Backend Programming

M. Yauri M. Attamimi

Upgrade your potential !

Foundation Class

- WEEK 2 (Session 1) -

Anonymous Functions

Anonymous Function

```go
package main

import "fmt"

func main() {
    message := "Hi Gopher!"

    func(m string) {
        fmt.Println(m)
    }(message)
}
```

Passing Anonymous Function

```go
package main

import "fmt"

func someFunction(f func(string) string) {
    result := f("G2LAB")
    fmt.Println(result)
}

func main() {
    anon := func(name string) string {
        return name + "'s Gopher"
    }
    someFunction(anon)
}
```

Closures

Function literals in Go are closures: they may refers to variables defined in an enclosing function. Such variables:

- are shared between the surrounding function and the function literal
- survive as long as they are accessible

```go
package main

import "fmt"

func main() {
    n := 0
    f1 := func() int {
        n++
        return n
    }
    fmt.Println(f1())
    fmt.Println(f1())
    fmt.Println(f1())
}
```

```go
package main

import "fmt"

func new() func() {
    n := 0
    return func() {
        n++
        fmt.Println(n)
    }
}

func main() {
    f1, f2 := new(), new()
    f1()
    f2()
    f1()
    f2()
}
```

Variadic Functions

Definition

⇒ a function that can accept variable number of arguments/parameters
⇒ last parameter is denoted by `...T`

Example

```
func append(slice []Type, elems ...Type) []Type
```

Method

Basic Syntax

```
func (t Type) methodName(parameter list) {
}
```

A method is just a function with a special receiver type that is written between the func keyword and the method name. The receiver can either be a struct type or non-struct type.

Example

```
package main

import "fmt"

type boxer struct {
    name    string
    ranking string
}

func (b boxer) printBoxer() {
    fmt.Printf("%s is a %s champion\n", b.name, b.ranking)
}

func main() {
    b1 := &boxer{"Dick Tiger", "Middleweight"}
    b1.printBoxer()
}
```

Why Method?

- Go is not pure OOP and doesn’t support classes. Methods on types is a way to achieve behavior similar to classes.

- Methods with same name can be defined on different types whereas functions with the same names are not allowed.

```go
package main

import (
    "fmt"
    "math"
)

type rectangle struct {
    length, width int
}

type circle struct {
    radius float64
}

func (r rectangle) Area() int {
    return r.length * r.width
}

func (c circle) Area() float64 {
    return math.Pi * c.radius * c.radius
}

func main() {
    r := rectangle{10, 5}
    c := circle{12}
    fmt.Printf("Area of Rectangle %d\n", r.Area())
    fmt.Printf("Area of Circle %.2f\n", c.Area())
}
```

Value Receivers vs Pointer Receivers

```go
type superhero struct {
    name, city string
}

func (h superhero) changeHero(newName, newCity string) {
    h.name = newName
    h.city = newCity
}

func main() {
    hero := superhero{"Superman", "Metropolis"}
    fmt.Println("Our hero:", hero)
    hero.changeHero("Flash", "Central City")
    if hero == (superhero{"Superman", "Metropolis"}) {
        fmt.Println("Our hero is still the same hero")
    }
}
```

```go
type superhero struct {
    name, city string
}

func (h *superhero) changeHero(newName, newCity string) {
    h.name = newName
    h.city = newCity
}

func main() {
    hero := superhero{"Superman", "Metropolis"}
    fmt.Println("Our hero:", hero)
    hero.changeHero("Flash", "Central City")
    if hero != (superhero{"Superman", "Metropolis"}) {
        fmt.Println("Our hero now is:", hero)
    }
}
```

Methods of Anonymous Fields

Methods belonging to anonymous fields of a struct can be called as if they belong to the structure where the anonymous field is defined.

```
package main

import "fmt"

type origin struct {
    city, country string
}

func (o origin) fullOrigin() {
    fmt.Printf("Originated story from: %s - %s\n", o.city, o.country)
}

type superhero struct {
    name string
    origin
}

func main() {
    h := superhero{"Captain America", origin{"New York City", "USA"}}
    fmt.Print(h.name, " is ")
    h.fullOrigin()
}
```

Value Receivers vs Value Arguments

- When a function has a value argument, it will accept only a value argument (cannot accept pointer)
- When a method has a value receiver, it will accept both pointer and value receivers.

```
type superhero struct {
    name string
}

func show(s superhero) {
    fmt.Printf("function is calling %s\n", s.name)
}

func (s superhero) show() {
    fmt.Printf("method is calling %s\n", s.name)
}

func main() {
    h := superhero{"Hulk"}
    show(h)
    h.show()
    fmt.Printf("\nCreating new pointer for our superhero\n\n")
    p := &h
    //show(p) //compiled error
    p.show() // implicitly called as (*p).show()
}
```

Pointer Receivers vs Pointer Arguments

- When a function has a pointer argument, it will accept only a pointer argument (cannot accept value)
- When a method has a pointer receiver, it will accept both pointer and value receivers.

```go
type superhero struct {
    name string
}

func show(s *superhero) {
    fmt.Printf("function is calling %s\n", s.name)
}

func (s *superhero) show() {
    fmt.Printf("method is calling %s\n", s.name)
}

func main() {
    h := &superhero{"Hulk"}
    show(h)
    h.show()
    fmt.Printf("\ncreating new value for our superhero\n\n")
    v := superhero{"Hulk"}
    //show(v)  // compiled error
    v.show() // implicitly called as (&v).show()
}
```

A Few Additional Notes

- We can also define methods on non-struct types.
- To define a method on a type, the definition of the receiver type of the method and the definition of the method should be in the same package.

```
package main

import "fmt"

type myInt int

func (a myInt) add(b myInt) myInt {
    return a + b
}

func main() {
    num1 := myInt(5)
    num2 := myInt(10)
    fmt.Printf("num1 + num2 = %d\n", num1.add(num2))
}
```

Interface

Definition

- In terms of OOP, interface defines a set of an object behaviors, which specifies what the object is supposed to do.

- In Go, interface is a set of method signatures. Interface specifies what methods a type should have and the type decides how to implement those methods.

```go
type vowelsFinder interface {
    FindVowels() []rune
}

type dcHero string

func (h dcHero) FindVowels() []rune {
    var vowels []rune
    for _, v := range h {
        switch v {
        case 'a', 'i', 'u', 'e', 'o', 'A', 'I', 'U', 'E', 'O':
            vowels = append(vowels, rune(v))
        }
    }
    return vowels
}

func main() {
    var v vowelsFinder
    hero1 := dcHero("Superman")
    hero2 := dcHero("GI-JOE")
    hero3 := dcHero("Flash")
    v = hero1
    fmt.Printf("%s Vowels are %c\n", hero1, v.FindVowels())
    v = hero2
    fmt.Printf("%s Vowels are %c\n", hero2, v.FindVowels())
    v = hero3
    fmt.Printf("%s Vowels are %c\n", hero3, v.FindVowels())
}
```

Empty Interface

- An interface which has zero methods is called empty interface and represented as `interface{}`.

- All types implements the empty interface. it can be considered as supertype for all types such as `java.lang.Object` in the Java class hierarchy.

```
func describe(i interface{}) {
    fmt.Printf("Type = %T, value = %v\n", i, i)
}

func main() {
    s := "Hi Gopher"
    describe(s)
    i := 99
    describe(i)
    strt := struct {
        name string
    }{
        name: "G2Lab",
    }
    describe(strt)
}
```

Type Assertion

- is used to extract the underlying value of the interface.
- syntax : i.(T)

```
func assert(i interface{}) {
    s := i.(int)
    fmt.Println(s)
}

func main() {
    var s interface{} = 99
    assert(s)
}
```

Could you guess the output?

```go
func assert(i interface{}) {
    s := i.(int)
    fmt.Println(s)
}

func main() {
    var s interface{} = "Hi Gopher"
    assert(s)
}
```

Avoiding “panic” at runtime

```
func assert(i interface{}) {
    v, ok := i.(int)
    if !ok {
        fmt.Printf("invalid type conversion! cannot convert %T to int.", i)
    } else {
        fmt.Println(v)
    }
}

func main() {
    var s interface{} = "Hi Gopher"
    assert(s)
}
```

Type Switch

```
func findType(i interface{}) {
    switch i.(type) {
    case string:
        fmt.Printf("interface is of type string, has value: %s\n", i.(string))
    case int:
        fmt.Printf("interface is of type int, has value: %d\n", i.(int))
    default:
        fmt.Printf("Detected as %T by runtime.\n", i)
    }
}

func main() {
    findType("G2LAB")
    findType(99)
    findType(99.77)
}
```

Compare a Type to an Interface (I)

```
type person string

type warrior struct {
    name, weapon string
}

type player interface {
    attack()
    changeWeapon(string)
}

func (p person) attack() {
    fmt.Printf("%s attacks with his speed as a football player\n", p)
}

func (p person) changeWeapon(string) {
    fmt.Printf("%s doesn't have any weapon\n", p)
}

func (p warrior) attack() {
    if p.weapon == "" {
        fmt.Printf("%s attacks with an empty hand\n", p.name)
    } else {
        fmt.Printf("%s attacks with %s\n", p.name, p.weapon)
    }
}

func (p *warrior) changeWeapon(w string) {
    p.weapon = w
}
```

Compare a Type to an Interface (II)

```
func react(i interface{}) {
    switch v := i.(type) {
    case player:
        v.attack()
    default:
        fmt.Println("Unknown player capability...")
    }
}

func main() {
    var p player
    p1 := person("Ronaldo")
    p2 := warrior{name: "Wong Kai Yin"}
    p = p1
    react(p)
    p = &p2
    p.changeWeapon("Long Spears")
    react(p)
}
```

Implement Multiple Interfaces

```go
type GamePlayer interface {
    ChangeGameLevel(int)
}

type GameCharacter interface {
    ChangeCharacter(string)
}

type person string

func (p person) ChangeGameLevel(level int) {
    fmt.Printf("%s changes game level to level #%d\n", p, level)
}

func (p person) ChangeCharacter(character string) {
    fmt.Printf("%s changes game character to %s\n", p, character)
}

func main() {
    p := person("Yauri")
    var gp GamePlayer = p
    gp.ChangeGameLevel(9)
    var gc GameCharacter = p
    gc.ChangeCharacter("Ryu Hayabusa")
}
```

Embedding Interface

```
type GamePlayer interface {
    ChangeGameLevel(int)
}

type GameCharacter interface {
    ChangeCharacter(string)
}

type Game interface {
    GamePlayer
    GameCharacter
}

type person string

func (p person) ChangeGameLevel(level int) {
    fmt.Printf("%s changes game level to level #%d\n", p, level)
}

func (p person) ChangeCharacter(character string) {
    fmt.Printf("%s changes game character to %s\n", p, character)
}

func main() {
    p := person("Yauri")
    var game Game = p
    game.ChangeGameLevel(9)
    game.ChangeCharacter("Ryu Hayabusa")
}
```

Check the exercises at https://exercism.io

SUMMARY

You have studied how to build various function in Go, including how to develop methods and design your software contract by using an interface as a similar way to achieve an Object Oriented Programming in Go.

Thank You