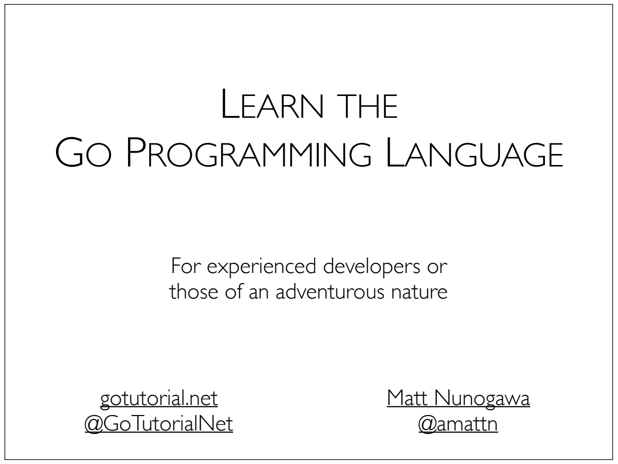


Learn the Go Programming Language

For experienced developers or
those of an adventurous nature

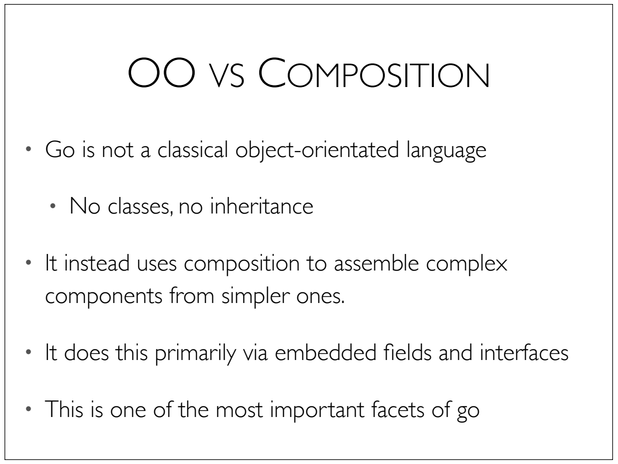
gotutorial.net
@GoTutorialNet

Matt Nunogawa
@amattn

Level 04

Composition
Embedded Fields & Interfaces

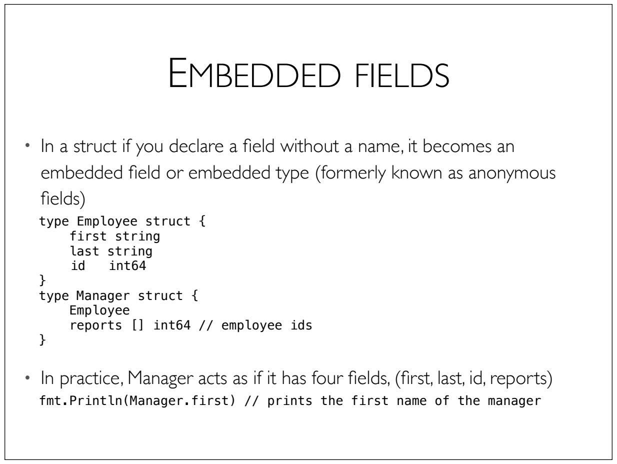
`v0.5 draft`



OO vs Composition

- Go is not a classical object-orientated language
  - No classes, no inheritance
- It instead uses composition to assemble complex components from simpler ones.
- It does this primarily via embedded fields and interfaces
- This is one of the most important facets of go

The primary authors of Go do, in fact, call it an OO language because you can do OO-ish things with it.



EMBEDDED FIELDS

- In a struct if you declare a field without a name, it becomes an embedded field or embedded type (formerly known as anonymous fields)

```
type Employee struct {
    first string
    last string
    id   int64
}
type Manager struct {
    Employee
    reports [] int64 // employee ids
}
```

- In practice, Manager acts as if it has four fields, (first, last, id, reports)

```
fmt.Println(Manager.first) // prints the first name of the manager
```

Literals: Embedded fields

- Struct Literals with Embedded Fields:

```
m := Manager{
   Employee{“Bob”, “Jones”, 3},
   []int64{5,8,12},
}

fmt.Println(m.first) // prints Bob
```

Embedded Doesn't Mean Inaccessible

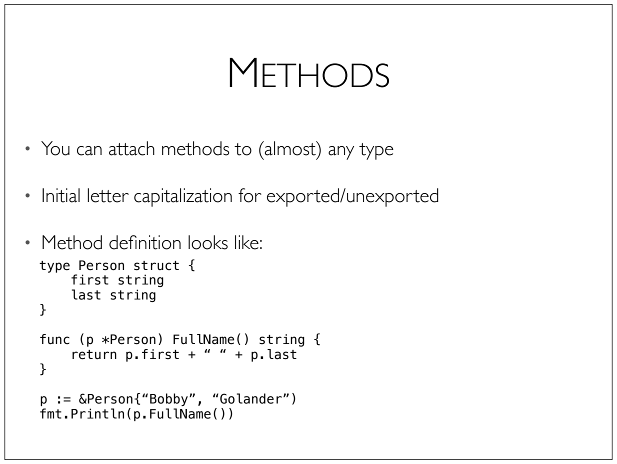
- Each Embedded field is accessible via the name of the type

```
fmt.Println(m.first)          // prints Bob
fmt.Println(m.Employee.first) // prints Bob
fmt.Println(m.id)            // prints 3
fmt.Println(m.Employee.id)   // prints 3
```

Embedded Field Conflicts

- When you have naming conflicts:
  - If the same name appears twice at the same level, it is an error if used. (If it's not used, it doesn't matter.)
  - An outer name hides and inner name

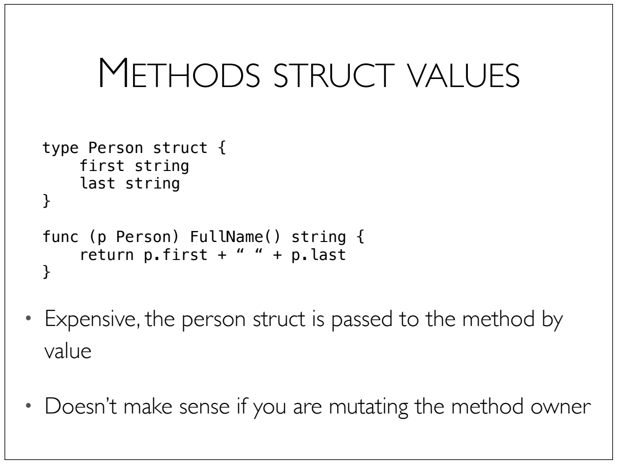
```
type A struct { a int }
type B struct { a, b int }
type C struct { A; B }
var c C

c.a     // error!
c.A.a   // ok
c.B.a   // ok

type D { A; a int}
var d D
d.a     // not an error!
d.A.a   // also ok
```



METHODS

- You can attach methods to (almost) any type
- Initial letter capitalization for exported/unexported
- Method definition looks like:

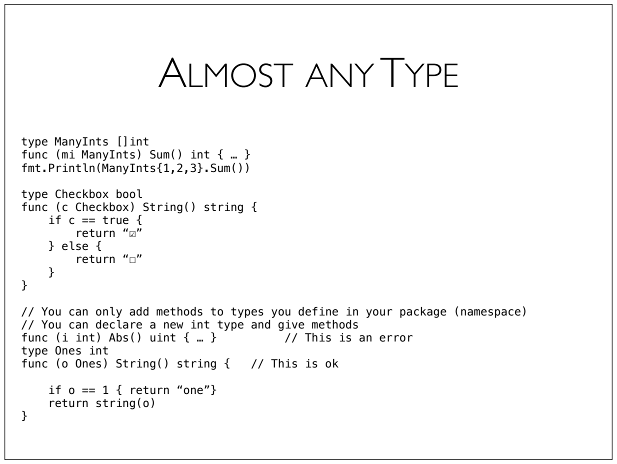
```
type Person struct {
    first string
    last string
}

func (p *Person) FullName() string {
    return p.first + “ “ + p.last
}

p := &Person{“Bobby”, “Golander”)
fmt.Println(p.FullName())
```



METHODS STRUCT VALUES

```
type Person struct {
    first string
    last string
}

func (p Person) FullName() string {
    return p.first + “ “ + p.last
}
```

- Expensive, the person struct is passed to the method by value
- Doesn’t make sense if you are mutating the method owner



Almost any Type

```
type ManyInts []int
func (mi ManyInts) Sum() int { … }
fmt.Println(ManyInts{1,2,3}.Sum())

type Checkbox bool
func (c Checkbox) String() string {
    if c == true {
        return “☑”
    } else {
        return “□”
    }
}

// You can only add methods to types you define in your package (namespace)
// You can declare a new int type and give methods
func (i int) Abs() uint { … }          // This is an error
type Ones int
func (o Ones) String() string {   // This is ok

    if o == 1 { return “one”}
    return string(o)
}
```

AUTOMATIC DEREFERENCING

```
  type Person struct {
      first string
      last string
  }

  func (p *Person) FullName() string {
      return p.first + " " + p.last
  }

  p := Person{"Bobby", "Golander")
  fmt.Println(p.FullName())
// compiler is nice enough to convert to
// (&p).FullName() for us
```

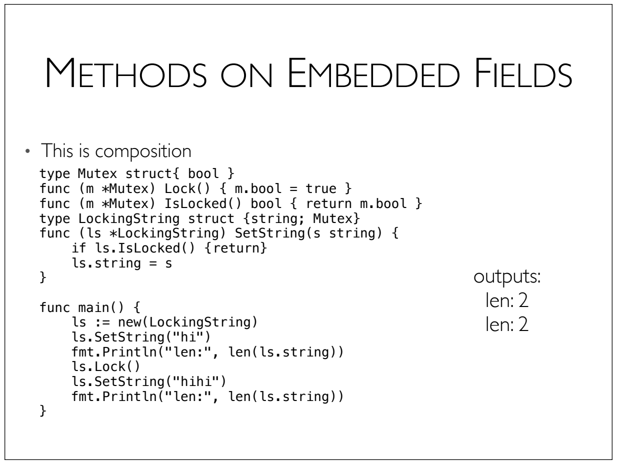
Methods on Embedded Fields

```
type Person struct {
    first string
    last string
}
type Employee struct {
    Person
    id   int64
}

func (p *Person) FullName() string {
    return p.first + “ “ + p.last
}

e := Employee{Person{“Bobby”,”Golander”}, 9}
e.FullName()         // this works!
e.Person.FullName()  // this also works, and is more typing
```

Methods on Embedded Fields

- This mechanism is one way to emulate inheritance
- Can also do overriding:

```
func (e *Employee) FullName() string {
    return e.Person.FullName() + “ “ + string(e.id)
}

// These two lines no longer print the same thing
e.FullName()
e.Person.FullName()
```



Methods on Embedded Fields

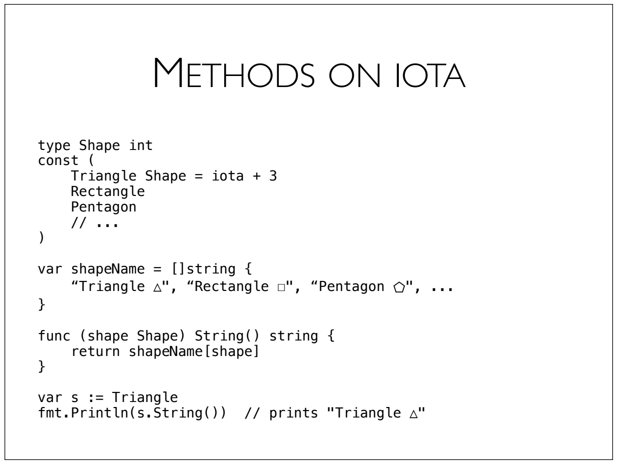
- This is composition

```
type Mutex struct{ bool }
func (m *Mutex) Lock() { m.bool = true }
func (m *Mutex) IsLocked() bool { return m.bool }
type LockingString struct {string; Mutex}
func (ls *LockingString) SetString(s string) {
    if ls.IsLocked() {return}
    ls.string = s
}

func main() {
    ls := new(LockingString)
    ls.SetString("hi")
    fmt.Println("len:", len(ls.string))
    ls.Lock()
    ls.SetString("hihi")
    fmt.Println("len:", len(ls.string))
}
```

outputs:

```
len: 2
len: 2
```

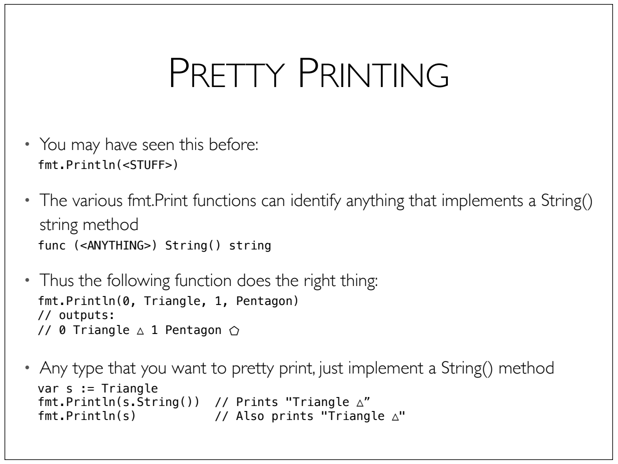
http://play.golang.org/p/lqT0TSxhiR



Methods on iota

```
type Shape int
const (
    Triangle Shape = iota + 3
    Rectangle
    Pentagon
    // ...
)

var shapeName = []string {
    “Triangle △", “Rectangle □", “Pentagon ⬠", ...
}

func (shape Shape) String() string {
    return shapeName[shape]
}

var s := Triangle
fmt.Println(s.String())  // prints "Triangle △"
```

http://play.golang.org/p/A8l1gRxgCT



Pretty Printing

- You may have seen this before:

```
fmt.Println(<STUFF>)
```

- The various fmt.Print functions can identify anything that implements a String() string method

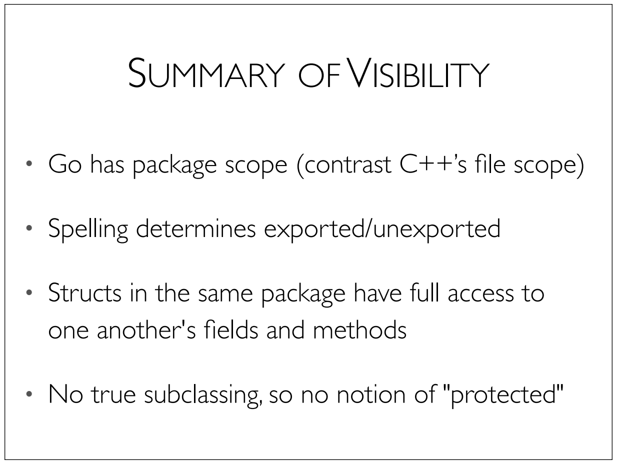
```
func (<ANYTHING>) String() string
```

- Thus the following function does the right thing:

```
fmt.Println(0, Triangle, 1, Pentagon)
// outputs:
// 0 Triangle △ 1 Pentagon ⬠
```

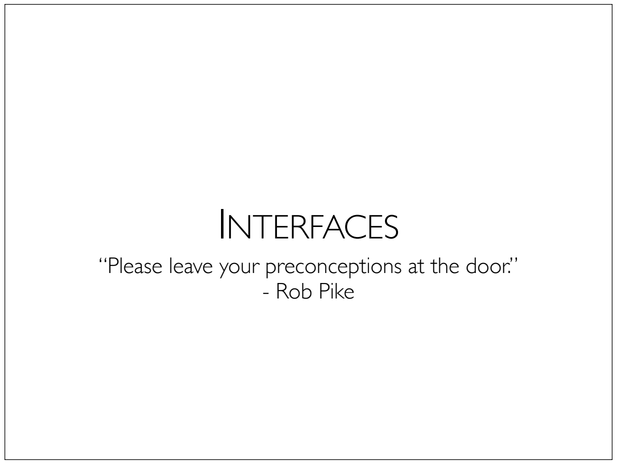
- Any type that you want to pretty print, just implement a String() method

```
var s := Triangle
fmt.Println(s.String())  // Prints "Triangle △”
fmt.Println(s)           // Also prints "Triangle △"
```



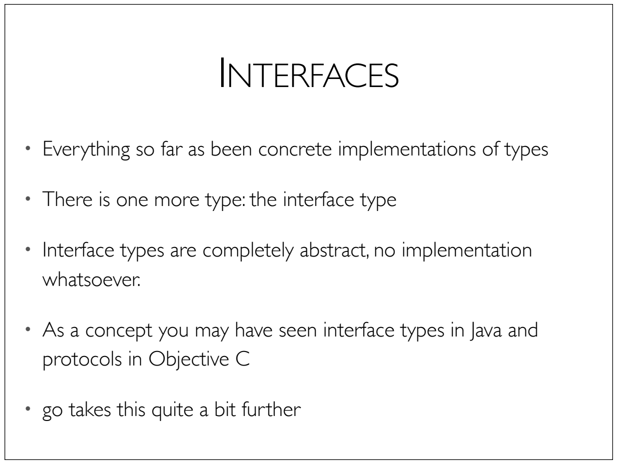
SUMMARY OF VISIBILITY

- Go has package scope (contrast C++'s file scope)
- Spelling determines exported/unexported
- Structs in the same package have full access to one another's fields and methods
- No true subclassing, so no notion of "protected"



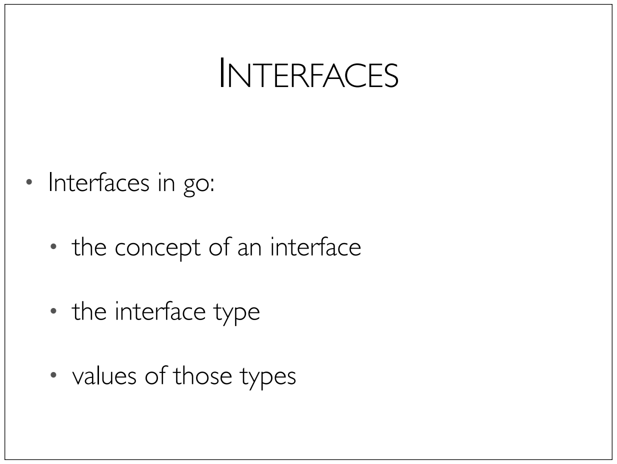
INTERFACES

"Please leave your preconceptions at the door."
- Rob Pike



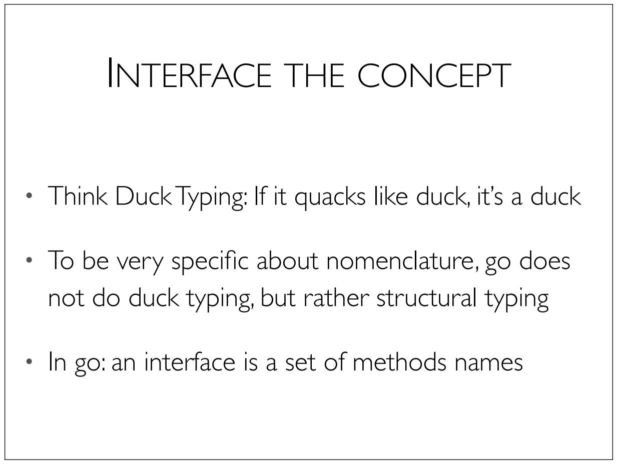
INTERFACES

- Everything so far as been concrete implementations of types
- There is one more type: the interface type
- Interface types are completely abstract, no implementation whatsoever.
- As a concept you may have seen interface types in Java and protocols in Objective C
- go takes this quite a bit further



INTERFACES

- Interfaces in go:
  - the concept of an interface
  - the interface type
  - values of those types



Interface the concept

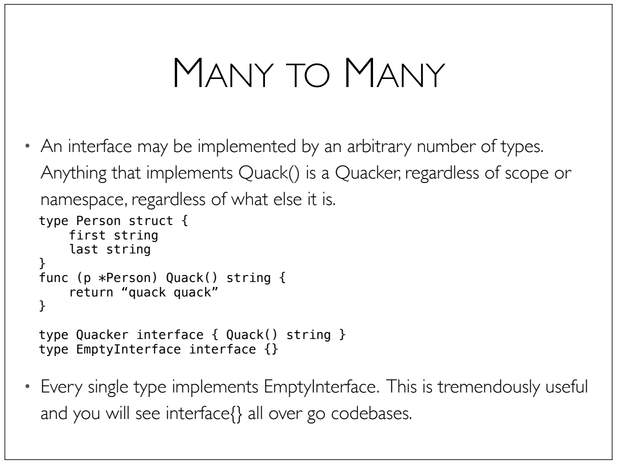
- Think Duck Typing: If it quacks like duck, it's a duck
- To be very specific about nomenclature, go does not do duck typing, but rather structural typing
- In go: an interface is a set of methods names

Go doesn't strictly do duck typing, but the nomenclature is too useful to not use.

THE INTERFACE TYPE

- An interface type is the specification of an interface:

```
type Quacker interface {
    Quack() string
}
type Swimmer interface {
    Swim()
    HoldBreath(int) error
}
```

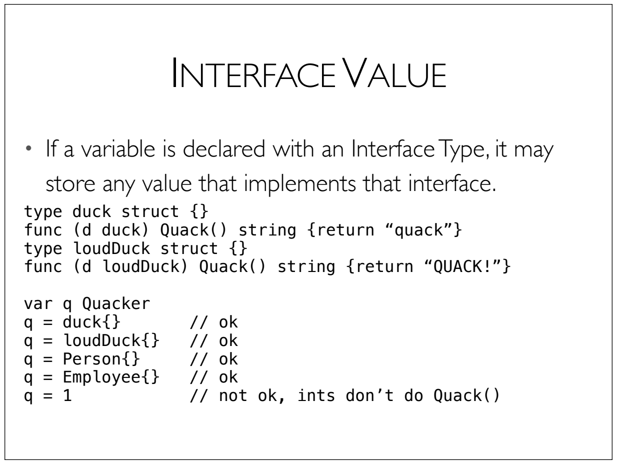


Many to Many

- An interface may be implemented by an arbitrary number of types. Anything that implements Quack() is a Quacker, regardless of scope or namespace, regardless of what else it is.

```
type Person struct {
    first string
    last string
}
func (p *Person) Quack() string {
    return "quack quack"
}

type Quacker interface { Quack() string }
type EmptyInterface interface {}
```

- Every single type implements EmptyInterface. This is tremendously useful and you will see interface{} all over go codebases.



INTERFACE VALUE

- If a variable is declared with an Interface Type, it may store any value that implements that interface.

```
type duck struct {}
func (d duck) Quack() string {return “quack”}
type loudDuck struct {}
func (d loudDuck) Quack() string {return “QUACK!”}

var q Quacker
q = duck{}       // ok
q = loudDuck{}   // ok
q = Person{}     // ok
q = Employee{}   // ok
q = 1            // not ok, ints don’t do Quack()
```

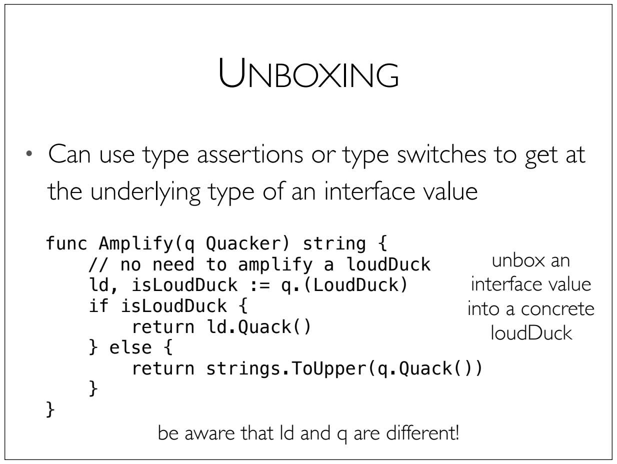
The Three Rules of Interfaces By Rob Pike

- Interfaces define sets of methods. They are pure and abstract: no implementation, no data fields. Go has a clear separation between interface and implementation.
- Interface values are just that: values. They contain any concrete value that implements all the methods defined in the interface. **That concrete value may or may not be a pointer.**
- Types implement interfaces just by having methods. They do not have to declare that they do so. For instance, every type implements the empty interface, interface{}.

More on the Empty Interface{}

- Remember how we say that every type implements the EmptyInterface?
- That means if we want a arbitrary type or container we can use `interface{}`

```
// can put anything in this slice:
s := make([]interface{}, 0)
// can put any value in this map:
m := make(map[string]interface{})
// can call this func with any type for an arg
func DoSomething(x interface{})
```



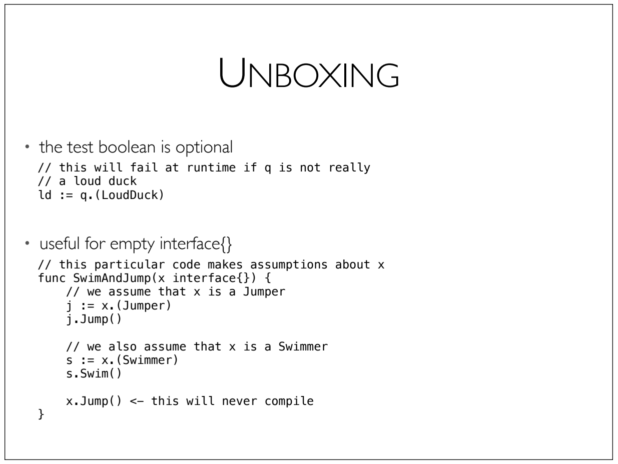
UNBOXING

- Can use type assertions or type switches to get at the underlying type of an interface value

```
func Amplify(q Quacker) string {
    // no need to amplify a loudDuck
    ld, isLoudDuck := q.(LoudDuck)
    if isLoudDuck {
        return ld.Quack()
    } else {
        return strings.ToUpper(q.Quack())
    }
}
```

unbox an interface value into a concrete loudDuck

be aware that ld and q are different!



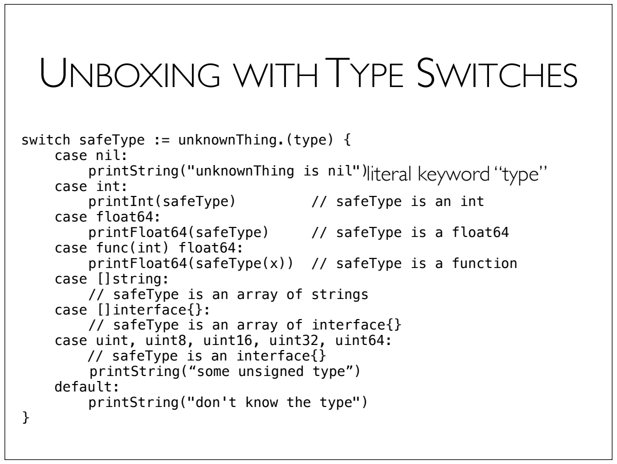
Unboxing

- the test boolean is optional

```
// this will fail at runtime if q is not really
// a loud duck
ld := q.(LoudDuck)
```

- useful for empty interface{}

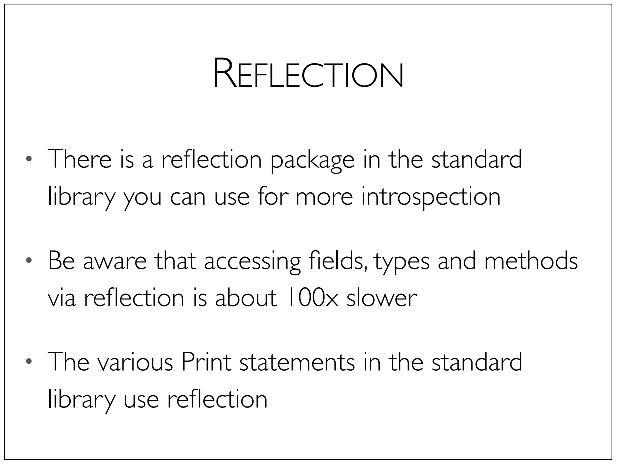
```
// this particular code makes assumptions about x
func SwimAndJump(x interface{}) {
    // we assume that x is a Jumper
    j := x.(Jumper)
    j.Jump()

    // we also assume that x is a Swimmer
    s := x.(Swimmer)
    s.Swim()

    x.Jump() <- this will never compile
}
```



UNBOXING WITH TYPE SWITCHES

```
switch safeType := unknownThing.(type) {
    case nil:
        printString("unknownThing is nil")
    case int:
        printInt(safeType)         // safeType is an int
    case float64:
        printFloat64(safeType)     // safeType is a float64
    case func(int) float64:
        printFloat64(safeType(x))  // safeType is a function
    case []string:
        // safeType is an array of strings
    case []interface{}:
        // safeType is an array of interface{}
    case uint, uint8, uint16, uint32, uint64:
        // safeType is an interface{}
        printString(“some unsigned type”)
    default:
        printString("don't know the type")
}
```

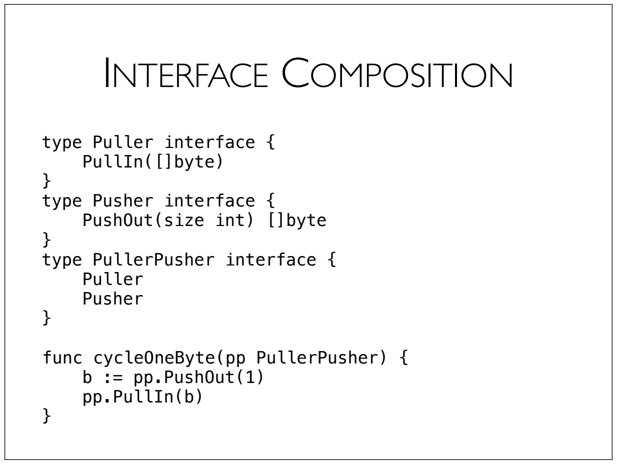
literal keyword “type”



Reflection

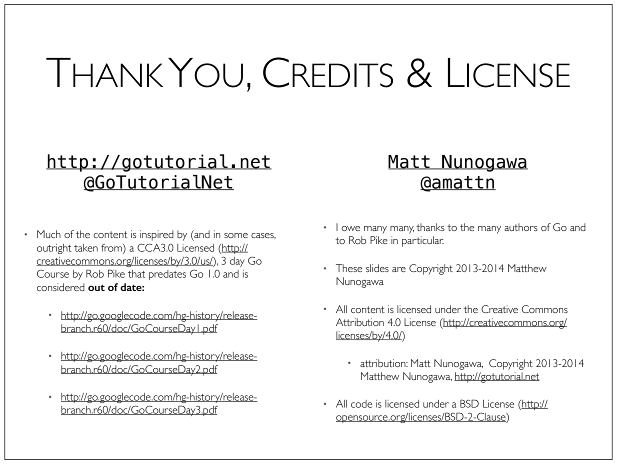
- There is a reflection package in the standard library you can use for more introspection
- Be aware that accessing fields, types and methods via reflection is about 100x slower
- The various Print statements in the standard library use reflection

(typically milliseconds instead of 10s of nanoseconds)



Interface Composition

```
type Puller interface {
    PullIn([]byte)
}
type Pusher interface {
    PushOut(size int) []byte
}
type PullerPusher interface {
    Puller
    Pusher
}

func cycleOneByte(pp PullerPusher) {
    b := pp.PushOut(1)
    pp.PullIn(b)
}
```



Thank You, Credits & License

These are the slides that I used to learn go back in 2011.
"out of date": The actually syntax has not significantly changed. Some of the terminology is no longer in use, typically because after contact with the community, misunderstandings have occurred.

In the creation of these slides, I have, to the utmost of my ability, attempted to make sure that these are correct and updated. Any errors are likely my fault. I make no guarantee that these slides are correct or will remain correct under the inevitable progression of time.