Part 19: Interfaces - II

17 JUNE 2017

This tutorial was updated on September 3rd, 2017.

Welcome to tutorial no. 19 in Golang tutorial series. This is the second part in our 2 part interface tutorial. In case you missed the first part, you can read it from here https://golangbot.com/interfaces-part-1/

Implementing interfaces using pointer receivers vs value receivers

All the example interfaces we discussed in part 1 were implemented using value receivers. It is also possible to implement interfaces using pointer receivers. There is a subtlety to be noted while implementing interfaces using pointer receivers. Lets understand that using the following program.

```
1  package main
2
3  import "fmt"
4
5  type Describer interface {
6      Describe()
7  }
8  type Person struct {
9      name string
10     age  int
11 }
12
13 func (p Person) Describe() { //implemented using va
14     fmt.Printf("%s is %d years old\n", p.name, p.ag
15 }
16
17 type Address struct {
18     state   string
19     country string
20 }
21
22 func (a *Address) Describe() { //implemented using
23     fmt.Printf("State %s Country %s", a.state, a.co
24 }
25
26 func main() {
27     var d1 Describer
28     p1 := Person{"Sam", 25}
29     d1 = p1
30     d1.Describe()
31     p2 := Person{"James", 32}
32     d1 = &p2
33     d1.Describe()
34
35     var d2 Describer
36     a := Address{"Washington", "USA"}
37
38     /* compilation error if the following line is
39        uncommented
40        cannot use a (type Address) as type Describe
41        in assignment: Address does not implement
42        Describer (Describe method has pointer
43        receiver)
44     */
45     //d2 = a
46
47     d2 = &a //This works since Describer interface
48     //is implemented by Address pointer in line 22
49     d2.Describe()
50
51 }
```

Run in playground

In the program above, the `Person` struct implements the `Describer` interface using value receiver in line no. 13.

As we have already learnt during our discussion about methods, methods with value receivers accept both pointer and value receivers. *It is legal to call a value method on anything which is a value or whose value can be dereferenced.*

p1 is a value of type `Person` and it is assigned to `d1` in line no. 29. `Person` implements the `d1` interface and hence line no. 30 will print `Sam is 25 years old`.

Similarly `d1` is assigned to `&p2` in line no. 32 and hence line no. 33 will print `James is 32 years old`. Awesome :).

The `Address` struct implements the `Describer` interface using pointer receiver in line no. 22.

If line. no 45 of the program above is uncommented, we will get the compilation error **main.go:42: cannot use a (type Address) as type Describer in assignment: Address does not implement Describer (Describe method has pointer receiver)**. This is because, the `Describer` interface is implemented using a Address Pointer receiver in line 22 and we are trying to assign `a` which is a value type and it has not implemented the `Describer` interface. This will definitely surprise you since we learnt earlier that methods with pointer receivers will accept both pointer and value receivers. Then why isn't the code in line no. 45 working.

The reason is that it is legal to call a pointer-valued method on anything that is already a pointer or whose address can be taken. The concrete value stored in an interface is not addressable and hence it is not possible for the compiler to automatically take the address of `a` in line no. 45 and hence this code fails.

Line no. 47 works because we are assigning the address of `a` `&a` to `d2`.

The rest of the program is self explanatory. This program will print,

```
Sam is 25 years old
James is 32 years old
State Washington Country USA
```


Implementing multiple interfaces

A type can implement more than one interface. Lets see how this is done in the following program.

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  type SalaryCalculator interface {
8      DisplaySalary()
9  }
10
11 type LeaveCalculator interface {
12     CalculateLeavesLeft() int
13 }
14
15 type Employee struct {
16     firstName string
17     lastName string
18     basicPay int
19     pf int
20     totalLeaves int
21     leavesTaken int
22 }
23
24 func (e Employee) DisplaySalary() {
25     fmt.Printf("%s %s has salary $%d", e.firstName,
26 }
27
28 func (e Employee) CalculateLeavesLeft() int {
29     return e.totalLeaves - e.leavesTaken
30 }
31
32 func main() {
33     e := Employee {
34         firstName: "Naveen",
35         lastName: "Ramanathan",
36         basicPay: 5000,
37         pf: 200,
38         totalLeaves: 30,
39         leavesTaken: 5,
40     }
41     var s SalaryCalculator = e
42     s.DisplaySalary()
43     var l LeaveCalculator = e
44     fmt.Println("\nLeaves left =", l.CalculateLeave
45 }
```

Run in playground

The program above has two interfaces `SalaryCalculator` and `LeaveCalculator` declared in lines 7 and 11 respectively.

The `Employee` struct defined in line no. 15 provides implementations for the `DisplaySalary` method of `SalaryCalculator` interface in line no. 24 and the `CalculateLeavesLeft` method of `LeaveCalculator` interface interface in line no. 28. Now `Employee` implements both `SalaryCalculator` and `LeaveCalculator` interfaces.

In line no. 41 we assign `e` to a variable of type `SalaryCalculator` interface and in line no. 43 we assign the same variable `e` to a variable of type `LeaveCalculator`. This is possible since `e` which of type `Employee` implements both `SalaryCalculator` and `LeaveCalculator` interfaces.

This program outputs,

```
Naveen Ramanathan has salary $5200
Leaves left = 25
```

Embedding interfaces

Although go does not offer inheritance, it is possible to create a new interfaces by embedding other interfaces.

Lets see how this is done.

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  type SalaryCalculator interface {
8      DisplaySalary()
9  }
10
11 type LeaveCalculator interface {
12     CalculateLeavesLeft() int
13 }
14
15 type EmployeeOperations interface {
16     SalaryCalculator
17     LeaveCalculator
18 }
19
20 type Employee struct {
21     firstName string
22     lastName string
23     basicPay int
24     pf int
25     totalLeaves int
26     leavesTaken int
27 }
28
29 func (e Employee) DisplaySalary() {
30     fmt.Printf("%s %s has salary $%d", e.firstName,
31 }
32
33 func (e Employee) CalculateLeavesLeft() int {
34     return e.totalLeaves - e.leavesTaken
35 }
36
37 func main() {
38     e := Employee {
39         firstName: "Naveen",
40         lastName: "Ramanathan",
41         basicPay: 5000,
42         pf: 200,
43         totalLeaves: 30,
44         leavesTaken: 5,
45     }
46     var empOp EmployeeOperations = e
47     empOp.DisplaySalary()
48     fmt.Println("\nLeaves left =", empOp.CalculateL
49 }
```

Run in playground

EmployeeOperations interface in line 15 of the program above is created by embedding *SalaryCalculator* and *LeaveCalculator* interfaces.

Any type is said to implement `EmployeeOperations` interface if it provides method definitions for the methods present in both *SalaryCalculator* and *LeaveCalculator* interfaces.

The `Employee` struct implements `EmployeeOperations` interface since it provides definition for both `DisplaySalary` and `CalculateLeavesLeft` methods in lines 29 and 33 respectively.

In line 46, `e` of type `Employee` is assigned to `empOp` of type `EmployeeOperations`. In the next two lines, the `DisplaySalary()` and `CalculateLeavesLeft()` methods are called on `empOp`.

This program will output

```
Naveen Ramanathan has salary $5200
Leaves left = 25
```

Zero value of Interface

The zero value of a interface is nil. A nil interface has both its underlying value and as well as concrete type as nil.

```
package main

import "fmt"

type Describer interface {
    Describe()
}

func main() {
    var d1 Describer
    if d1 == nil {
        fmt.Printf("d1 is nil and has type %T value
    }
}
```

Run in playground

d1 in the above program is `nil` and this program will output

```
d1 is nil and has type <nil> value <nil>
```

If we try to call a method on the `nil` interface, the program will panic since the `nil` interface neither has a underlying value nor a concrete type.

```
package main

type Describer interface {
    Describe()
}

func main() {
    var d1 Describer
    d1.Describe()
}
```

Run in background

Since `d1` in the program above is `nil`, this program will panic with runtime error **panic: runtime error: invalid memory address or nil pointer dereference [signal SIGSEGV: segmentation violation code=0xffffffff addr=0x0 pc=0xc8527]"**

Thats it for interfaces. Have a good day.

Next tutorial - Introduction to Concurrency

Naveen Ramanathan

iOS developer at dietco.de and Golang enthusiast.

Share this post

About

For any queries/suggestions, please contact us at **naveen[at]golangbot[dot]com**

Follow Us

Newsletter

Join Our Newsletter

Signup for our newsletter and get the **Golang tools cheat sheet for free**.

Email*

Subscribe