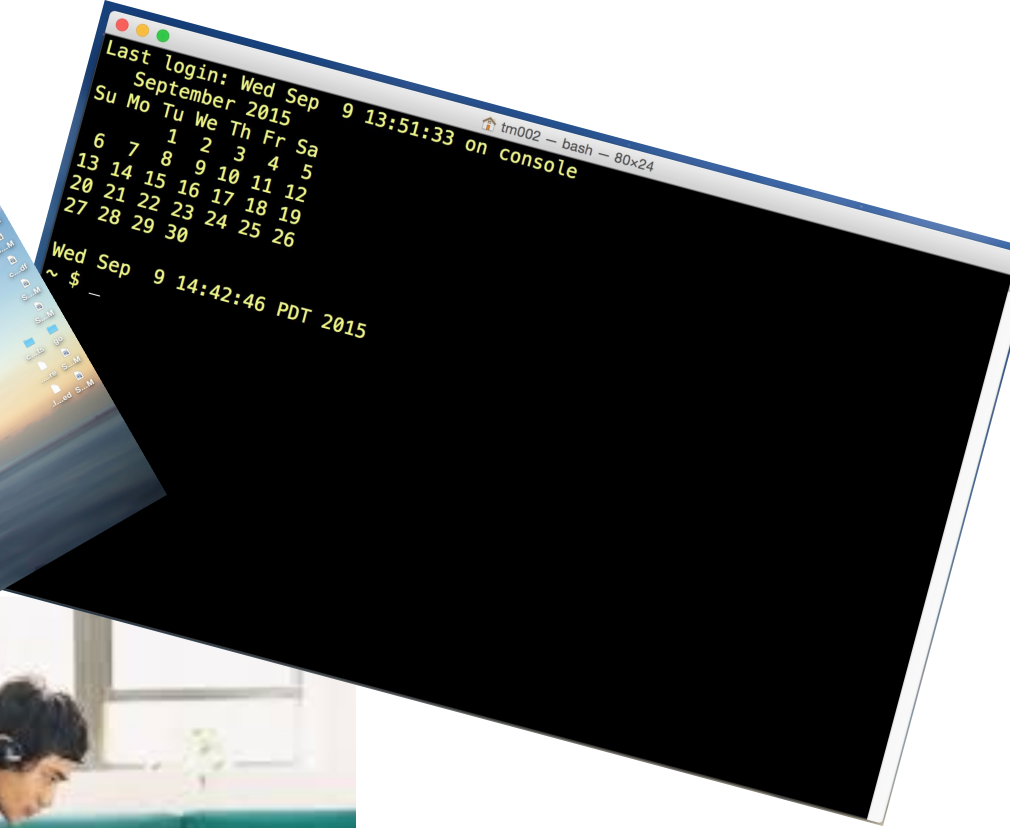
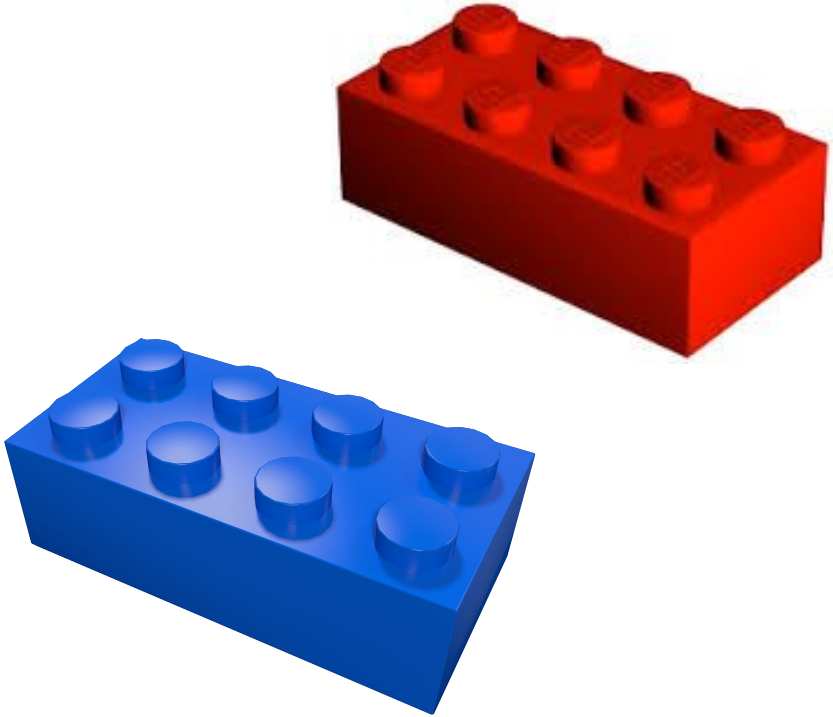
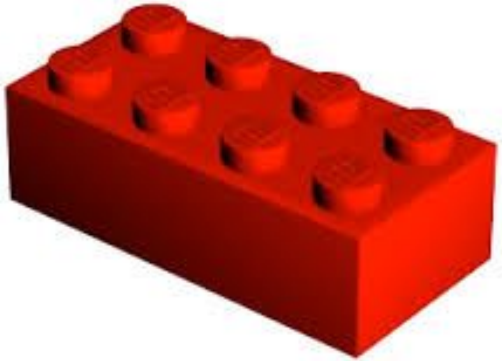
interfaces

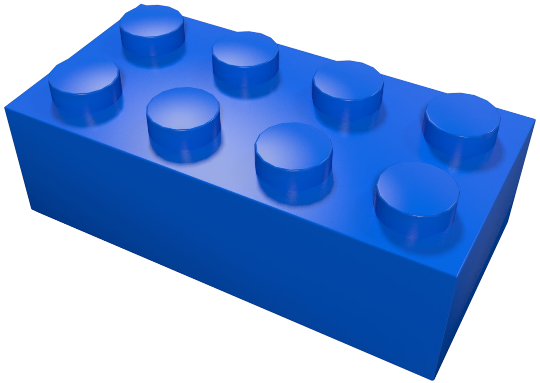
WebStorm
File
Edit
View
Navigate
Code
Refactor
Run
Tools
VCS
Window
Help
100%
Wed 2:42 PM
S...M
pics
O.jpg
misc
S...M
S...M
S...M
S...M
S...M
S...M
c...df
S...M
S...M
gp
c...ts
S...M
...re
S...M
J...ed



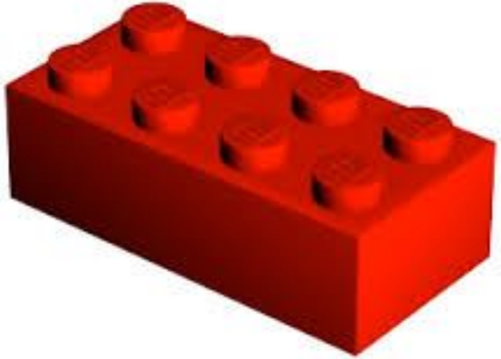
```
Last login: Wed Sep  9 13:51:33 on console
   September 2015
Su Mo Tu We Th Fr Sa
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30

Wed Sep  9 14:42:46 PDT 2015
~ $ _
```

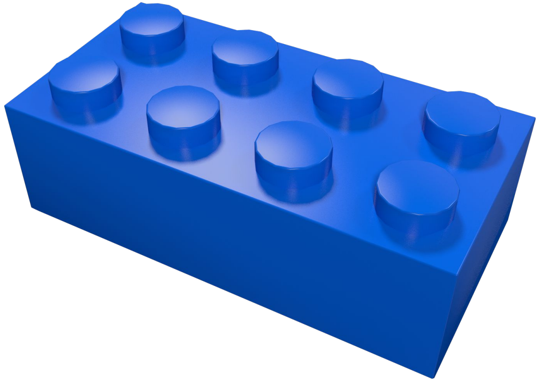


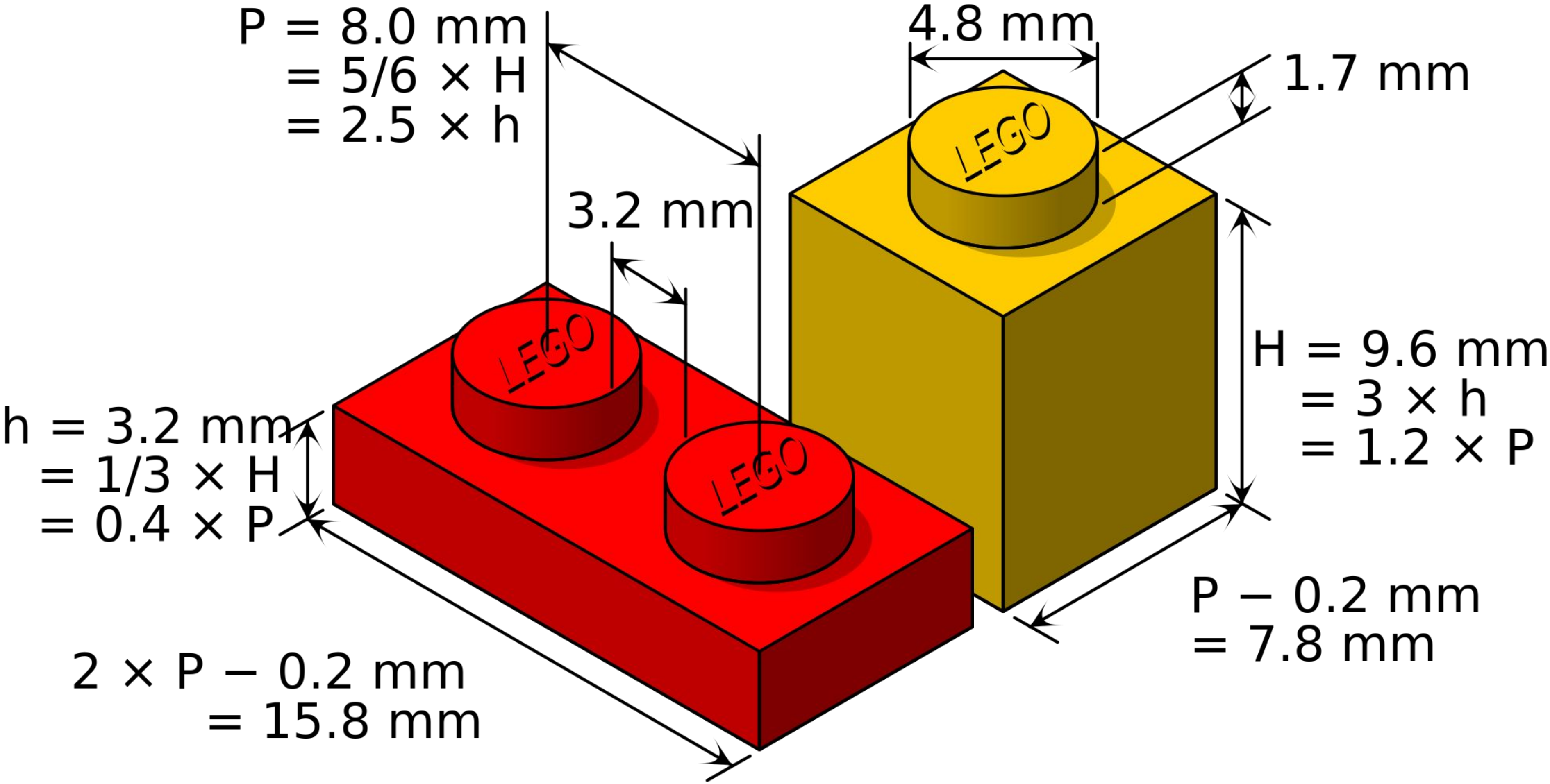




LINCOLN LOGS






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

interface

A common interface allows things to fit together.

Interfaces allow one thing to interact with another.

A common interface allows things to fit together.

interface

Interfaces allow one thing to interact with another.

An interface allows one thing to work with another.

A common interface allows things to fit together.

interface

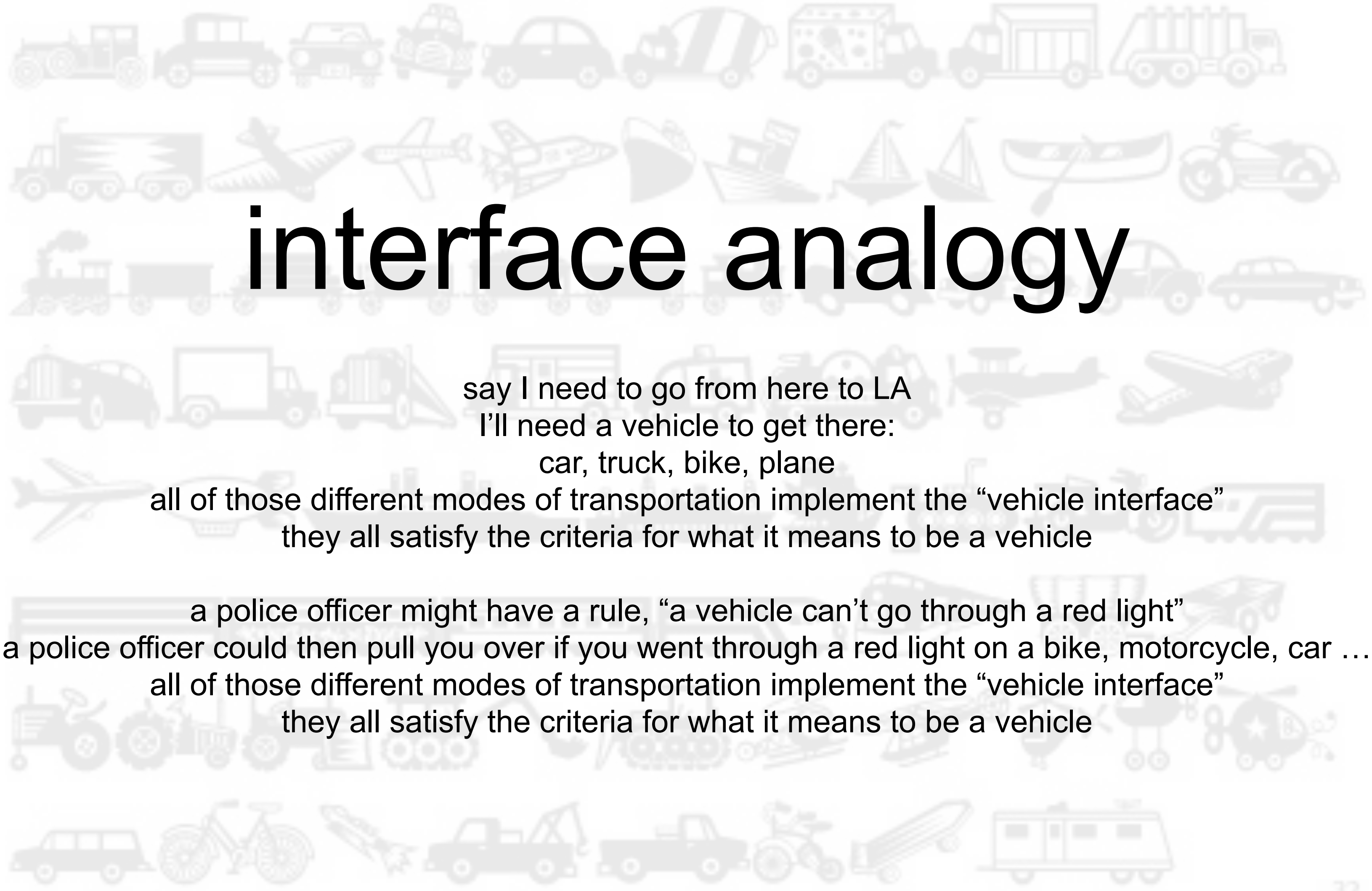
Interfaces allow one thing to interact with another.

An interface allows one thing to work with another.

A common interface allows things to fit together.

interface

These are all perfect descriptions of how we use interfaces in programming



interface analogy

say I need to go from here to LA
I’ll need a vehicle to get there:
car, truck, bike, plane
all of those different modes of transportation implement the “vehicle interface”
they all satisfy the criteria for what it means to be a vehicle

a police officer might have a rule, “a vehicle can’t go through a red light”
a police officer could then pull you over if you went through a red light on a bike, motorcycle, car …
all of those different modes of transportation implement the “vehicle interface”
they all satisfy the criteria for what it means to be a vehicle

interface

You could say
interfaces allow us to group things
by functionality; what they do; their methods

say I need to go from here to LA
I’ll need a vehicle to get there:
car, truck, bike, plane
all of those different modes of transportation implement the “vehicle interface”
they all satisfy the criteria for what it means to be a vehicle

a police officer might have a rule, “a vehicle can’t go through a red light”
a police officer could then pull you over if you went through a red light on a bike, motorcycle, car …
all of those different modes of transportation implement the “vehicle interface”
they all satisfy the criteria for what it means to be a vehicle

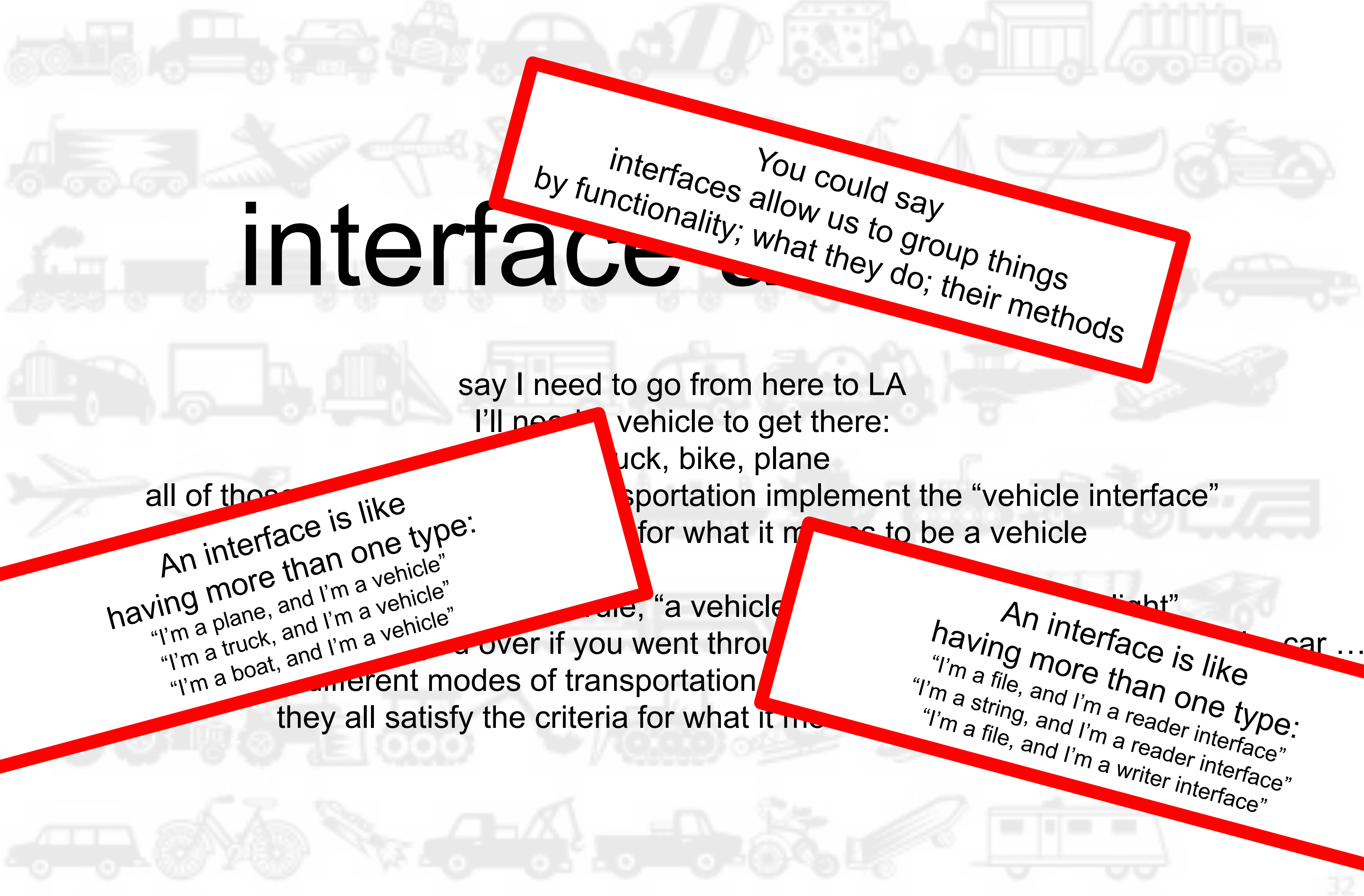
interface

You could say
interfaces allow us to group things
by functionality; what they do; their methods

say I need to go from here to LA
I'll nee vehicle to get there:
uck, bike, plane
all of thos sportation implement the "vehicle interface"
for what it means to be a vehicle

ule, "a vehicle can't go through a red light"
d over if you went through a red light on a bike, motorcycle, car …
ifferent modes of transportation implement the "vehicle interface"
they all satisfy the criteria for what it means to be a vehicle

An interface is like
having more than one type:
"I'm a plane, and I'm a vehicle"
"I'm a truck, and I'm a vehicle"
"I'm a boat, and I'm a vehicle"



interface

say I need to go from here to LA

I’ll nee vehicle to get there:

uck, bike, plane

all of thos sportation implement the “vehicle interface”

for what it m to be a vehicle

ight”

ule, “a vehicle car …

d over if you went throu

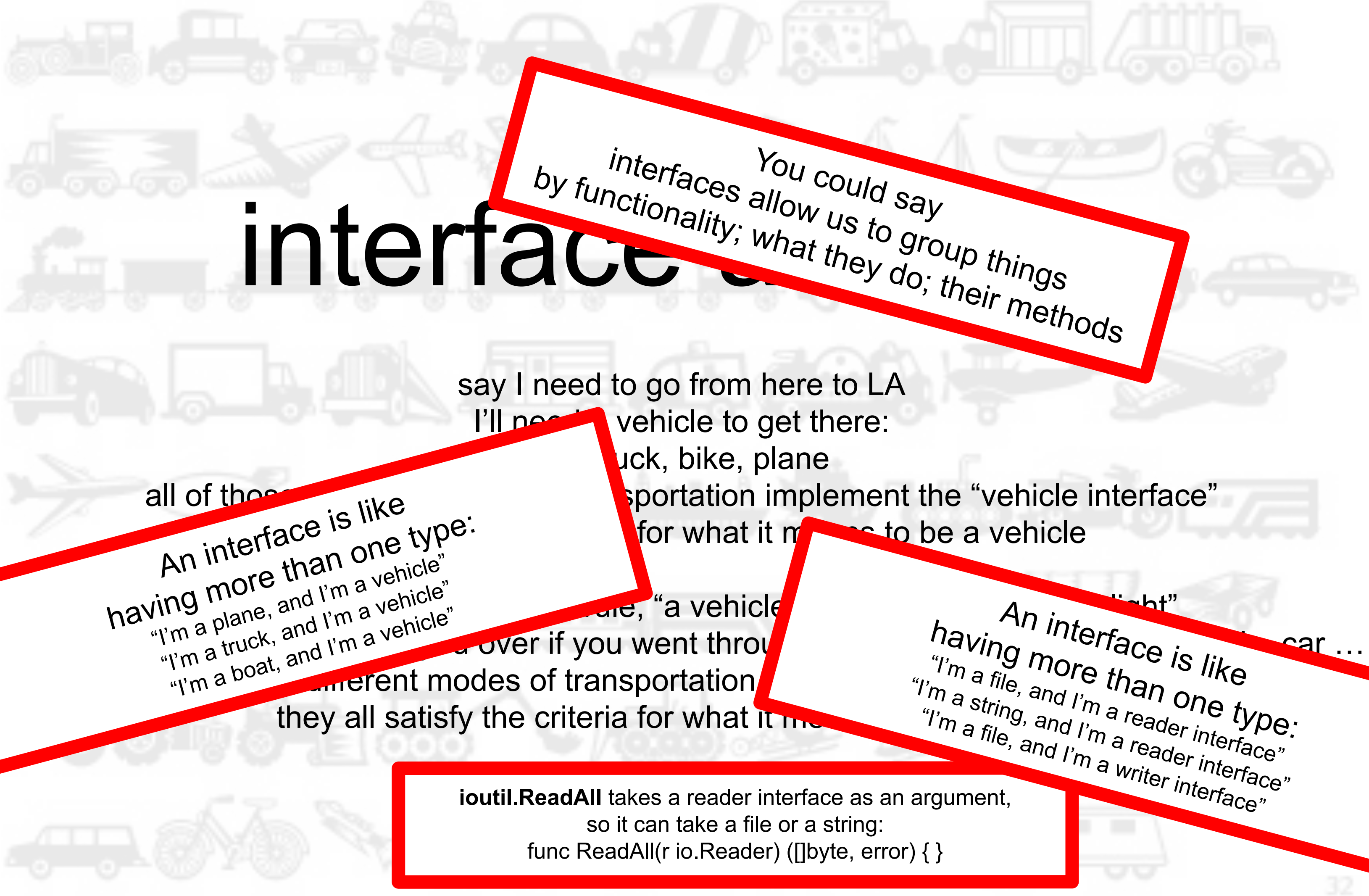
ifferent modes of transportation

they all satisfy the criteria for what it me

You could say
interfaces allow us to group things
by functionality; what they do; their methods

An interface is like
having more than one type:
“I’m a plane, and I’m a vehicle”
“I’m a truck, and I’m a vehicle”
“I’m a boat, and I’m a vehicle”

An interface is like
having more than one type:
“I’m a file, and I’m a reader interface”
“I’m a string, and I’m a reader interface”
“I’m a file, and I’m a writer interface”



interface a

say I need to go from here to LA
I'll nee vehicle to get there:
uck, bike, plane
all of thos sportation implement the "vehicle interface"
for what it m s to be a vehicle

light"
car …
ule, "a vehicle
d over if you went throu
ifferent modes of transportation
they all satisfy the criteria for what it me

You could say
interfaces allow us to group things
by functionality; what they do; their methods

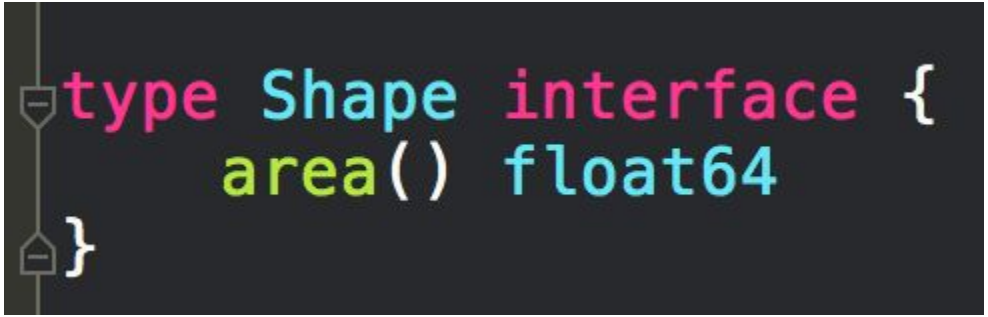
An interface is like
having more than one type:
"I'm a plane, and I'm a vehicle"
"I'm a truck, and I'm a vehicle"
"I'm a boat, and I'm a vehicle"

An interface is like
having more than one type:
"I'm a file, and I'm a reader interface"
"I'm a string, and I'm a reader interface"
"I'm a file, and I'm a writer interface"

ioutil.ReadAll takes a reader interface as an argument,
so it can take a file or a string:
func ReadAll(r io.Reader) ([]byte, error) { }

interface

a type
which defines a set of methods

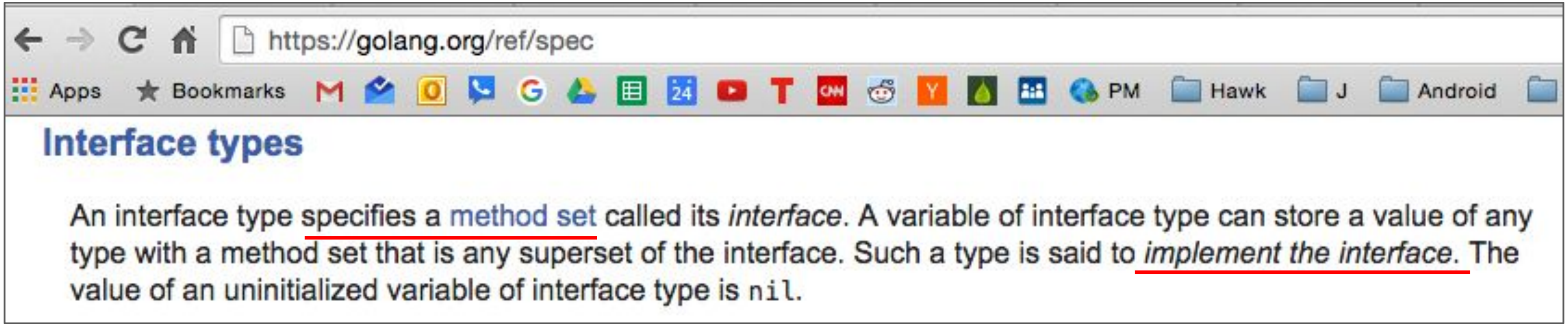


```
type Shape interface {
    area() float64
}
```

interface

a type
which defines a set of methods

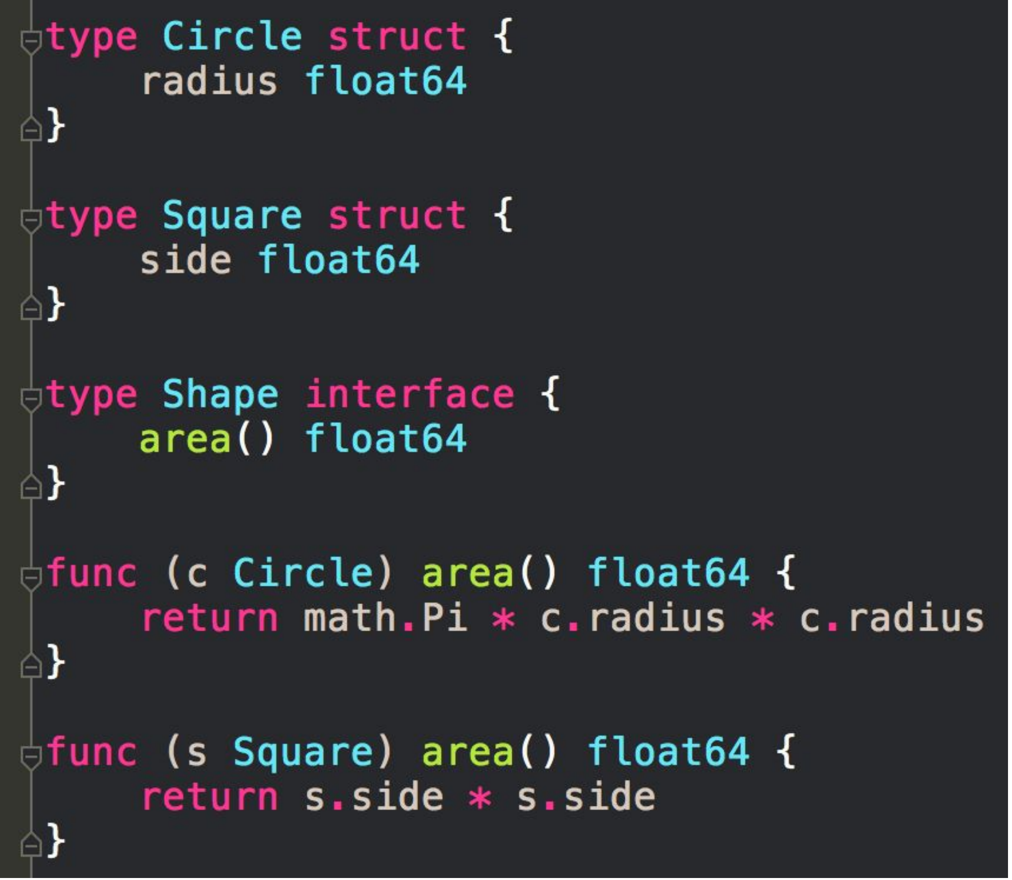
any type which has an area() float64 method
implements the Shape interface



Interface types

An interface type specifies a method set called its *interface*. A variable of interface type can store a value of any type with a method set that is any superset of the interface. Such a type is said to *implement the interface*. The value of an uninitialized variable of interface type is `nil`.

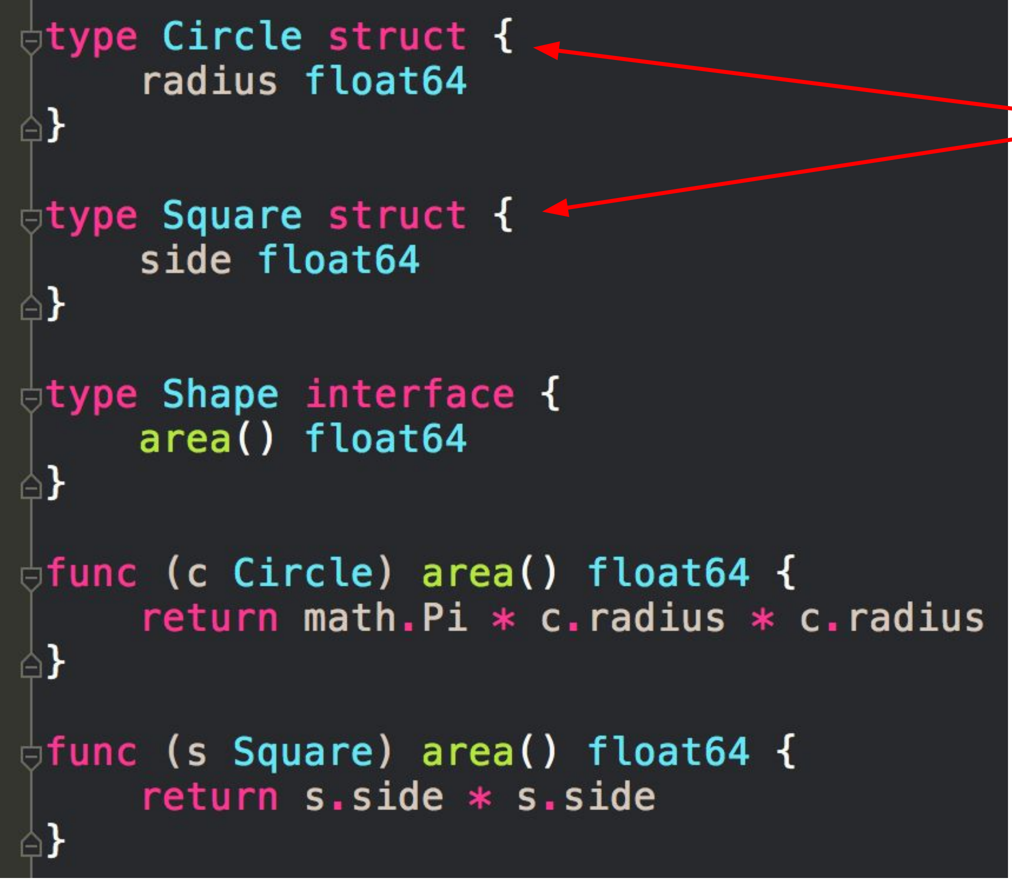
https://golang.org/ref/spec



```
type Circle struct {
	radius float64
}

type Square struct {
	side float64
}

type Shape interface {
	area() float64
}

func (c Circle) area() float64 {
	return math.Pi * c.radius * c.radius
}

func (s Square) area() float64 {
	return s.side * s.side
}
```

Are there any types here that implement the Shape interface?

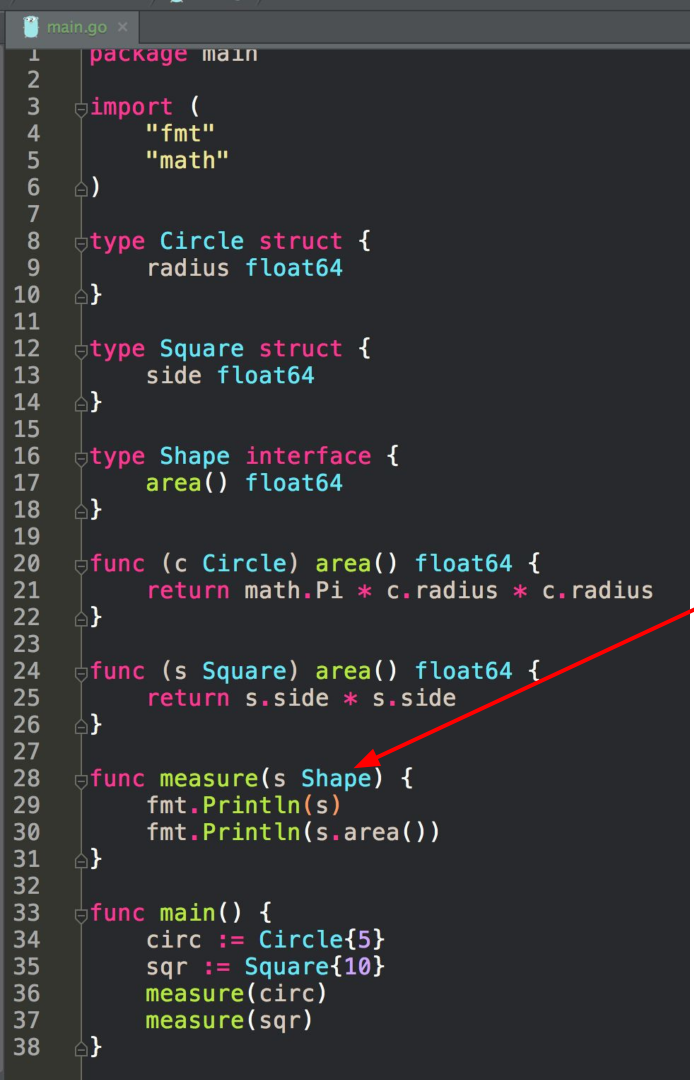
any type which has the methods defined by the Shape interface implements the Shape interface



```go
type Circle struct {
    radius float64
}

type Square struct {
    side float64
}

type Shape interface {
    area() float64
}

func (c Circle) area() float64 {
    return math.Pi * c.radius * c.radius
}

func (s Square) area() float64 {
    return s.side * s.side
}
```

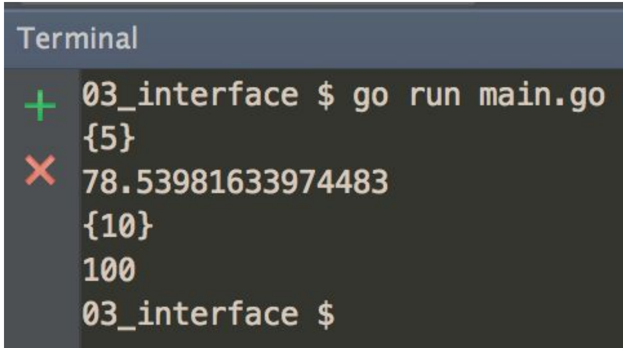
Are there any types here that implement the Shape interface?

any type which has the methods defined by the Shape interface implements the Shape interface

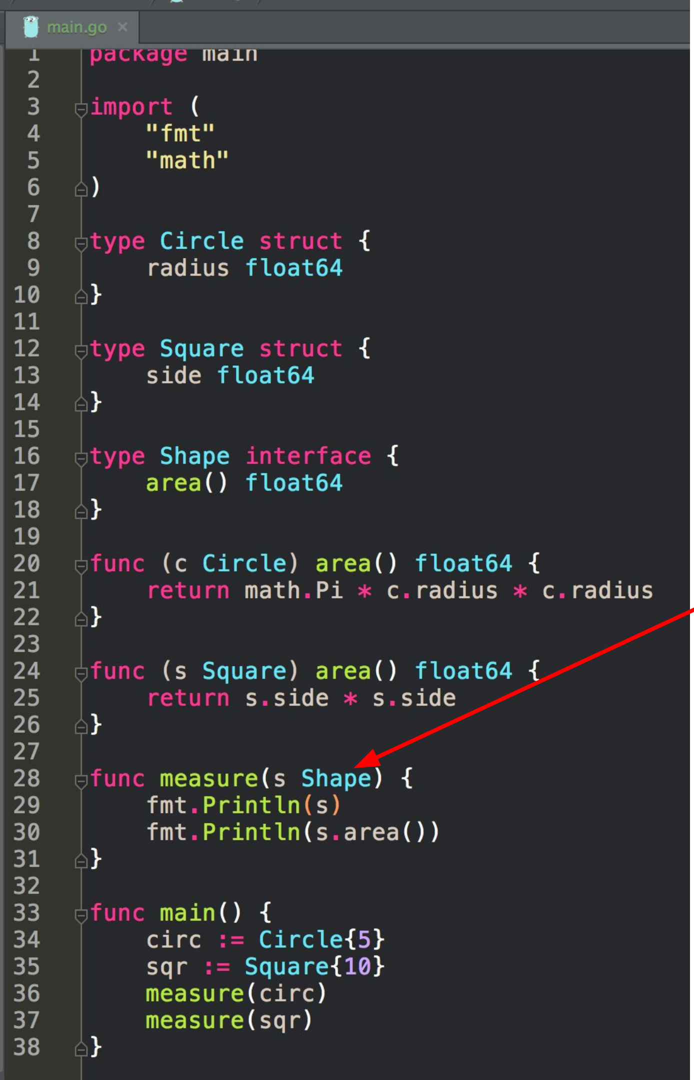


```go
package main

import (
	"fmt"
	"math"
)

type Circle struct {
	radius float64
}

type Square struct {
	side float64
}

type Shape interface {
	area() float64
}

func (c Circle) area() float64 {
	return math.Pi * c.radius * c.radius
}

func (s Square) area() float64 {
	return s.side * s.side
}

func measure(s Shape) {
	fmt.Println(s)
	fmt.Println(s.area())
}

func main() {
	circ := Circle{5}
	sqr := Square{10}
	measure(circ)
	measure(sqr)
}
```

Because func measure has type Shape as a parameter, anything that implements the Shape interface can be passed into this func



```
03_interface $ go run main.go
{5}
78.53981633974483
{10}
100
03_interface $
```

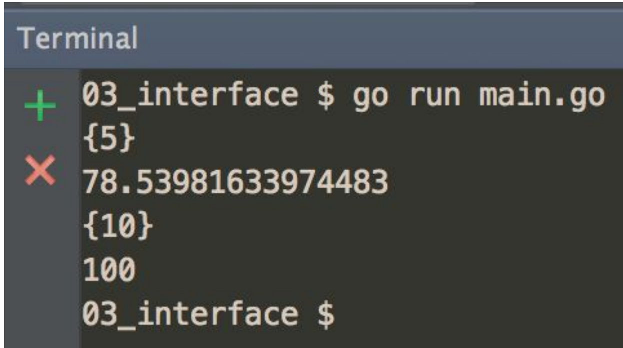


```go
package main

import (
	"fmt"
	"math"
)

type Circle struct {
	radius float64
}

type Square struct {
	side float64
}

type Shape interface {
	area() float64
}

func (c Circle) area() float64 {
	return math.Pi * c.radius * c.radius
}

func (s Square) area() float64 {
	return s.side * s.side
}

func measure(s Shape) {
	fmt.Println(s)
	fmt.Println(s.area())
}

func main() {
	circ := Circle{5}
	sqr := Square{10}
	measure(circ)
	measure(sqr)
}
```

Polymorphism

> *“In programming languages and type theory, polymorphism (from Greek πολύς, polys, "many, much" and μορφή, morphē, "form, shape") is the provision of a single interface to entities of different types. A polymorphic type is one whose operations can also be applied to values of some other type, or types.”*
>
> ~ Wikipedia

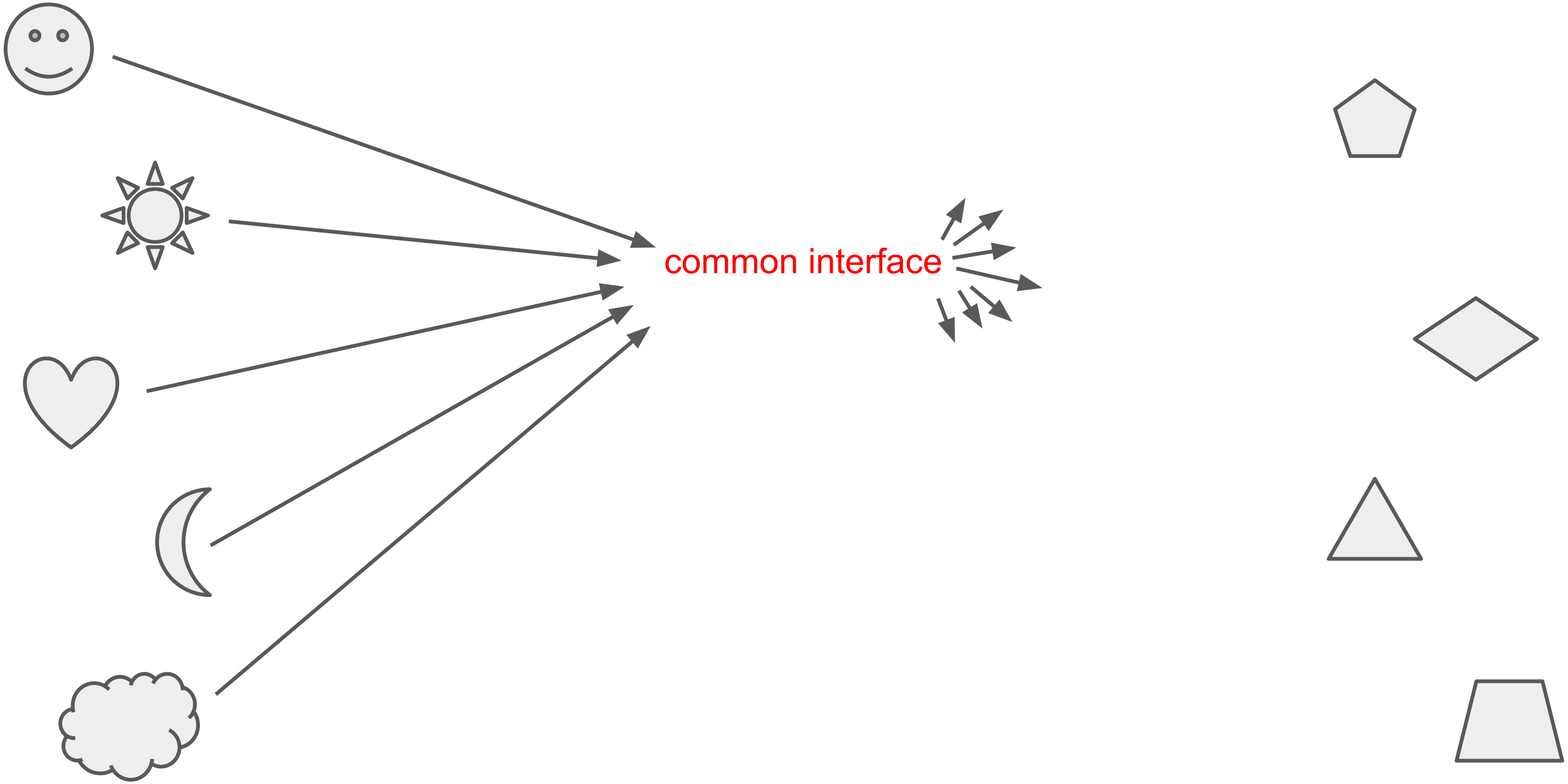
Because func measure has type Shape as a parameter, anything that implements the Shape interface can be passed into this func



```
03_interface $ go run main.go
{5}
78.53981633974483
{10}
100
03_interface $
```

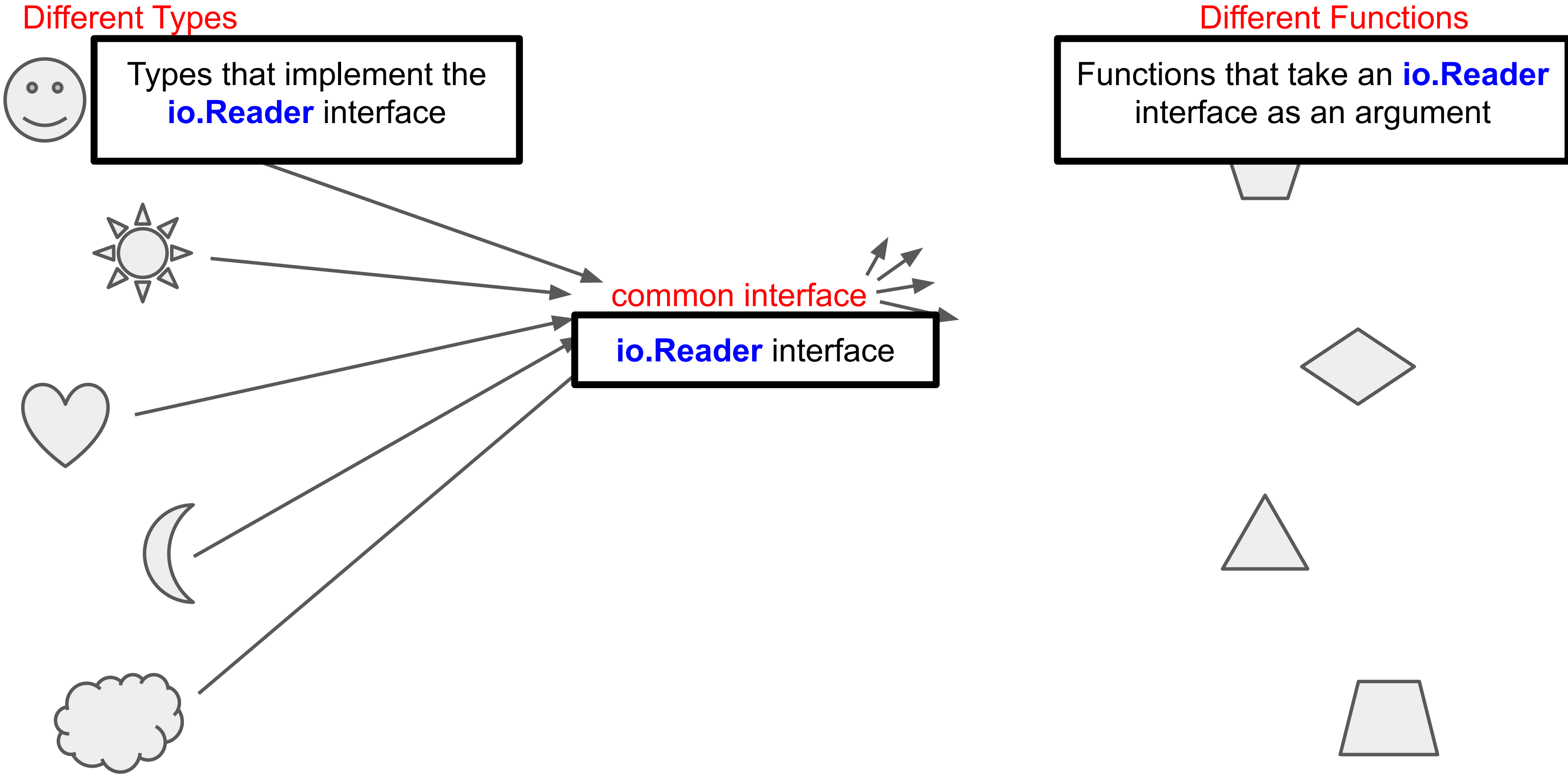
Different Types

Different Functions



common interface

any of the types on the left
can be matched with any of the functions on the right
because all of the functions share a common, agreed upon way of interfacing with each other



Different Types

Types that implement the **io.Reader** interface

common interface

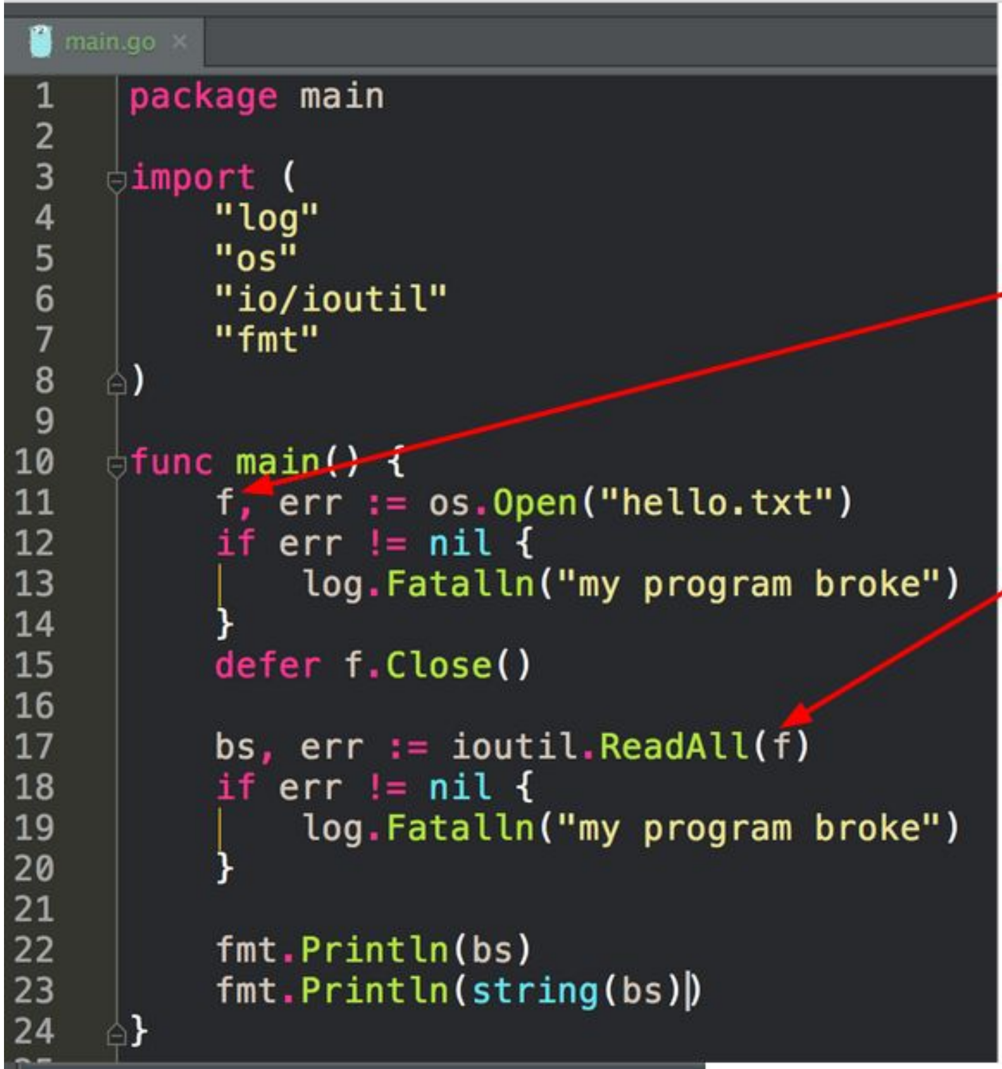
io.Reader interface

Different Functions

Functions that take an **io.Reader** interface as an argument

any of the types on the left
can be matched with any of the functions on the right
because all of the functions share a common, agreed upon way of interfacing with each other

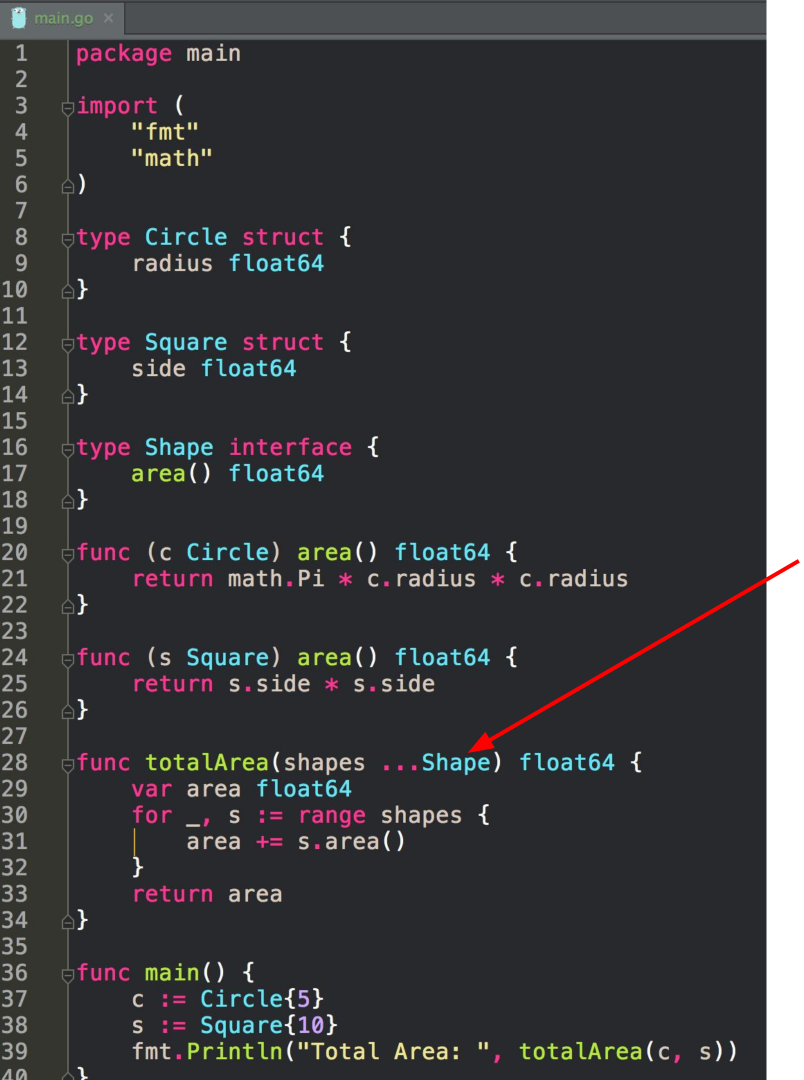
Reading A File



```go
package main

import (
    "log"
    "os"
    "io/ioutil"
    "fmt"
)

func main() {
    f, err := os.Open("hello.txt")
    if err != nil {
        log.Fatalln("my program broke")
    }
    defer f.Close()

    bs, err := ioutil.ReadAll(f)
    if err != nil {
        log.Fatalln("my program broke")
    }

    fmt.Println(bs)
    fmt.Println(string(bs))
}
```

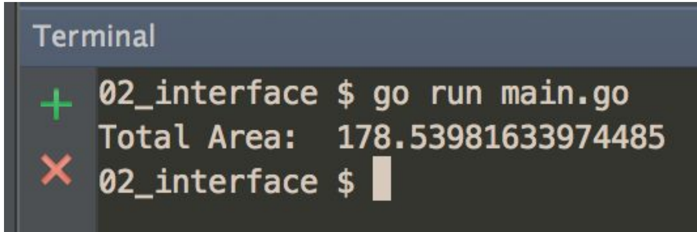
f
implements the
reader interface

ReadAll
takes a reader interface



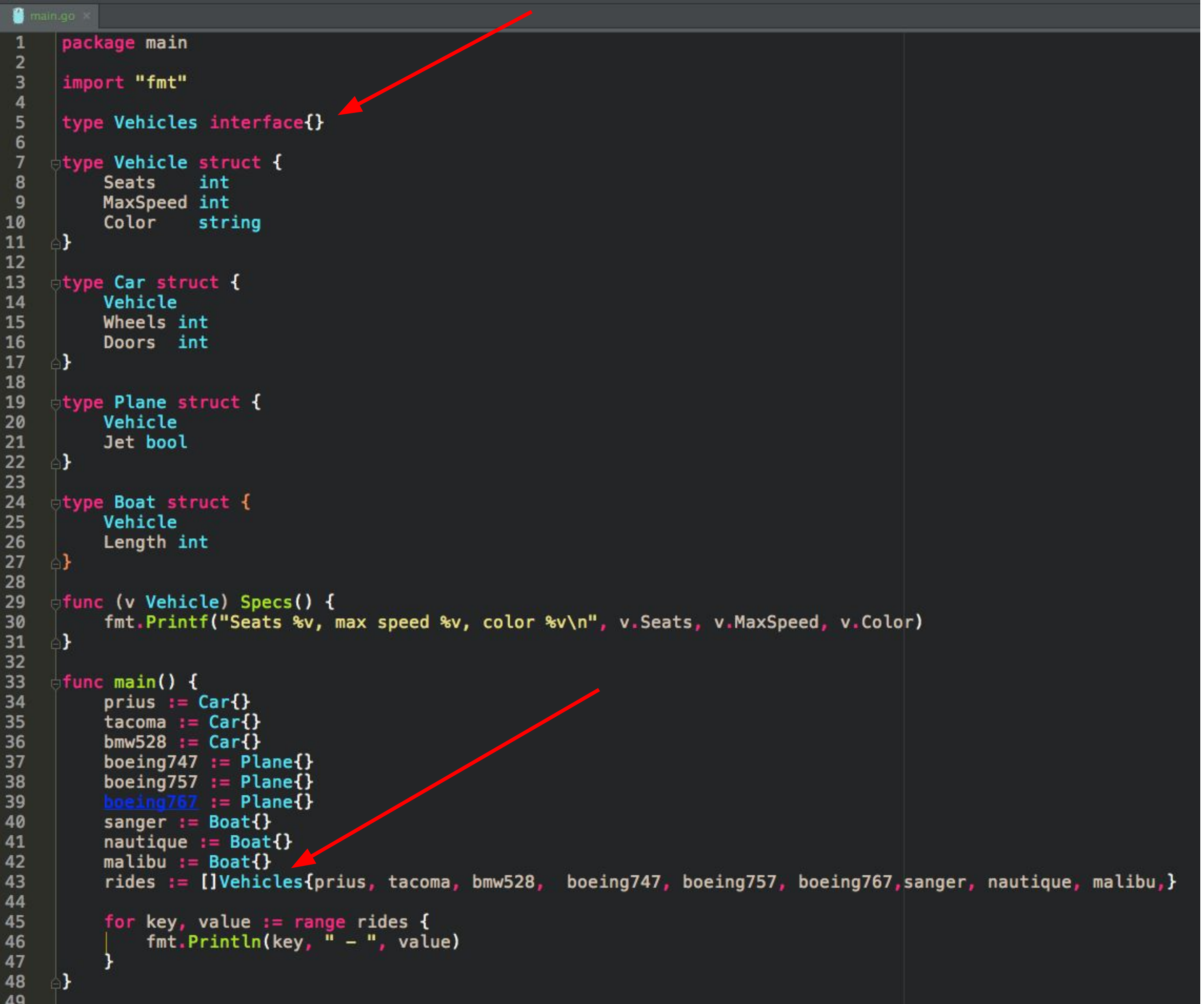
```go
package main

import (
	"fmt"
	"math"
)

type Circle struct {
	radius float64
}

type Square struct {
	side float64
}

type Shape interface {
	area() float64
}

func (c Circle) area() float64 {
	return math.Pi * c.radius * c.radius
}

func (s Square) area() float64 {
	return s.side * s.side
}

func totalArea(shapes ...Shape) float64 {
	var area float64
	for _, s := range shapes {
		area += s.area()
	}
	return area
}

func main() {
	c := Circle{5}
	s := Square{10}
	fmt.Println("Total Area: ", totalArea(c, s))
}
```

Because func totalArea has type Shape as a variadic parameter, anything that implements the Shape interface can be passed into this func

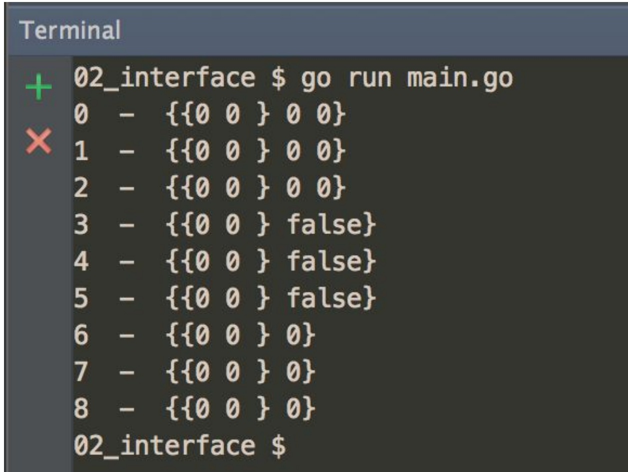


```
02_interface $ go run main.go
Total Area:  178.53981633974485
02_interface $
```

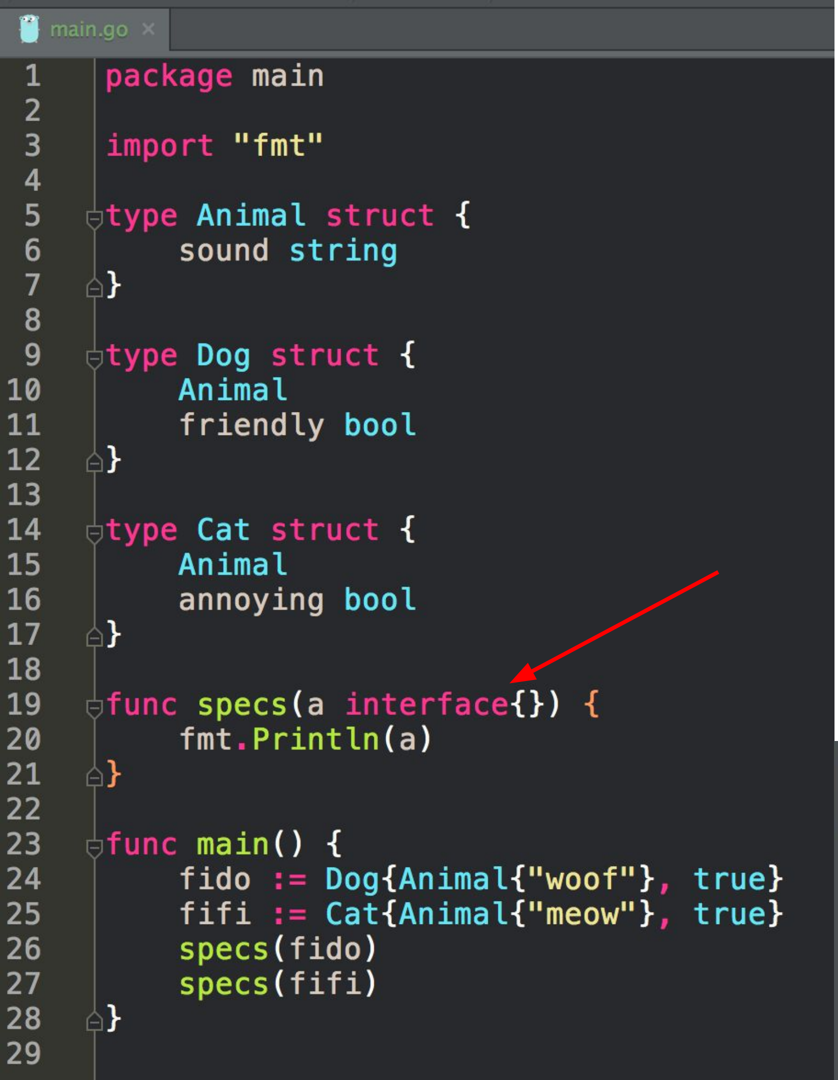
Everything implements the **empty interface**



```go
package main

import "fmt"

type Vehicles interface{}

type Vehicle struct {
	Seats    int
	MaxSpeed int
	Color    string
}

type Car struct {
	Vehicle
	Wheels int
	Doors  int
}

type Plane struct {
	Vehicle
	Jet bool
}

type Boat struct {
	Vehicle
	Length int
}

func (v Vehicle) Specs() {
	fmt.Printf("Seats %v, max speed %v, color %v\n", v.Seats, v.MaxSpeed, v.Color)
}

func main() {
	prius := Car{}
	tacoma := Car{}
	bmw528 := Car{}
	boeing747 := Plane{}
	boeing757 := Plane{}
	boeing767 := Plane{}
	sanger := Boat{}
	nautique := Boat{}
	malibu := Boat{}
	rides := []Vehicles{prius, tacoma, bmw528,  boeing747, boeing757, boeing767,sanger, nautique, malibu,}

	for key, value := range rides {
		fmt.Println(key, " - ", value)
	}
}
```

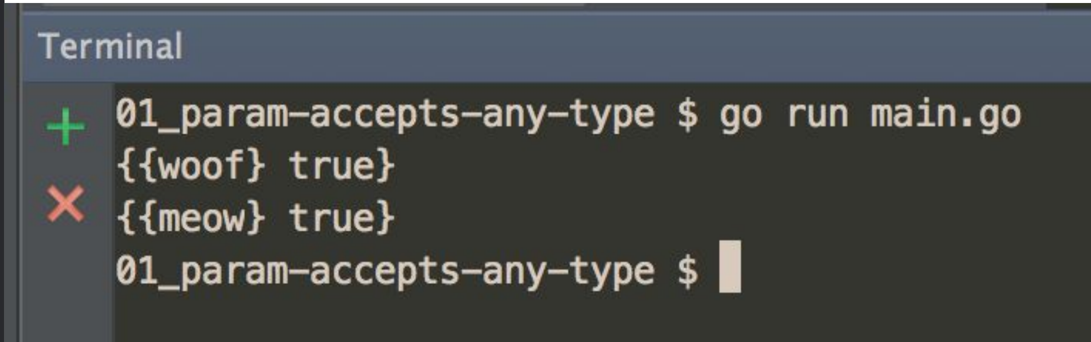


```
02_interface $ go run main.go
0  -  {{0 0 } 0 0}
1  -  {{0 0 } 0 0}
2  -  {{0 0 } 0 0}
3  -  {{0 0 } false}
4  -  {{0 0 } false}
5  -  {{0 0 } false}
6  -  {{0 0 } 0}
7  -  {{0 0 } 0}
8  -  {{0 0 } 0}
02_interface $
```



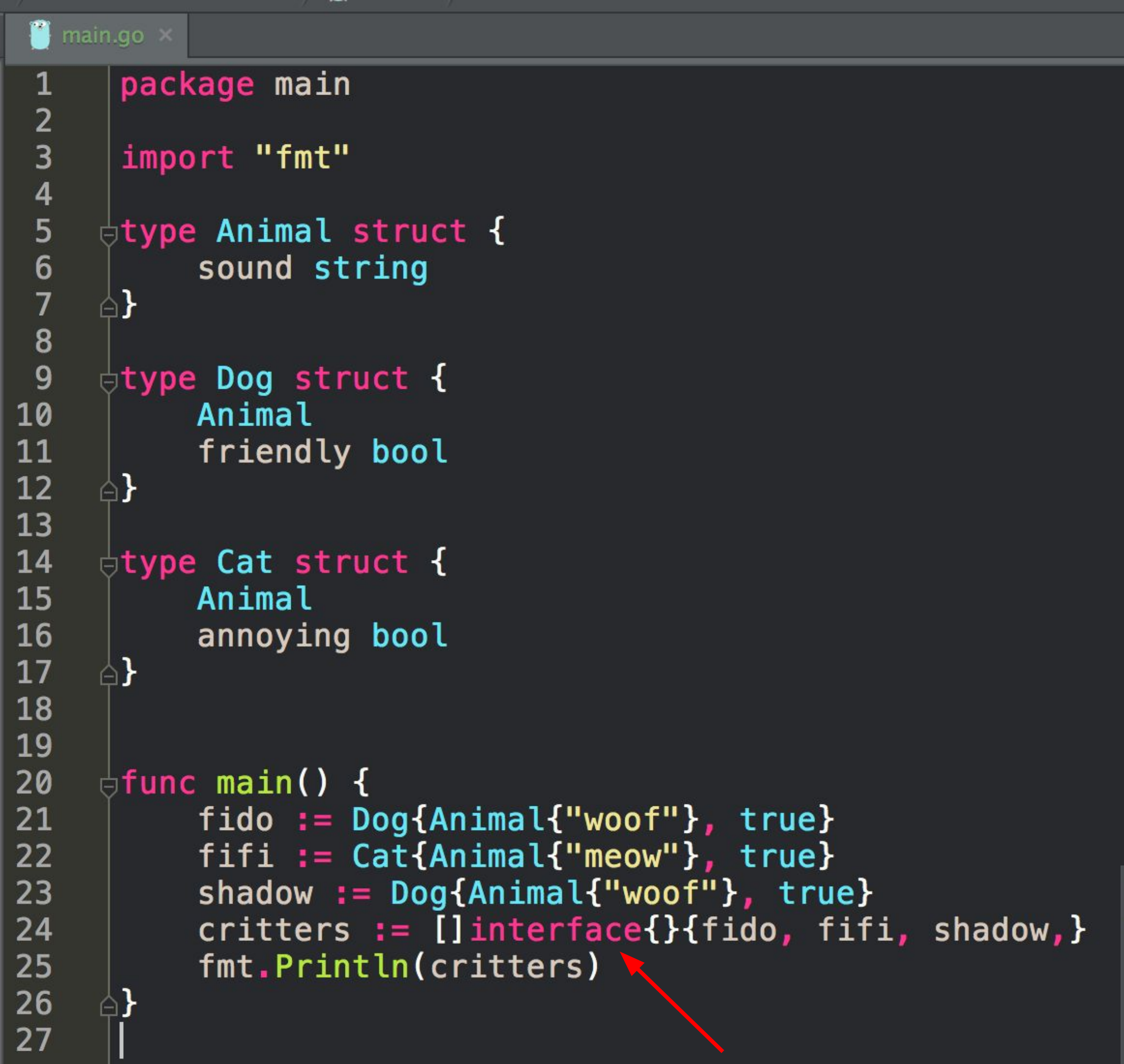
```go
package main

import "fmt"

type Animal struct {
	sound string
}

type Dog struct {
	Animal
	friendly bool
}

type Cat struct {
	Animal
	annoying bool
}

func specs(a interface{}) {
	fmt.Println(a)
}

func main() {
	fido := Dog{Animal{"woof"}, true}
	fifi := Cat{Animal{"meow"}, true}
	specs(fido)
	specs(fifi)
}
```

Everything implements
the **empty interface**

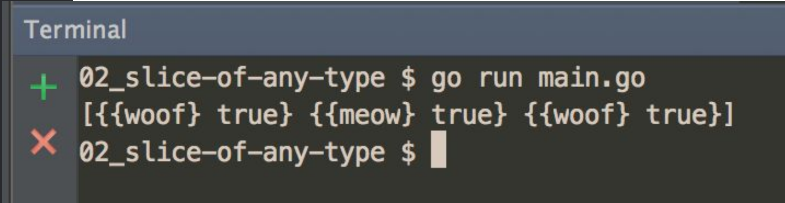


```
01_param-accepts-any-type $ go run main.go
{{woof} true}
{{meow} true}
01_param-accepts-any-type $ 
```

Everything implements the **empty interface**



```go
package main

import "fmt"

type Animal struct {
    sound string
}

type Dog struct {
    Animal
    friendly bool
}

type Cat struct {
    Animal
    annoying bool
}


func main() {
    fido := Dog{Animal{"woof"}, true}
    fifi := Cat{Animal{"meow"}, true}
    shadow := Dog{Animal{"woof"}, true}
    critters := []interface{}{fido, fifi, shadow,}
    fmt.Println(critters)
}

```



```
02_slice-of-any-type $ go run main.go
[{{woof} true} {{meow} true} {{woof} true}]
02_slice-of-any-type $
```

exercise

write a program
that uses an anonymous interface
to store any type in a slice

exercise

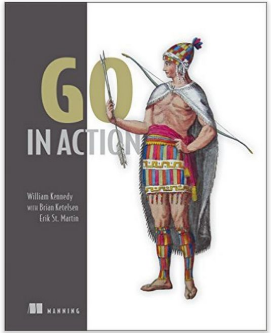
write a program
that has a function that uses an anonymous interface as a param
demonstrate that function being used
with different types

interfaces

in more depth

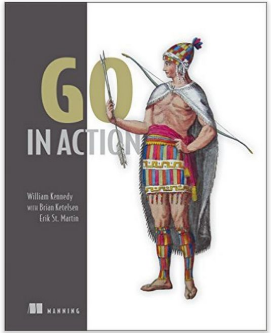
Interfaces are types that just declare behavior. This behavior is never implemented by the interface type directly, but instead by user-defined types via methods. When a user-defined type implements the set of methods declared by an interface type, values of the user-defined type can be assigned to values of the interface type. This assignment stores the value of the user-defined type into the interface value.

If a method call is made against an interface value, the equivalent method for the stored user-defined value is executed. Since any user-defined type can implement any interface, method calls against an interface value are polymorphic in nature. The user-defined type in this relationship is often called a *concrete type*, since interface values have no concrete behavior without the implementation of the stored user-defined value.



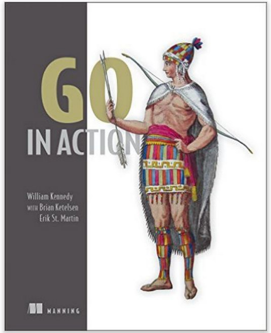
Implementation Details

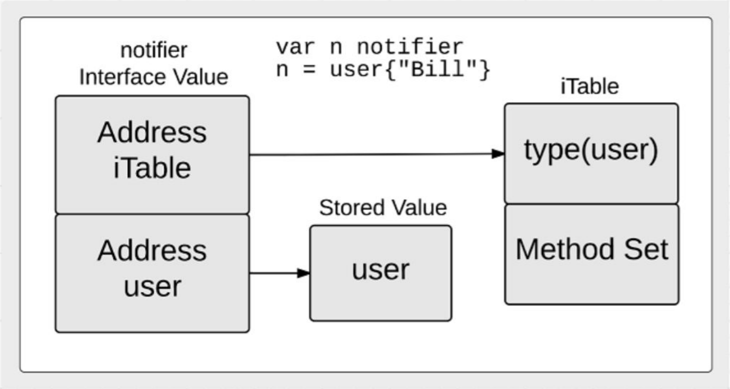
- Interface:
    - two-word data structure
        - first word
            - pointer to an internal table called an iTable
            - contains information about the stored value
                - the type
                - associated methods
        - second word
            - pointer to the stored value



Implementation Details

- Interface:
    - two-word data structure
        - first word
            - pointer to an internal table called an iTable
            - contains information about the stored value
                - the type
                - associated methods
        - second word
            - pointer to the stored value



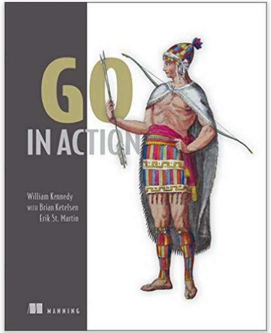


```
var n notifier
n = user{"Bill"}
```

Figure 5.1 A simple view of an interface value after concrete type value assignment

Implementation Details

- Interface:
    - two-word data structure
        - first word
            - pointer to an internal table called an iTable
            - contains information about the stored value
                - the type
                - associated methods
        - second word
            - pointer to the stored value



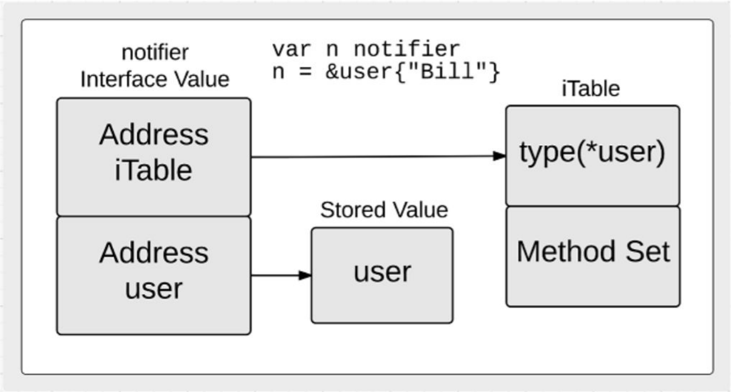


Figure 5.2 A simple view of an interface value after concrete type pointer assignment

Review

- interface type
    - an interface defines a set of methods

Review Questions

interfaces

- How would a bike, truck, and car all implement the vehicle interface?

interfaces

- In your own words, describe why interfaces are useful.