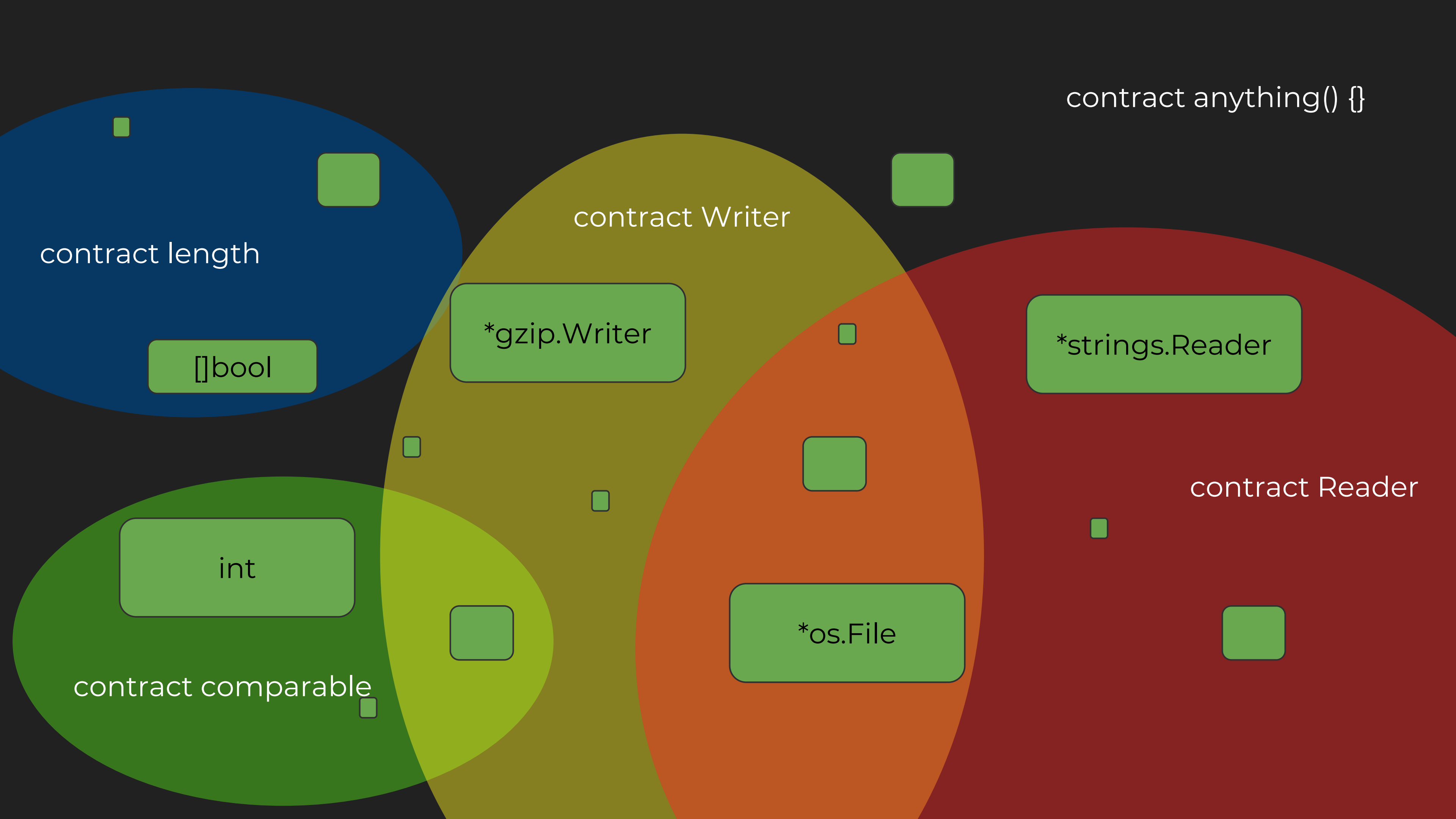
where each operation becomes a constraint.

A comparable type

```
contract comparable(v T) {
    var b bool = v < v
}

func Max(type T comparable)(vs []T) T { … }


Max([]int)([]int{1, 3, 2})              // 3
Max([]string)([]string{"a", "c", "b"})  // "c"
```

A type with length

```
contract length(v T) {
    var n int = len(v)
}

func Length(type T length)(x T) int { return len(x) }


Length([]int)([]int{1, 2, 3})               // 3
Length(map[int]int)(map[int]int{1: 1})      // 1
```


contract anything() {}
contract length
contract Writer
[]bool
*gzip.Writer
*strings.Reader
contract Reader
int
*os.File
contract comparable

so … interfaces or contracts?

Interfaces vs Contracts

Interfaces and contracts define **sets of types**.

Interfaces constrain on methods, contracts on anything.

→ all interfaces can be represented as contracts

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

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

Interfaces vs Contracts

Interfaces constrain on methods, contracts on anything.

→ not all contracts can be represented as interfaces

```
contract comparable(v T) {
    var b bool = v < v
}

contract comparable(v T) {
    var b bool = v.Equal(v)
}
```

putting the con back in contract

Contracts are great, but

Once instantiated they are concrete types, so unlike interfaces:

- They hide no implementation
- They don’t provide dynamic dispatching

This is has some good effects:

- the type system once compiled doesn’t change.
- there’s no change to the reflect package.

contracts are* too *smart*

*probably

Print

Total: **42 pages**

Cancel Save

Destination Save as PDF

Change...

Pages All

e.g. 1-5, 8, 11-13

Layout Portrait

More settings

Print using system dialog... (⌥⌘P)

Open PDF in Preview

Contracts — Draft Design

Ian Lance Taylor
Robert Griesemer
August 27, 2018

Abstract

We suggest extending the Go language to add optional type parameters to types and functions. Type parameters may be constrained by contracts: they may be used as ordinary types that only support the operations described by the contracts. Type inference via a unification algorithm is supported to permit omitting type arguments from function calls in many cases. Depending on a detail, the design can be fully backward compatible with Go 1.

For more context, see the generics problem overview.

Background

There have been many requests to add additional support for generic programming in Go. There has been extensive discussion on the issue tracker and on a living document.

There have been several proposals for adding type parameters, which can be found by looking through the links above. Many of the ideas presented here have appeared before. The main new features described here are the syntax and the careful examination of contracts.

This draft design suggests extending the Go language to add a form of parametric polymorphism, where the type parameters are bounded not by a subtyping relationship but by explicitly defined structural constraints. Among other languages that support parametric polymorphism this design is perhaps most similar to CLU or Ada, although the syntax is completely different. Contracts also somewhat resemble C++ concepts.

This design does not support template metaprogramming or any other form of compile-time programming.

As the term *generic* is widely used in the Go community, we will use it below as a shorthand to mean a function or type that takes type parameters. Don't confuse the term generic as used in this design with the same term in other languages like C++, C#, Java, or Rust; they have similarities but are not always the same.

Design

We will describe the complete design in stages based on examples.

Type parameters

contracts are probably too *smart*

Remember the comparable contract?

```
contract comparable(v T) {
    var b bool = v.Equal(v)
}
```

What is the type of the `Equal` method?

- `func (v  T) Equal(x T) bool`
- `func (v *T) Equal(x T) bool`
- `func (v  T) Equal(x interface{}) bool`

contracts vs interfaces

Interfaces provide:
- generic algorithms
- hidden implementation
- interception points

Implicit satisfaction:
- break dependencies

Type assertions:
- to extend behaviors
- to classify errors
- to maintain compatibility

Contracts **would** provide:
- generic algorithms
- generic types

Libraries for:
- Concurrency patterns
- Functional programming

Remove duplication:
- int, int32, int64 … float32, float64 …
- `bytes` vs `strings`

Conclusion

Interfaces are amazing and I still love them <3

Type parameters are great!

- A library of concurrency patterns?
- Functional programming in Go?

Contracts are interesting, but probably *too smart* for Go.

I hope to see new iterations of the draft simplifying the concepts.

Go is about simplicity

I’d argue contracts are not

תודה!

Thanks, @francesc