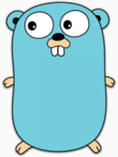
Programming in Go

Matt Holiday
Christmas 2020



Composition, not Inheritance

Composition

The fields of an **embedded** `struct` are *promoted* to the level of the embedding structure

```
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 *embedding* structure may declare the same methods and so override the promoted methods

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

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

// now define the String() 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) // Length of /usr is 121 with hash 0xfdfe
```

Composition is not inheritance

A `PairWithLength` **“has a”** `Pair` but it isn’t one and **is not** *substitutable* for `Pair`

```go
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 is not inheritance

We can make an interface that `PairWithLength` will satisfy with a method promoted from `Pair`

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

interface Filenamer {
    Filename() string
}

// this works because Pair's method is promoted

var fn Filenamer = PairWithLength{Pair{"/usr", "0xfdfe"}, 121}

name := fn.Filename()
```

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
```

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`

```go
//  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:

```go
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:

```go
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]
```

Make Nil Useful

Make the nil value useful

```go
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) Pop() string {
    if l := len(s.data); l > 0 {
        t := s.data[l-1]
        s.data = s.data[:l-1]
        return t
    }

    panic("pop from empty stack")
}
```

Nil as a receiver value

Nothing in Go prevents calling a method with a `nil` receiver

```
type IntList struct {
    Value int
    Tail  *IntList
}

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

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