understanding the interface

@francesc

what is an interface?

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

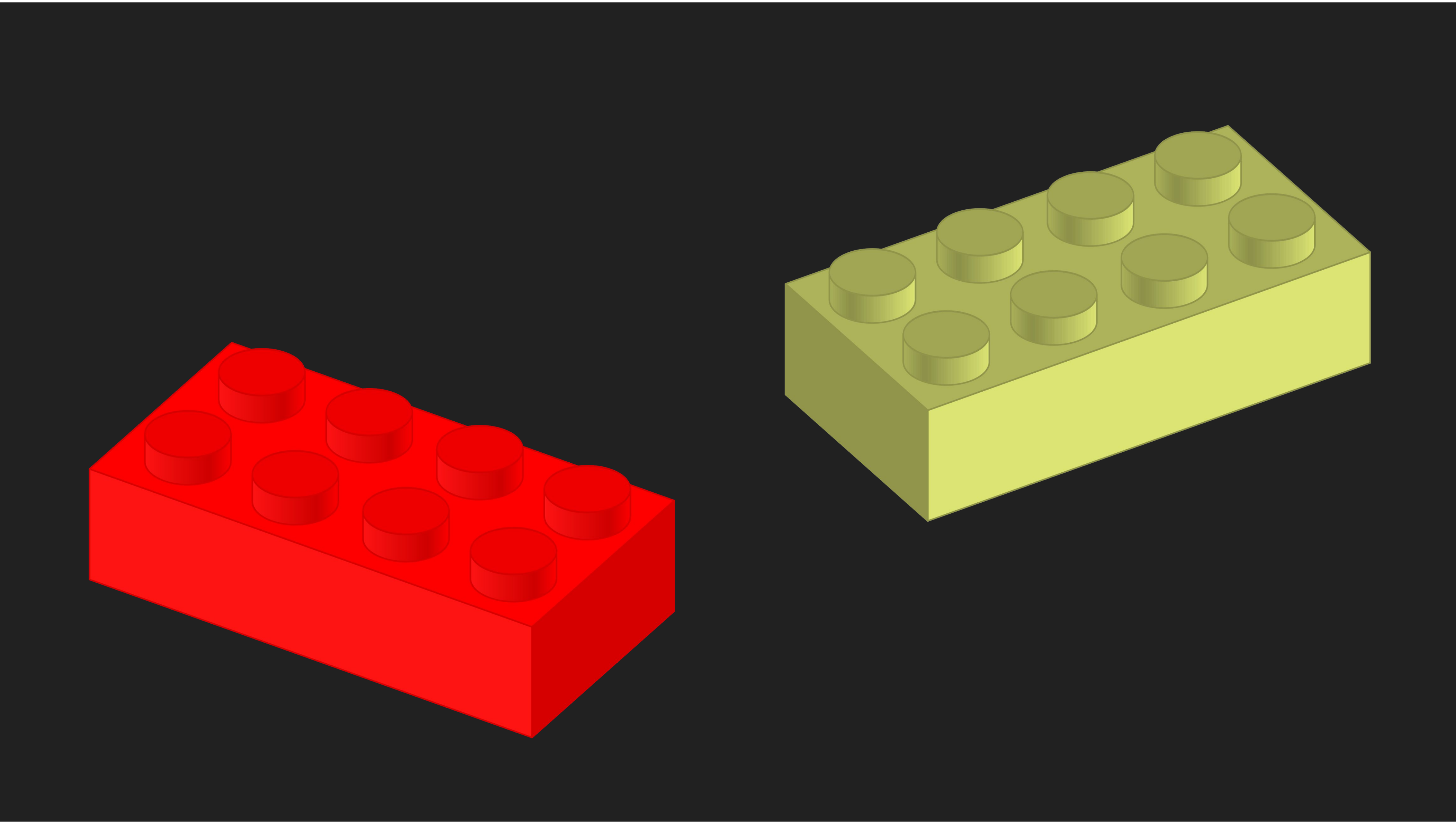
- wikipedia

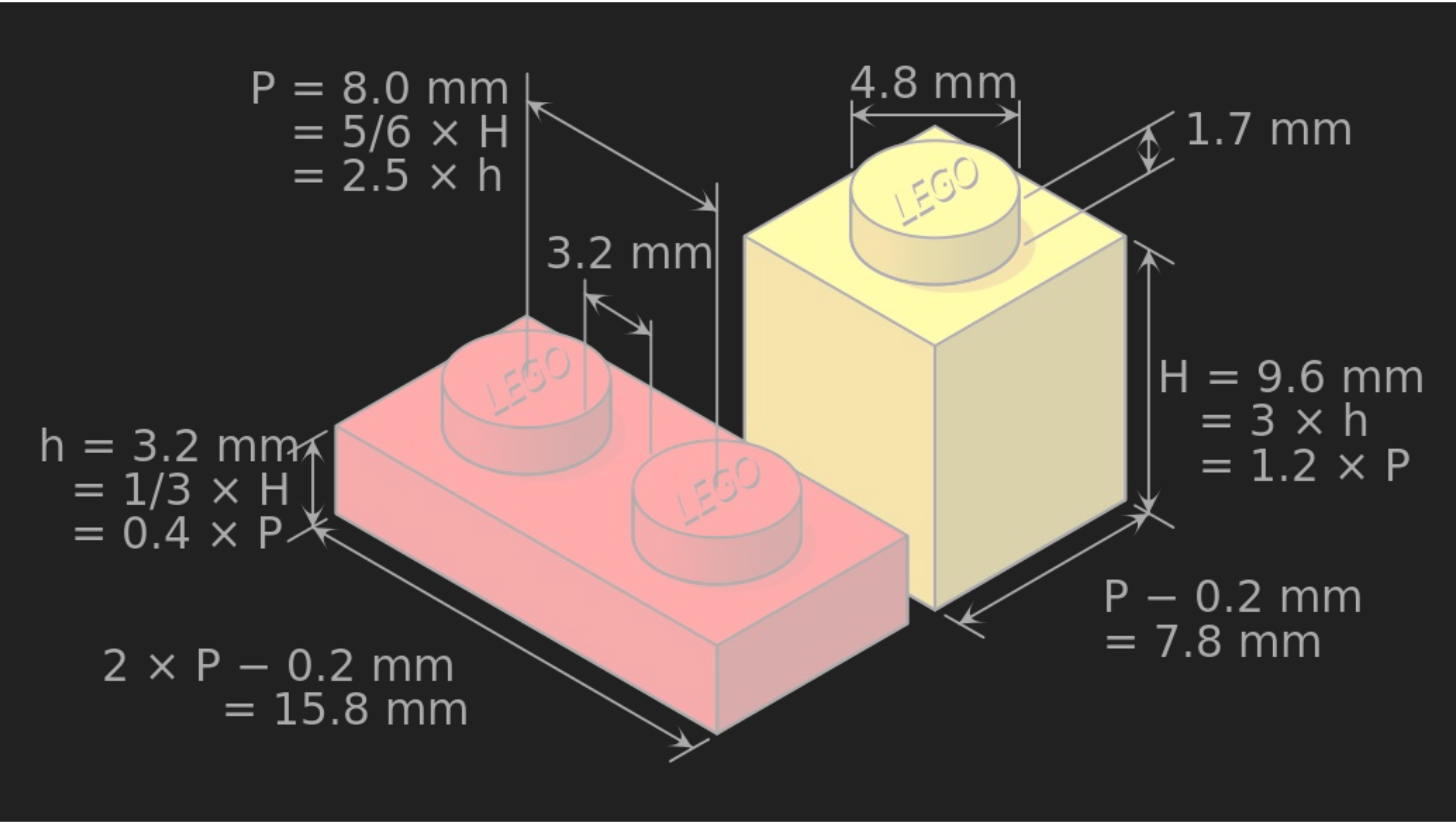
"In object-oriented programming, a protocol or interface is a common means for unrelated objects to communicate with each other"

\- 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

what is a Go interface?

abstract types

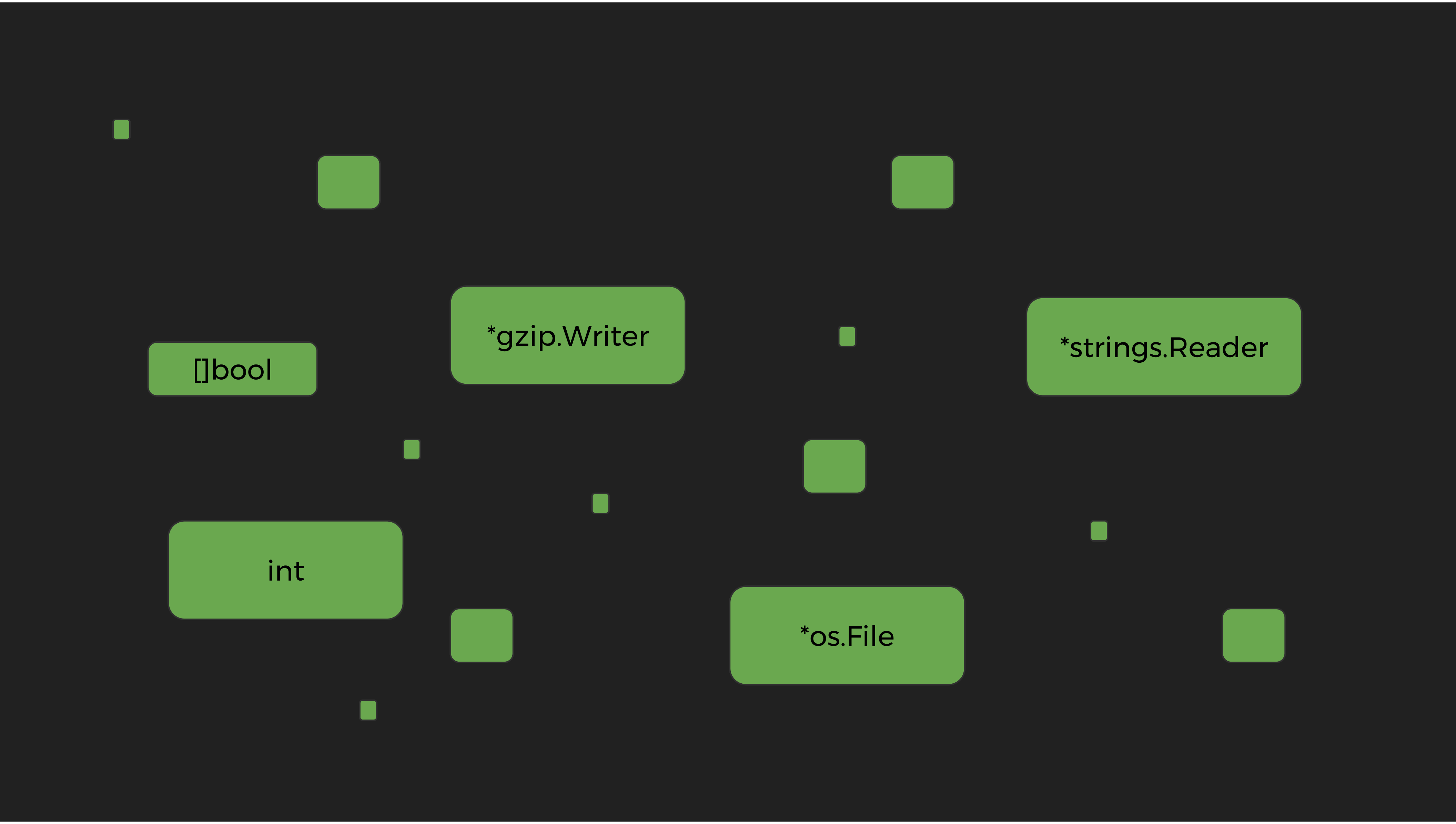
concrete types

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

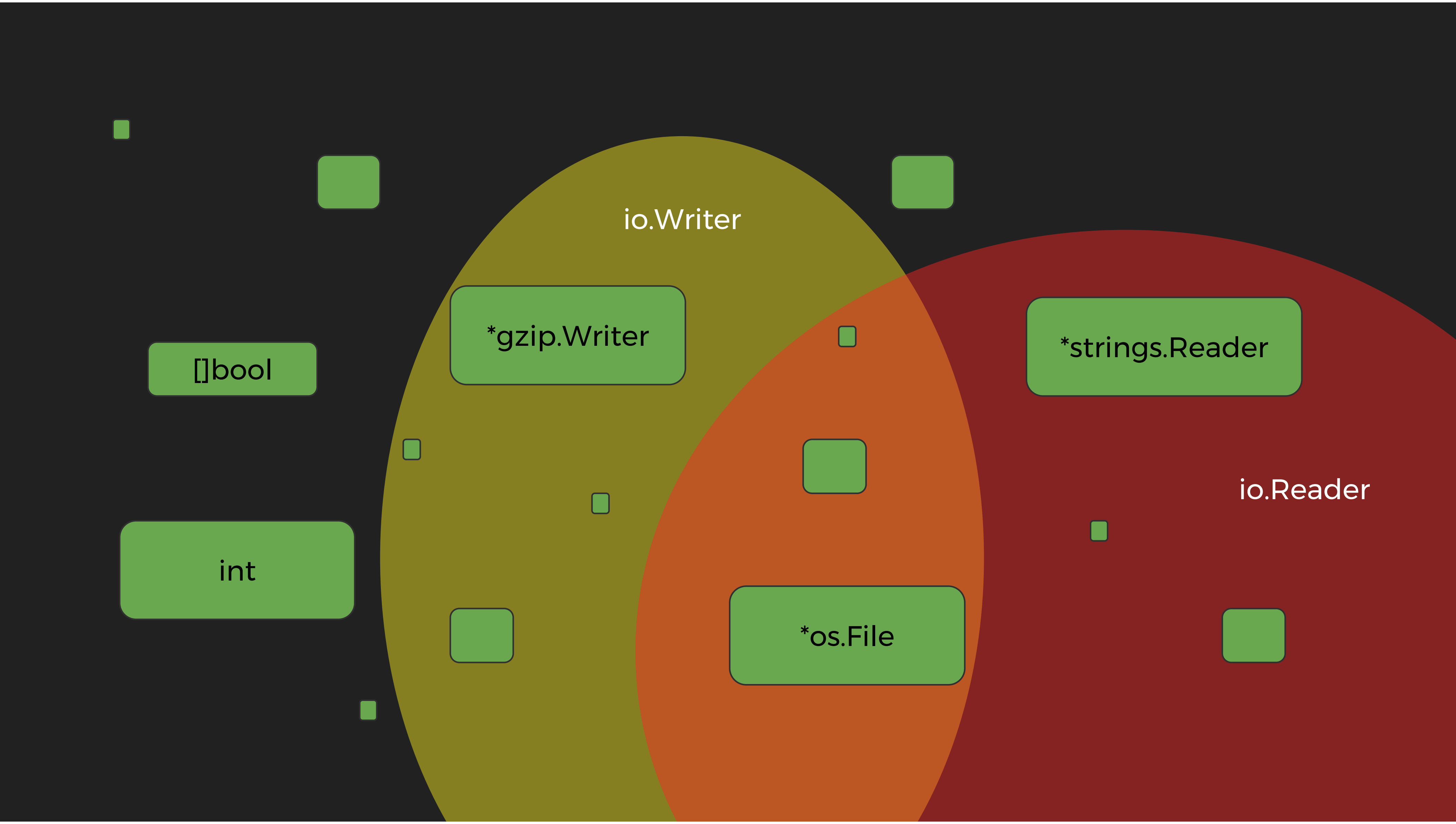
io.Reader | io.Writer | fmt.Stringer

- 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)
}
```

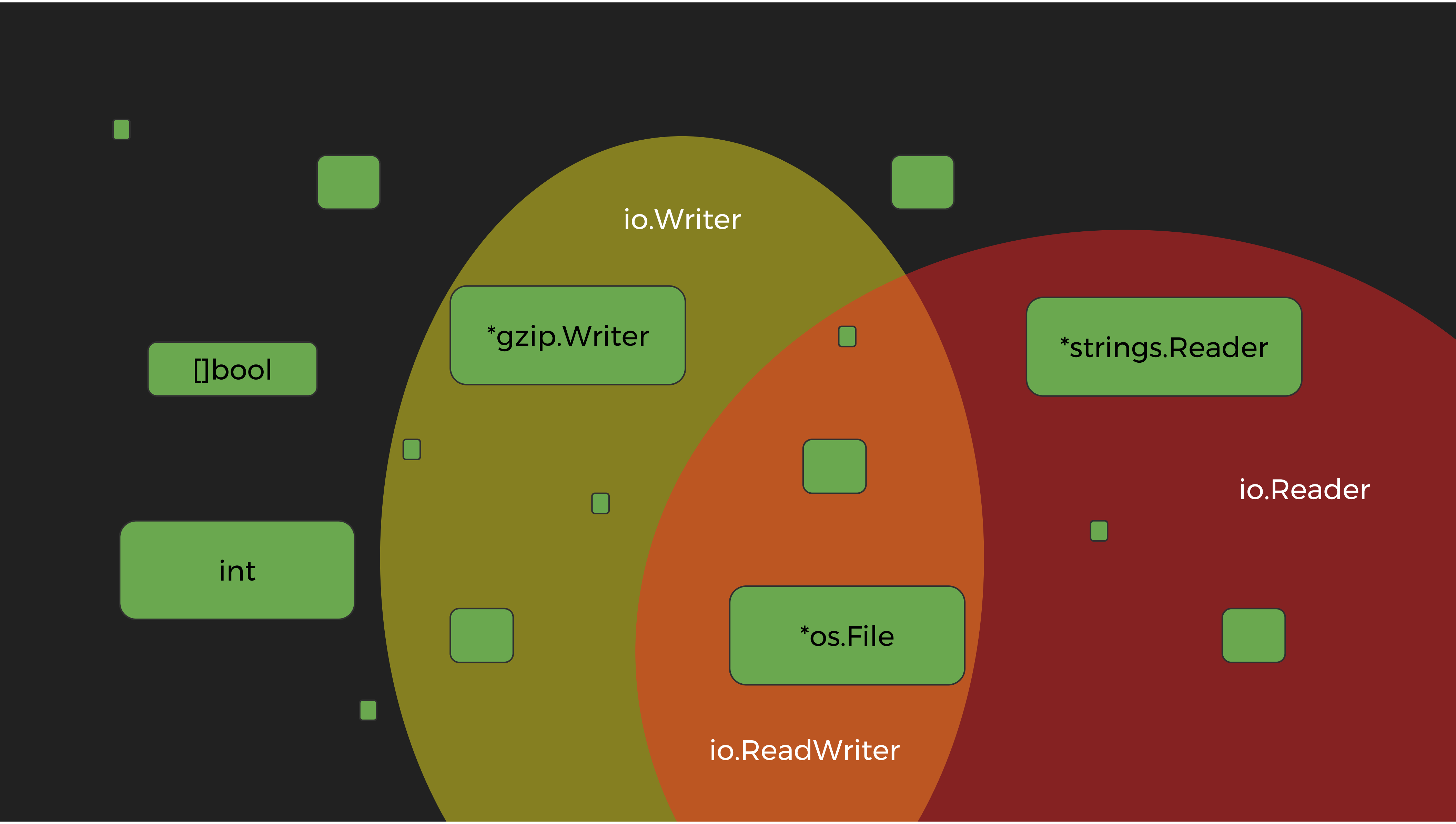

io.Writer
*gzip.Writer
*strings.Reader
[]bool
io.Reader
int
*os.File

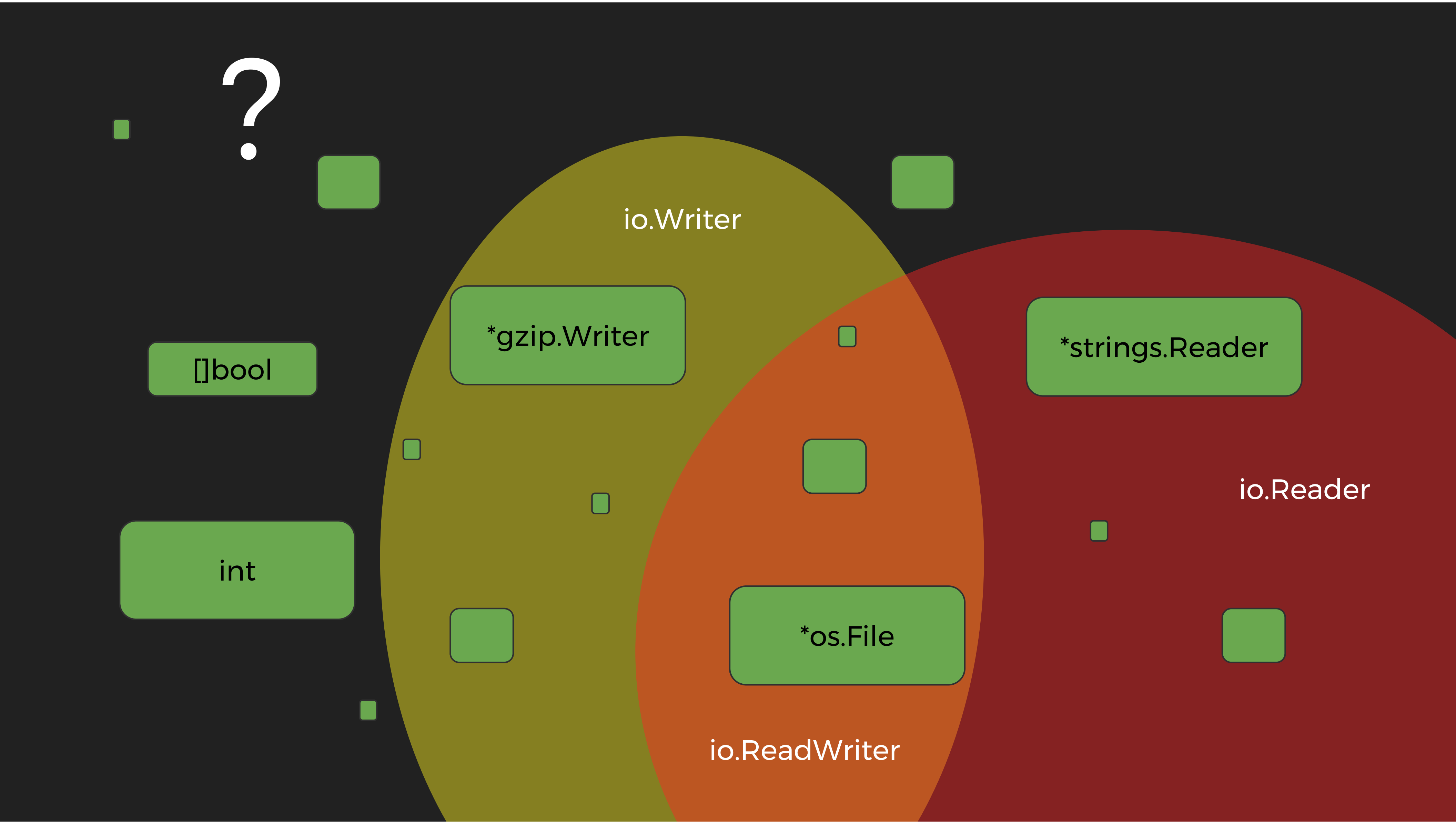
union of interfaces

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

union of interfaces

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

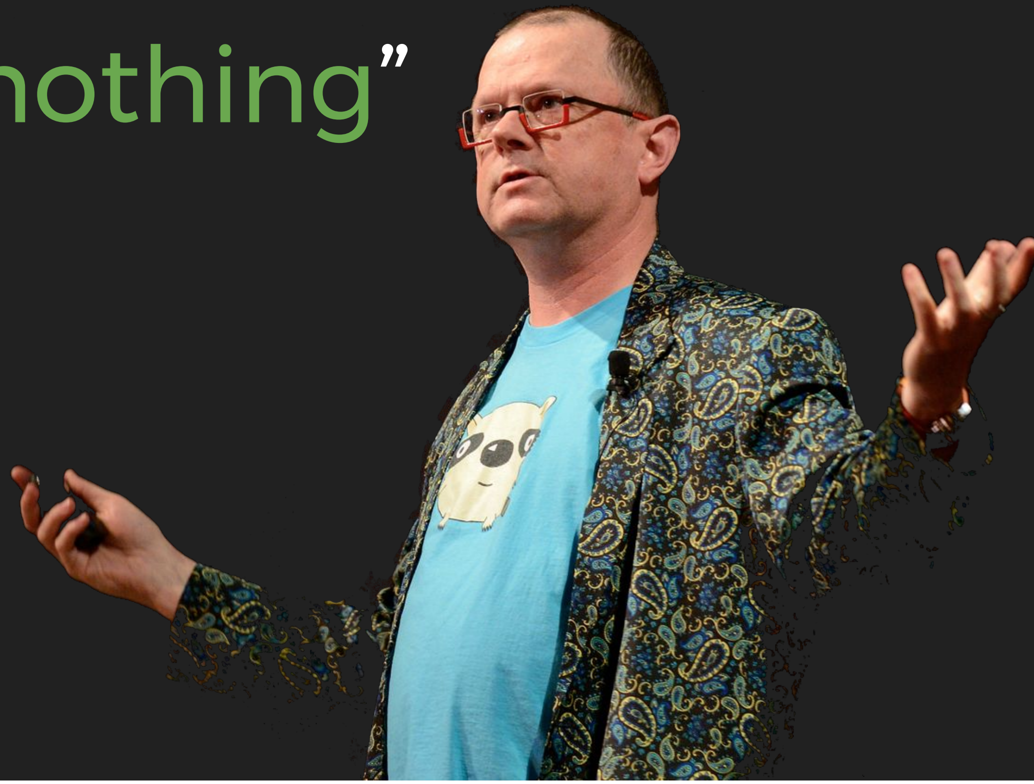

io.Writer
*gzip.Writer
[]bool
*strings.Reader
io.Reader
int
*os.File
io.ReadWriter


?
io.Writer
*gzip.Writer
*strings.Reader
[]bool
int
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)
```
func WriteTo(f *os.File) error
```

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

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

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

```go
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

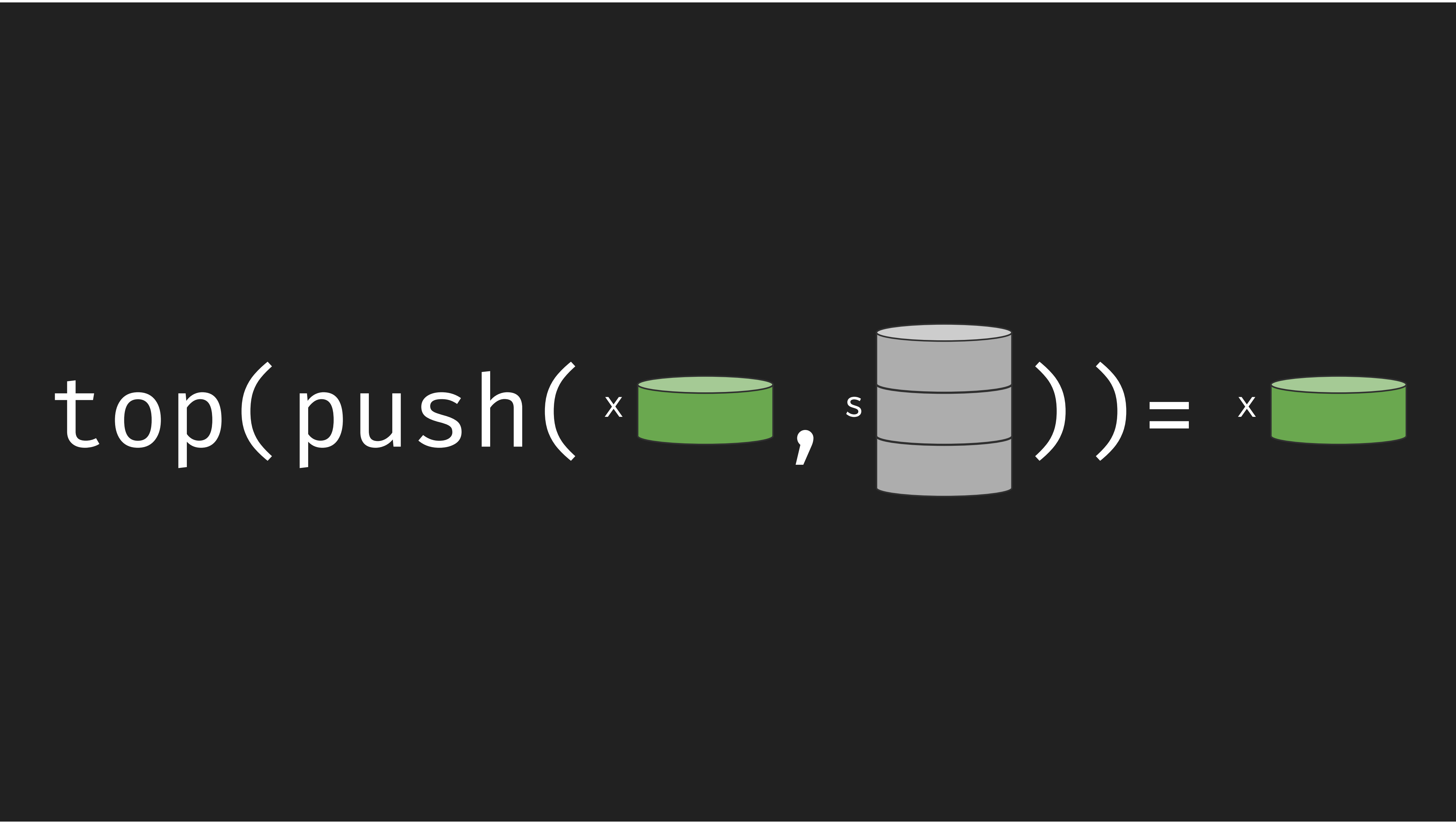
Abstract Data Types

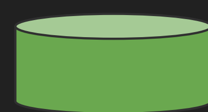
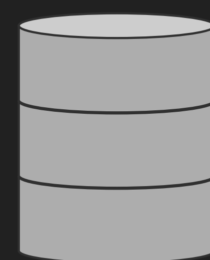
Abstract Data Types

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
    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?

```
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.

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

b) `func New() io.ReadWriteCloser`

c) `func New() io.Writer`

d) `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

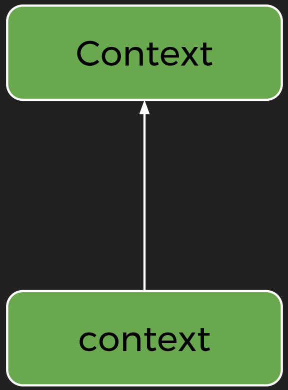
Hiding implementation details

Use interfaces to hide implementation details:

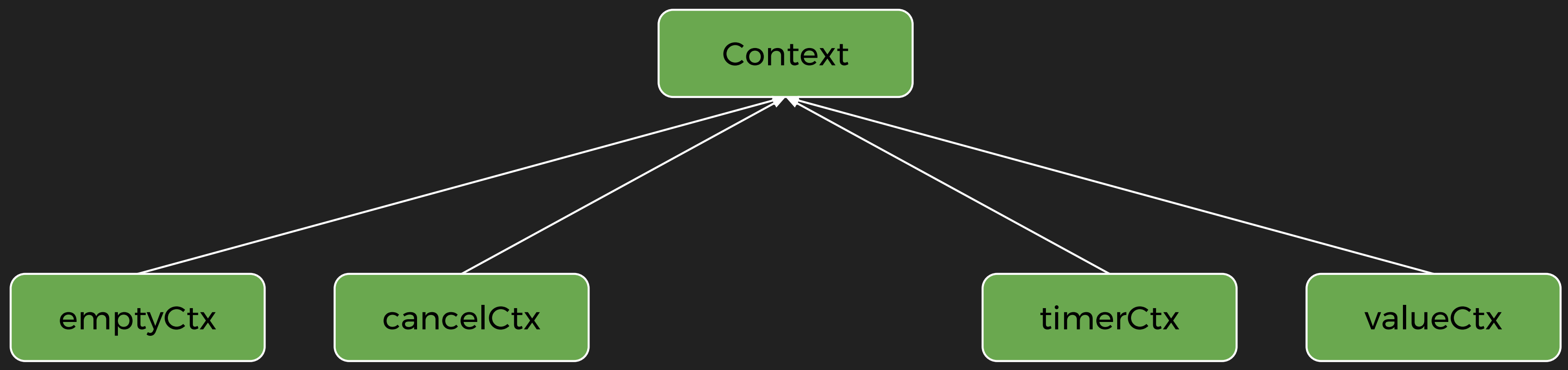
- decouple implementation from API
- easily switch between implementations / or provide multiple ones

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)
}
```

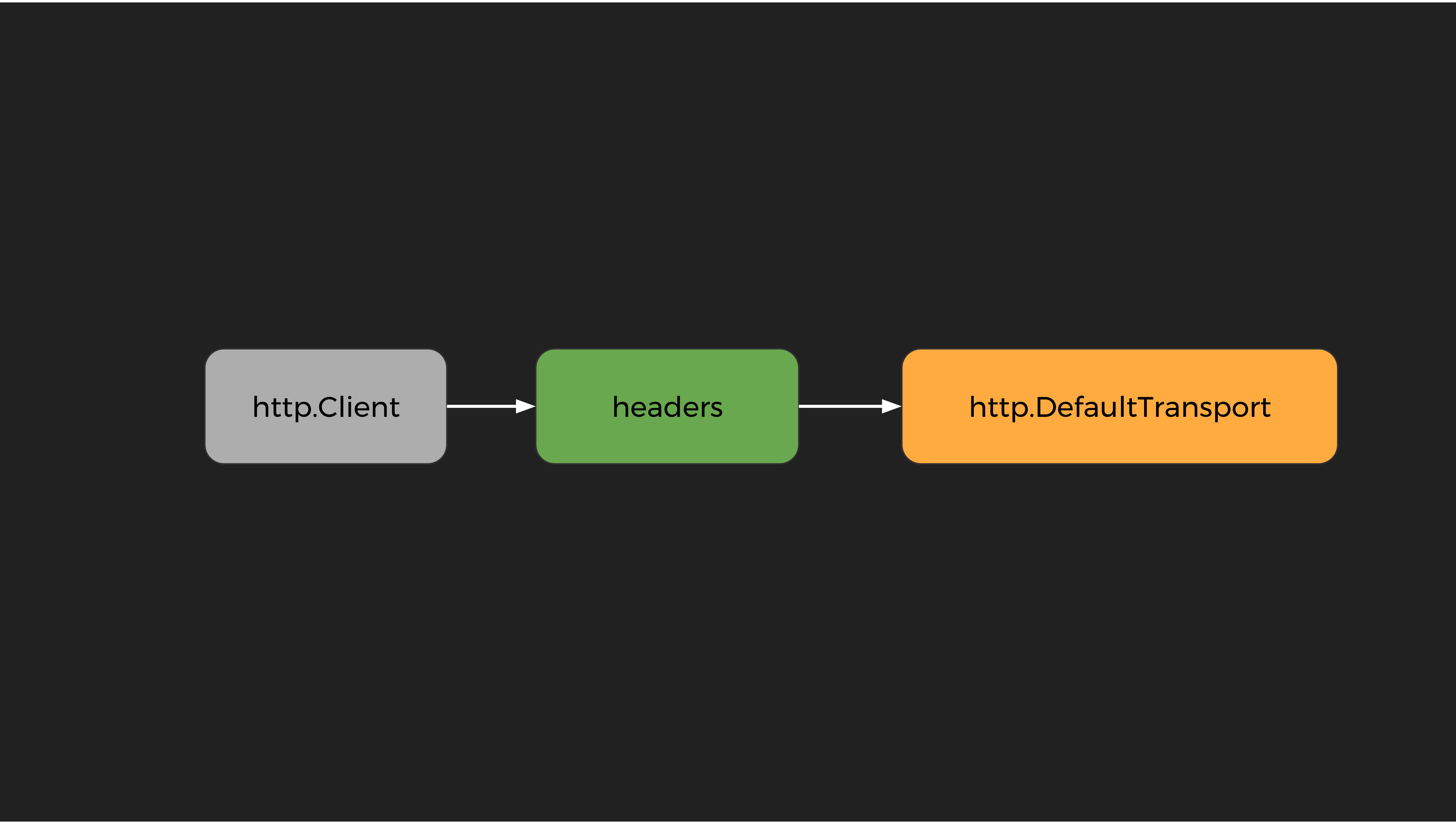

http.Client
http.DefaultTransport

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

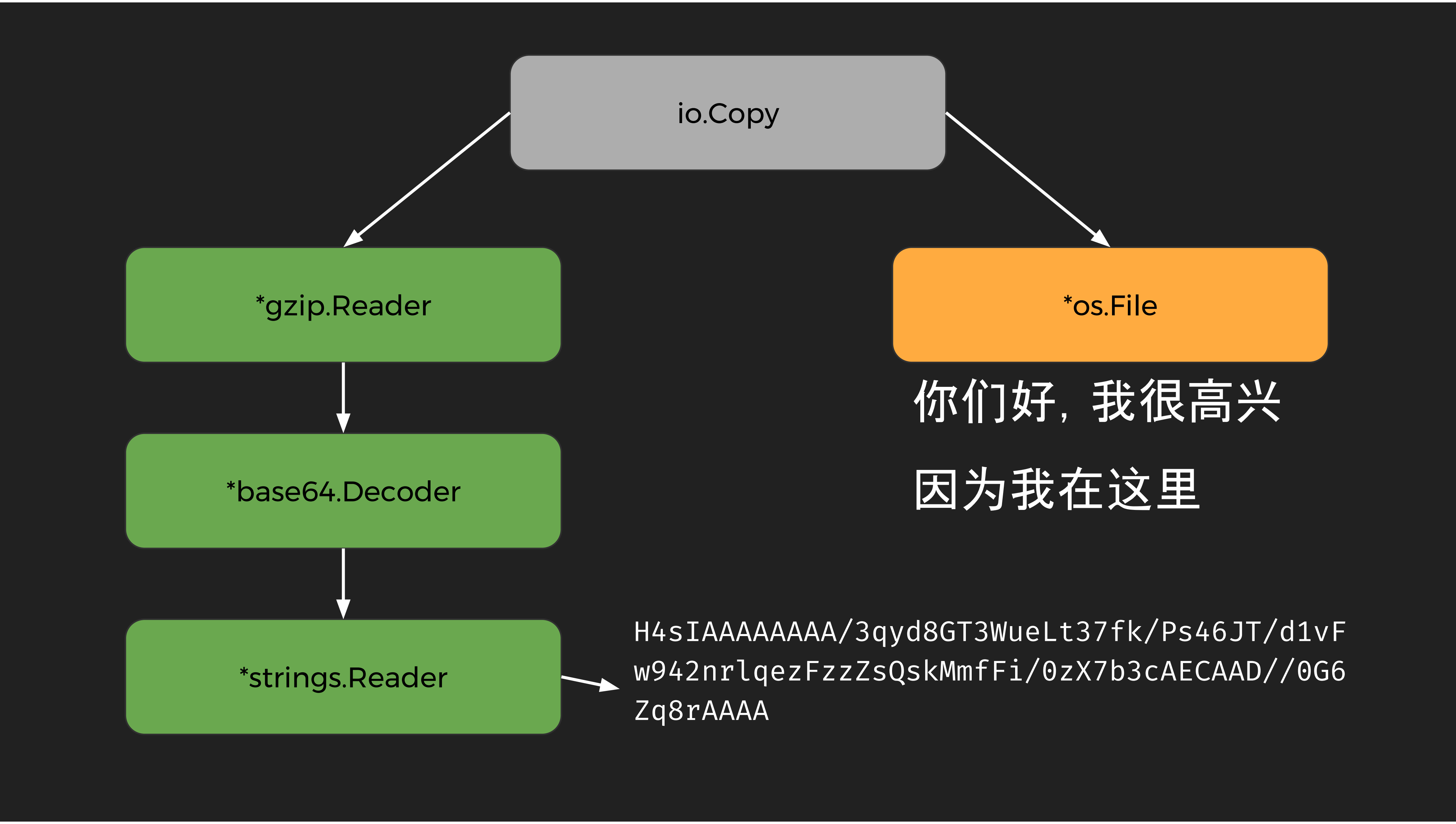
```
c := &http.Client{
    Transport: headerRoundTripper{
        rt:  http.DefaultTransport,
        v:   map[string]string{“foo”: “bar”},
    },
}

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


http.Client
headers
http.DefaultTransport

chaining interfaces

Chaining interfaces

```
const input =
`H4sIAAAAAAAA/3qyd8GT3WueLt37fk/Ps46JT/d1vFw942nrlqezFzzZsQskMmfFi/0zX7b3cAECA
AD//0G6Zq8rAAAA`

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
*base64.Decoder
*strings.Reader
*os.File
你们好，我很高兴
因为我在这里
H4sIAAAAAAAA/3qyd8GT3WueLt37fk/Ps46JT/d1vF
w942nrlqezFzzZsQskMmfFi/0zX7b3cAECAAD//0G6
Zq8rAAAA

interfaces are interception points

why do we use interfaces?

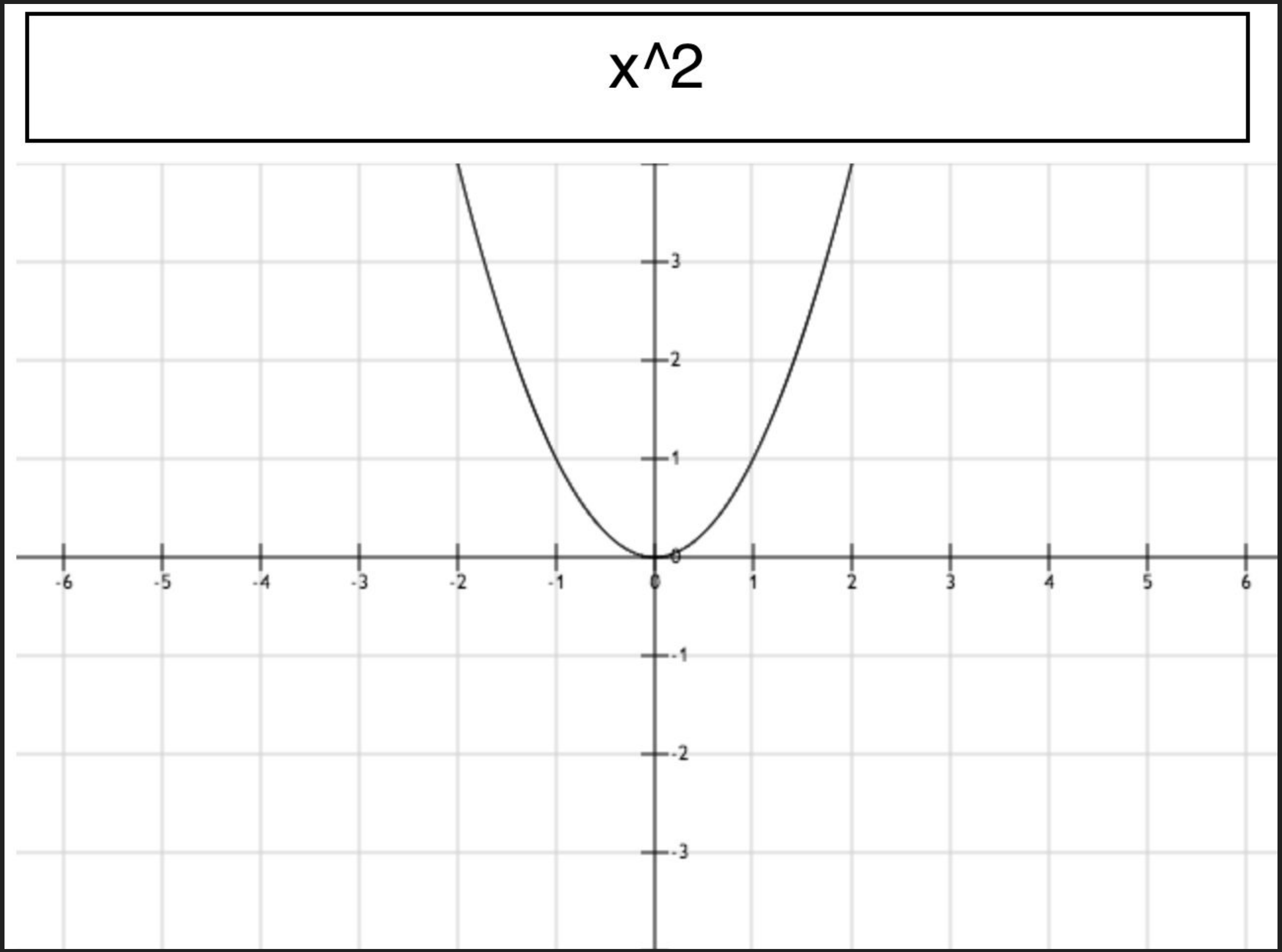
- writing generic algorithms
- hiding implementation details
- providing interception points

so … what’s new?

implicit interface satisfaction

no “implements”

funcdraw



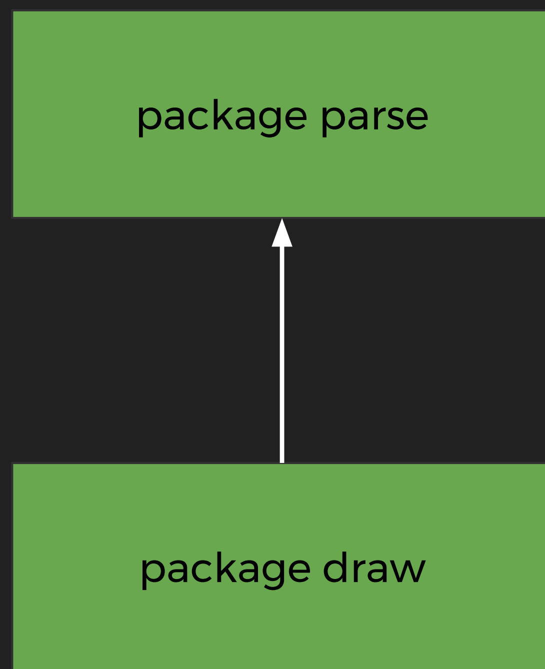
Two packages: parse and draw

```
package parse


func Parse(s string) *Func


type Func struct { … }

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

Two packages: parse and draw

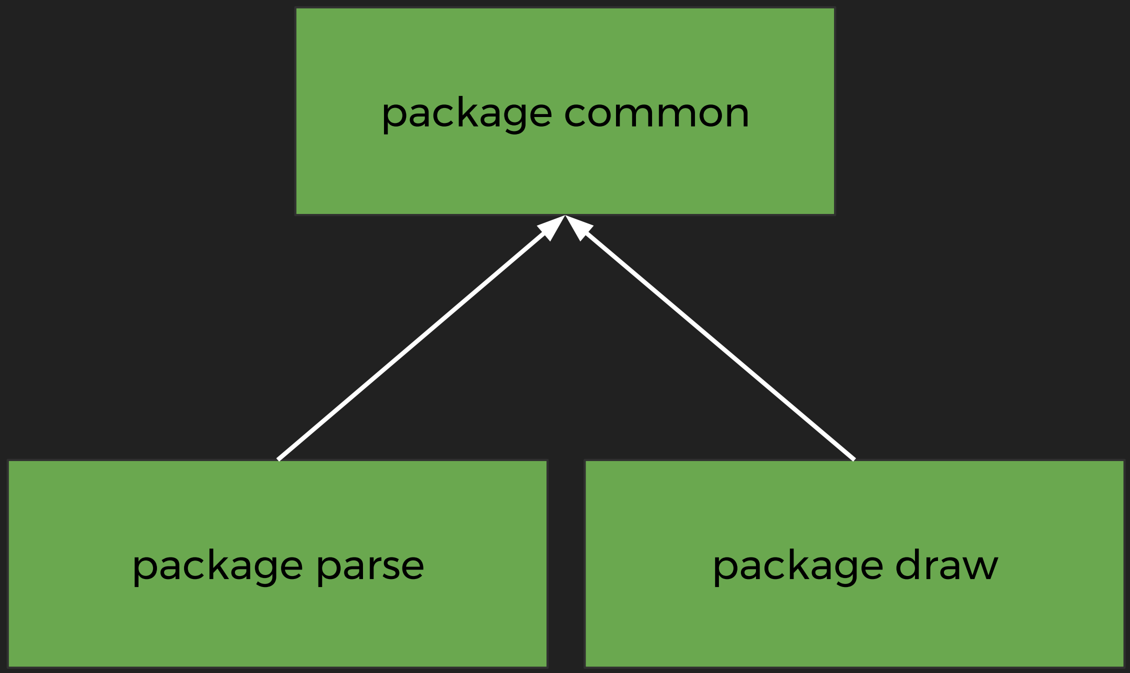
```
package draw

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))
    }
    …
}
```

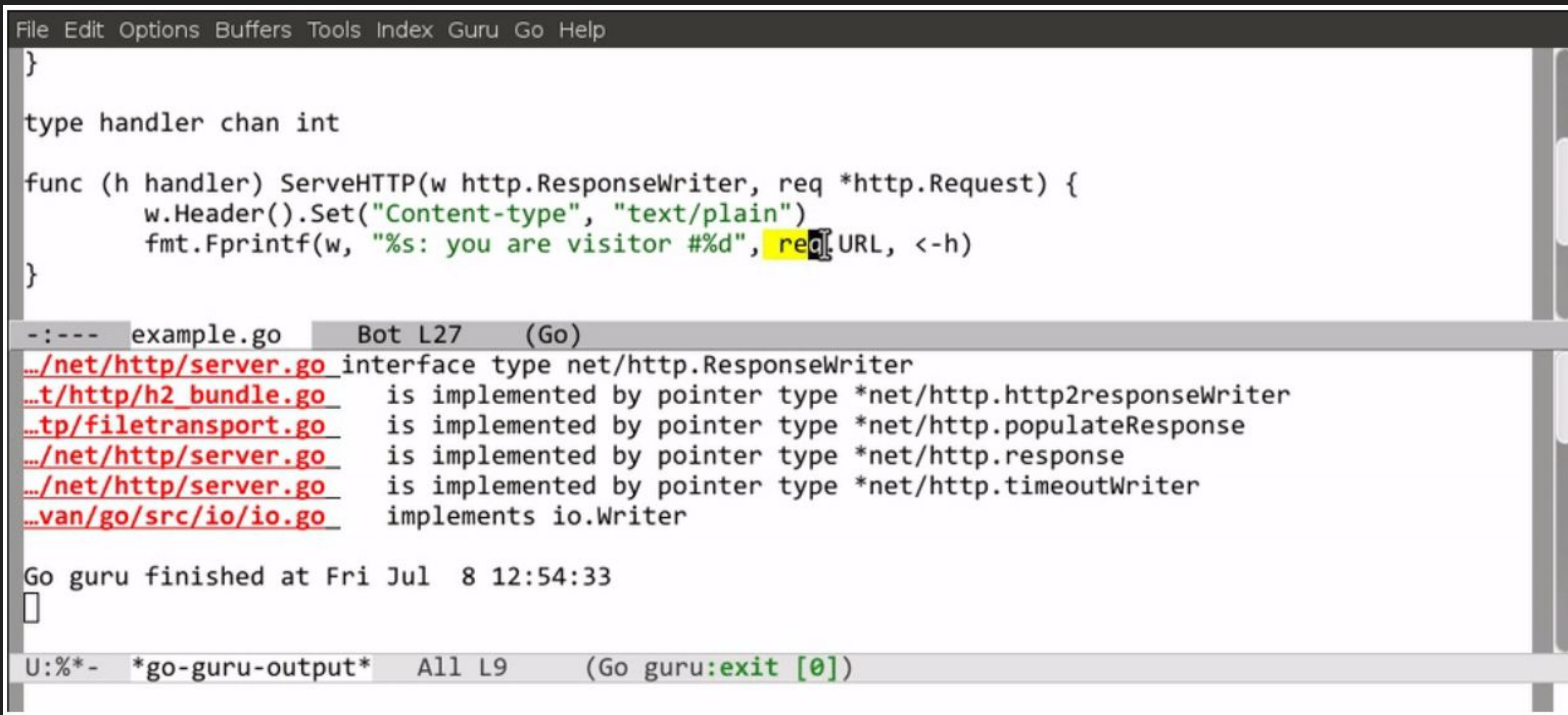
interfaces can break dependencies

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.



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

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{}) {

    select v.(type) {

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

    Default:


    }
}
```

type assertions from interface to concrete type

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

    select 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”)

    }
}
```

avoid abstract to concrete assertions

type assertions from interface to interface

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

runtime checks interface to concrete type

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

    select v.(type) {

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

    Default:


    }
}
```

runtime checks interface to concrete type

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

    select s := v.(type) {

    case fmt.Stringer:      // s is of type int
        fmt.Println(s.String())

    default:                // t 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
- json.Marshaler/Unmarhsaler
- ...

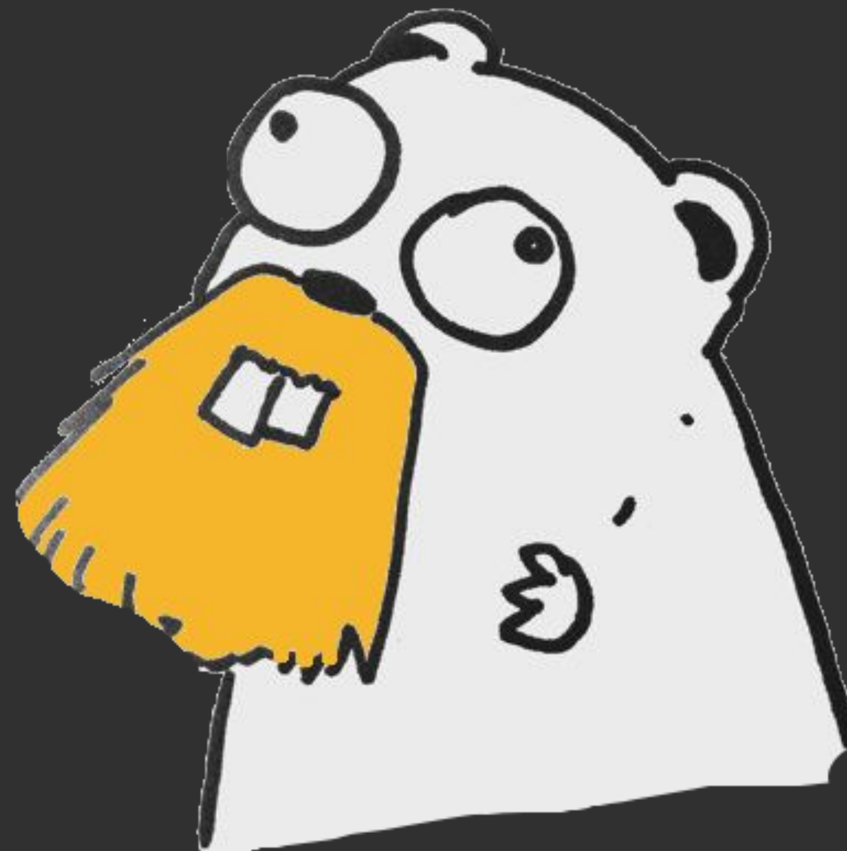
and adapt their behavior accordingly.

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 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

谢谢

Thanks, @francesc