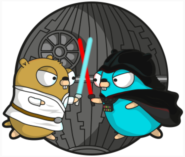
Programming in Go
Lesson 4: Objects, Methods & Interfaces

Matt Holiday
7 May 2019

Cardinal Peak



Lesson #4

What we'll cover today:

- Homework #3
- Object-oriented programming in Go
- Binding methods to objects
- Composing with `struct`
- Interfaces
- Nil and empty interfaces
- Sorting

Homework #3

Sample output: 2144 comics as of 5/4/2019

```
$ go run main.go --get xkcd.json
skipping 404: got 404
skipping 2146: got a 404
skipping 2147: got a 404
. . .
skipping 2198: got a 404
skipping 2199: got a 404
read 2144 comics

$ go run main.go --find "Someone is in bed" xkcd.json
read 2144 comics
https://xkcd.com/571/ 4/20/2009
found 1 comics
```

Homework #3

```go
package main

import (
	"encoding/json"
	"flag"
	"fmt"
	"io"
	"net/http"
	"os"
	"strings"
)
```

Homework #3

```go
// { "month":      "4",
//   "num":        571,
//   "year":       "2009",
//   . . .
//   "transcript": "[[Someone is in bed, . . . long int.",
//   "img":        "https://imgs.xkcd.com/comics/cant_sleep.png",
//   "title":      "Can't Sleep",
//   "day":        "20"
// }

type xkcd struct {
    Num        int    `json:"num"`
    Day        string `json:"day"`
    Month      string `json:"month"`
    Year       string `json:"year"`
    Title      string `json:"title"`
    Transcript string `json:"transcript"`
}
```

Homework #3

```go
func getOne(i int) []byte {
    url := fmt.Sprintf("https://xkcd.com/%d/info.0.json", i)
    resp, err := http.Get(url)

    if err != nil {
        fmt.Fprintf(os.Stderr, "stopped reading: %s\n", err)
        os.Exit(-1)
    }

    defer resp.Body.Close()
```

Homework #3

```go
    if resp.StatusCode != http.StatusOK {
        // easter egg: #404 returns HTTP 404 - not found

        fmt.Fprintf(os.Stderr, "skipping %d: got %d\n",
                    i, resp.StatusCode)
        return nil
    }

    body, err := ioutil.ReadAll(resp.Body)

    if err != nil {
        fmt.Fprintf(os.Stderr, "bad body: %s\n", err)
        os.Exit(-1)
    }

    return body
}
```

Homework #3

```go
var (
    getFlag  = flag.Bool("get", false, "fetch items")
    termFlag = flag.String("find", "", "term to search for")
)

func main() {
    flag.Parse()

    if *getFlag {
        // we're going to get all the entries and write
        // them one at a time to the output file (or stdout)

        var (
            output io.WriteCloser = os.Stdout
            err    error
            cnt    int
            data   []byte
        )
```

Homework #3

```go
if len(flag.Args()) > 0 {
    output, err = os.Create(flag.Arg(0))

    if err != nil {
        fmt.Fprintf(os.Stderr, "bad file: %s", err)
        os.Exit(-1)
    }

    defer output.Close()
}

// the output will be in the form of a JSON array,
// so add the brackets before and after

fmt.Fprint(output, "[")
defer fmt.Fprint(output, "]")
```

Homework #3

```go
for i := 1; i < 2200; i++ {
    if data = getOne(i); data == nil {
        continue
    }

    if cnt > 0 {
        fmt.Fprint(output, ",")  // OB1
    }

    _, err = io.Copy(output, bytes.NewBuffer(data))

    if err != nil {
        fmt.Fprintf(os.Stderr, "stopped: %s", err)
        os.Exit(-1)
    }

    cnt++
}
```

Homework #3

```go
    fmt.Fprintf(os.Stderr, "read %d comics\n", cnt)
    return
}

// if we get here we are doing the "find" function
// let's make sure we've got valid command-line inputs

if len(*termFlag) == 0 {
    fmt.Fprintln(os.Stderr, "no search term")
    os.Exit(-1)
}

if len(flag.Args()) < 1 {
    fmt.Fprintln(os.Stderr, "no file given")
    os.Exit(-1)
}
```

Homework #3

```go
var (
    items []xkcd
    input io.ReadCloser
    cnt   int
    err   error
)

if input, err = os.Open(flag.Arg(0)); err != nil {
    fmt.Fprintf(os.Stderr, "invalid file: %s", err)
    os.Exit(-1)
}

if err = json.NewDecoder(input).Decode(&items); err != nil {
    fmt.Fprintf(os.Stderr, "decode failed: %s\n", err)
    os.Exit(-1)
}

fmt.Fprintf(os.Stderr, "read %d comics\n", len(items))
```

Homework #3

```go
    for _, v := range items {
        if strings.Contains(v.Title, *termFlag) ||
           strings.Contains(v.Transcript, *termFlag) {
            fmt.Printf("https://xkcd.com/%d/ %s/%s/%s\n",
                       v.Num, v.Month, v.Day, v.Year)
            cnt++
        }
    }

    fmt.Fprintf(os.Stderr, "found %d comics\n", cnt)
}
```

Object-Oriented Programming

What does that mean?

For many people, the essential elements of object-oriented programming have been:

- abstraction
- encapsulation
- polymorphism
- inheritance

Sometimes those last two items are combined or confused

Go's approach to OO programming is similar but different

Abstraction

Abstraction: decoupling behavior from the implementation details

The Unix file system API is a great example of effective abstraction

Roughly five basic functions hide all the messy details:

- open
- close
- read
- write
- ioctl

Many different operating system things can be treated like files

Encapsulation

Encapsulation: hiding implementation details from misuse

It's hard to maintain an abstraction if the details are exposed:

- the internals may be manipulated in ways contrary to the concept behind the abstraction
- users of the abstraction may come to depend on the internal details — but those might change

Encapsulation usually means controlling the visibility of names ("private" variables)

Polymorphism

Polymorphism literally means “many shapes” — multiple types behind a single interface

Three main types are recognized:

- ad-hoc: typically found in function/operator overloading
- parametric: this is what generic programming is about
- subtype: subclasses substituting for superclasses

“Protocol-oriented” programming uses explicit interface types, now supported in many popular languages (an ad-hoc method)

In this case, behavior is completely separate from implementation, which is good for abstraction

Inheritance

Inheritance has conflicting meanings:

- substitution (subtype) polymorphism
- structural sharing of implementation details

In theory, inheritance should always imply subtyping:
the subclass should be a “kind of” the superclass

See the Liskov substitution principle

Theories about substitution can be pretty messy

Why would inheritance be bad?

Inheritance injects a dependence on the superclass into the subclass:

- what if the superclass changes behavior?
- what if the subclass never really met the abstract concept?

Deep inheritance hierarchies have proven to be fragile

Not having inheritance means better encapsulation & isolation

“Interfaces will force you to think in term of communication between objects” — Nicolò Pignatelli in Inheritance is evil

See also Composition over inheritance

OO in Go

Go offers four main supports for OO programming:

- encapsulation using the package for visibility control
- abstraction & polymorphism using interface types
- enhanced *composition* to provide structure sharing

Go does not offer inheritance or substitutability based on types

Substitutability is based only on **interfaces**: purely a function of abstract **behavior**

See Go for Gophers

Methods

Methods are type-bound functions

A “method” is a special type of function with a special syntax

It has a “receiver” parameter and uses “dot” notation in calls

```
type Pair struct {
    Path string
    Hash string
}

func (p Pair) String() string {
    return fmt.Sprintf("Hash of %v is %v", p.Path, p.Hash)
}

s := p.String()
```

And so it’s grouped into the “method set” of a type

So why have methods?

This is not just about notation: calling `x.Do()`

Only methods may be used to satisfy an *interface*

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

func (p Pair) String() string {
    return fmt.Sprintf("Hash of %v is %v", p.Path, p.Hash)
}

var s Stringer = p
```

Methods can't be closures because that would be too complex

Not just structs

A method may be defined on any **named** type

That means methods can't be declared on `string`, but

```
type StringSlice []string

func (s StringSlice) String() string {
    // format and print the slice
}
```

The same method name may be bound to different types

The package name is not required to qualify the method name

Make the nil value useful

```go
package collection


type StringStack struct {
    data []string     // "zero" value ready-to-use
}


func (s *StringStack) Push(x string) {
    s.data = append(s.data, x)
}

func (s *StringStack) Size() int {
    return len(s.data)
}
```

Nil as a receiver value

Make `nil` useful: handle it as a receiver when possible

Nothing in Go prevents calling a method with a nil receiver

```
// Sum returns the sum of the list elements.
func (list *IntList) Sum() int {
    if list == nil {
        return 0
    }

    return list.Value + list.Tail.Sum()
}
```

Composition, not Inheritance

Composition

Go allows a new kind of composition (besides a normal field)

An embedded `struct` appears to have its fields live at the same "level" as the structure it becomes a part of *(promotion)*

```go
type Pair struct {
    Path string
    Hash string
}

type PairWithLength struct {
    Pair
    Length int
}

pl := PairWithLength{Pair{"/usr", "0xfdfe"}, 121}
fmt.Println(pl.Path, pl.Length)  // not pl.x.Path
```

Composition

The *methods* of an embedded `struct` are also promoted

Those methods **can't** see fields of the *embedding* `struct`

```
func (p Pair) String() string {
    return fmt.Sprintf("Hash of %v is %v", p.Path, p.Hash)
}

pl := PairWithLength{Pair{"/usr", "0xfdfe"}, 121}

// Pair.String() doesn't have visibility to pl.Length

fmt.Println(pl)   // prints "Hash of /usr is 0xfdfe"
```

Composition

The *methods* of an embedded `struct` are also promoted

Unless the new `struct` later defines the same method on itself

```
pl := PairWithLength{Pair{"/usr", "0xfdfe"}, 121}

fmt.Println(pl) // uses Pair.String()

// now, if we later define the same method

func (p PairWithLength) String() string {
    return fmt.Sprintf("Length of %v is %v with hash %v",
                       p.Path, p.Length, p.Hash)
}

fmt.Println(pl) // would now use PairWithLength.String
```

Composition is not inheritance

A `PairWithLength` **can't** substitute for a `Pair`

They are different and essentially unrelated types

```
func Filename(p Pair) string {
    return filepath.Base(p.Path)
}

pl := PairWithLength{Pair{"/usr", "0xfdfe"}, 121}

a := Filename(pl) // NOT ALLOWED even though pl.Path exists
```

The only substitution is through interface types!

Composition with pointer types

A struct can embed a pointer to another type; promotion of its fields and methods works the same way

```go
type Fizgig struct {
    *PairWithLength
    Broken bool
}

fg := Fizgig{
    &PairWithLength{
        Pair{"/usr", "0xfdfe"},
        121,
    },
    false,
}

fmt.Println(fg)
// Length of /usr is 121 with hash 0xfdfe
```

Interface Types

Interfaces

An interface type is a collection (aggregation) of methods

They define an abstraction through behavior, like *abstract classes* in other languages

Any type which defines both these methods satisfies the interface:

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

This is known as *structural* typing (“duck” typing)

No type will declare itself to implement `ReadWriter` explicitly

Interface variables

A variable of interface type can refer to any object that satisfies it

```
func io.Copy(w Writer, r Reader) (int, error) {
    . . .
}

f1, err := os.Open("input.txt")
f2, err := os.Open("output.txt")

n, err := io.Copy(f2, f1)
```

Here `w` and `r` are references ultimately to files

But it could be a `File` and a `bytes.Buffer` source; it wouldn't care — all it needs is the specific behaviors (write & read)

Interfaces

Interfaces are the basis for *substitutability* in Go

```go
type ByteCounter int

func (b *ByteCounter) Write(p []byte) (int, error) {
    *b += ByteCounter(len(p)) // convert int to ByteCounter
    return len(p), nil
}

var c ByteCounter

f, _ := os.Open("input.txt")
n, _ := io.Copy(c, f)
```

Lots of types are `Writers` and can be written/copied to;
see also Francesc Campoy Understanding Go Interfaces

Interfaces

An HTTP handler function is an instance of an interface

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}

type HandlerFunc func(ResponseWriter, *Request)

func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    f(w, r)
}

// handler matches type HandlerFunc and so interface Handler
// so the HTTP framework can call ServeHTTP on it

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, world! from %s\n", r.URL.Path[1:])
}
```

Interfaces

Interfaces are just types & values orthogonal to the rest of Go

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}

type HandlerFunc func(ResponseWriter, *Request)

func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    f(w, r)
}
```

Interfaces separate data from behavior (classes conflate them)

They provide a model of genericity (but not quite parametric polymorphism)

Interfaces and substitution

All the methods must be present to satisfy the interface

```
var w io.Writer

w = os.Stdout           // OK: *os.File has Write method
w = new(bytes.Buffer)   // OK: *bytes.Buffer has Write method
w = time.Second         // ERROR: no Write method

var rwc io.ReadWriteCloser

rwc = os.Stdout         // OK: *os.File has all 3 methods
rwc = new(bytes.Buffer) // ERROR: no Close method

w = rwc                 // OK: io.ReadWriteCloser has Write
rwc = w                 // ERROR: no Close method
```

Interface satisfiability

And the must be of the right type (pointer or value)

```go
type IntSet struct { /* ... */ }

func (*IntSet) String() string

var _ = IntSet{}.String() // ERROR: String needs *IntSet

var s IntSet
var _ = s.String()        // OK: s is a variable and
                          // &s has String

var _ fmt.Stringer = &s   // OK
var _ fmt.Stringer = s    // ERROR: no String method
```

Interface also offer composition

`ReadWriter` is actually defined by Go as two interfaces

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

type Writer interface {
    Write(p []byte) (n int, err error)
}

type ReadWriter interface {
    Reader
    Writer
}
```

Small interfaces with composition where needed are more flexible

Nil interfaces

An interface variable is `nil` until initialized

It really has two parts:

- a value or pointer of some type
- a pointer to information about the type so the correct actual method can be identified

```
var r io.Reader     // nil until initialized
var b bytes.Buffer  // ditto

r := b              // r is no longer nil!
                    // but it has a nil pointer to a Buffer
```

This causes a lot of confusion!

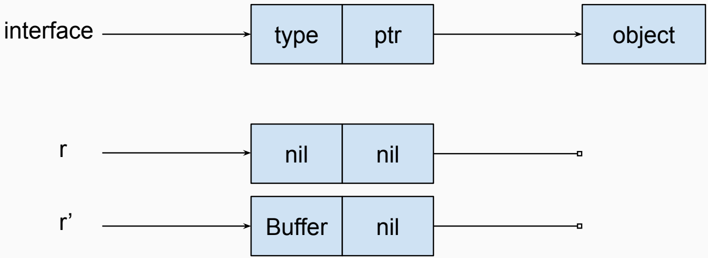
An interface variable is `nil` only if both parts are

Nil interfaces

An interface variable is `nil` until initialized

It really has two parts:

- a value or pointer of some type
- a pointer to information about the type so the correct actual method can be identified



Error is really an interface

We called `error` a special type, but it's really an interface

```
type error interface {
    func Error() string
}
```

We can compare it to `nil` unless we make a mistake

```
var err error

func XYZ(a int) (int, *errFoo) {
    return 0, nil
}

n, err := XYZ(1)   // BAD: interface gets a nil concrete ptr
```

See Why is my nil error value not equal to nil?

Empty interfaces

The `interface{}` type has no methods

So it is satisfied by anything!

Empty interfaces are commonly used; they're how the formatted I/O routines can print any type

```
func fmt.Printf(f string, args ...interface{})
```

Reflection is needed to determine what the concrete type is

We'll talk about that later

Sorting

Sortable interface

`sort.Interface` is defined as

```go
type Interface interface {
    // Len is the number of elements in the collection.
    Len() int

    // Less reports whether the element with
    // index i should sort before the element with index j.
    Less(i, j int) bool

    // Swap swaps the elements with indexes i and j.
    Swap(i, j int)
}
```

and `sort.Sort` as

```go
func Sort(data Interface)
```

Sortable built-ins

Slices of strings can be sorted using `StringSlice`

```
//  defined in the sort package
//  type StringSlice []string

entries := []string{"charlie", "able", "dog", "baker"}

sort.Sort(sort.StringSlice(entries))

fmt.Println(entries)   // [able baker charlie dog]
```

Sorting example

Implement `sort.Interface` to make a type sortable:

```
type Organ struct {
    Name   string
    Weight int
}

type Organs []Organ

func (s Organs) Len() int      { return len(s) }
func (s Organs) Swap(i, j int) { s[i], s[j] = s[j], s[i] }
```

From Andrew Gerrand's Go for gophers

Sorting example

Implement `sort.Interface` to make a type sortable:

```
type ByName struct{ Organs }

func (s ByName) Less(i, j int) bool {
    return s.Organs[i].Name < s.Organs[j].Name
}

type ByWeight struct{ Organs }

func (s ByWeight) Less(i, j int) bool {
    return s.Organs[i].Weight < s.Organs[j].Weight
}
```

Here we use *struct composition* which promotes the `Organs` methods

Sorting example

Make a struct of the correct type on the fly to sort:

```
s := []Organ{
    {"brain", 1340},
    {"heart", 290},
    {"liver", 1494},
    {"pancreas", 131},
    {"spleen", 162},
}

sort.Sort(ByWeight{s})      // pancreas first
fmt.Println(s)

sort.Sort(ByName{s})        // brain first
fmt.Println(s)
```

```
[{pancreas 131} {spleen 162} {heart 290} {brain 1340} {liver 1494}]
[{brain 1340} {heart 290} {liver 1494} {pancreas 131} {spleen 162}]
```

Sorting in reverse

Use `sort.Reverse` which is defined as:

```go
type reverse struct {
    // This embedded Interface permits Reverse to use the
    // methods of another Interface implementation.
    Interface
}

// Less returns the opposite of the embedded implementation's
// Less method.
func (r reverse) Less(i, j int) bool {
    return r.Interface.Less(j, i)
}

// Reverse returns the reverse order for data.
func Reverse(data Interface) Interface {
    return &reverse{data}
}
```

Sorting in reverse

Let's use `StringSlice` again:

```
//  defined in the sort package
//  type StringSlice []string

entries := []string{"charlie", "able", "dog", "baker"}

sort.Sort(sort.Reverse(sort.StringSlice(entries)))

fmt.Println(entries)   // [dog charlie baker able]
```

Details, Details

Pointer vs value receivers

A method can be defined on a pointer to a type

In this case, the receiver is passed "by reference"

```
type Point struct {
    x,y float32
}

func (p *Point) Add(x, y float32) {
    p.x, p.y = p.x + x, p.y + y
}

func (p Point) OffsetOf(p1 Point) (x float32, y float32) {
    x, y = p.x - p1.x, p.y - p1.y
    return
}
```

The same method name may **not** be bound to both `T` and `*T`

Pointer vs value receivers

Pointer methods may be called on non-pointers and vice versa

Go will automatically use * or & as needed

```
p1 := new(Point)     // *Point, at (0,0)
p2 := Point{1, 1}

p1.OffsetOf(p2)      // same as (*p1).OffsetOf(p2)

p2.Add(3, 4)         // same as (&p2).Add(3, 4)
```

Except & may only be applied to objects that are *addressable*

Pointer vs value receivers

Compatibility between objects and receiver types

	Pointer	**L-Value**	**R-Value**
pointer receiver	OK	OK &	**Not OK**
value receiver	OK *	OK	OK

A method requiring a pointer receiver may only be called on an addressable object

```
var p Point

p.Add(1, 2)                // OK, &p
Point{1, 1}.Add(2, 3)      // Not OK, can't take address
```

Consistency in receiver types

If a type has a method with a pointer receiver, then all its methods should take pointers

And in general objects of that type are probably not safe to copy

```
type Buffer struct {
    buf       []byte
    off       int
    . . .
}

func (b *Buffer) ReadString(delim byte) (string, error) {
    . . .
}
```

Copying a `Buffer` object will cause both copies to reference the same underlying byte slice — bad news in most cases

Method values

A selected method may be passed similar to a closure; the receiver is closed over at that point

```
func (p Point) Distance(q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}

p := Point{1, 2}
q := Point{4, 6}

distanceFromP := p.Distance    // this is a method value

fmt.Println(distanceFromP(q))  // and can be called later
```

The value of `p` is copied into the method value because `Distance` takes a *value* receiver; if `p` took a pointer receiver, it would have a reference to `p` and calculate based on `p`'s most current value

Homework

Homework #4

Exercise 7.11 from *GOPL:* web front-end for a database

Add additional handlers [to the database example in §7.7, which is program `gopl.io/ch7/http4`] so that clients can create, read, update, and delete database entries. For example, a request of the form

```
/update?item=socks&price=6
```

will update the price of an item in the inventory and report an error if the item does not exist or if the price is invalid.

(Warning: this change introduces concurrent variable updates — but we will *ignore* the race conditions for the purpose of this exercise.)

(Note: I dropped the `price` method from my solution, since it overlaps the new `read` operation.)