210515 A Tour of Go - Methods and interfaces

Methods

Value Receivers

- Go does not have classes. However, but can define methods on types.
- A methods is a function with a special receiver argument.
- The receiver appears in its own argument list between the `func` keyword and the method name.

```
package main

import (
    "fmt"
    "math"
)

type Vertex strcut {
    X, Y float64
}

func (v Vertex) Abs() float64 {
return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
    v := Vertex{3, 4}
    fmt.Println(v.Abs()) // 5
}
```

We can only declare a method

```
package main

import (
    "fmt"
    "math"
)

type MyFloat float64

func (f MyFloat) Abs() float64 {
    if f < 0 {
        return float64(-f)
    }
    return float64(f)
```

```
}

func main() {
    f := MyFloat(-math.Sqrt2)
    fmt.Println(f.Abs())
}
```

Pointer Recivers

```
package main

import (
    "fmt"
    "math"
)

type Vertex struct {
    X, Y float64
}

func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func (v *Vertex) Scale(f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}

func main() {
    v := Vertex{3, 4}
    v.Scale(10)
    fmt.Println(v.Abs())
}
```

Pointers and functions

passing the object as an argument is an alternative
to modify the outside object, we must pass is as a pointer, otherwise, it's a copy

```
package main

import (
    "fmt"
    "main"
)

type Vertex struct {
    X, Y float64
}

func Abs(v Vertex) float64 {
    return math.Sqrt(v.x*v.X + v.Y*v.Y)
}

func Scale(v *Vertex, f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}

func main() {
    v := Vertex {3, 4}
    Scale(&v, 10)
    fmt.Println(Abs(v))
}
```

Methods and pointer indirection

methods with pointer receivers take either a value or a pointer as the receiver when they are called.

```
```

```go
package main

import "fmt"

type Vertex struct {
    X, Y float64
}

func (v *Vertex) Scale(f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}

func ScaleFunc(v *Vertex, f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}

func main() {
    v := Vertex{3, 4}
    v.Scale(2)
    ScaleFunc(&v, 10)

    p := &Vertex{4, 3}
    p.Scale(3)
    ScaleFunc(p, 8)

    fmt.Println(v, p)
}
```

methods with value receivers take either a value or a pointer as the receiver when they are called

```go
package main
```

```
import (
    "fmt"
    "math"
)

type Vertex struct {
    X, Y float64
}

func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func AbsFunc(v Vertex) float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
    v := Vertex{3, 4}
    fmt.Println(v.Abs())
    fmt.Println(AbsFunc(v))

    p := &Vertex{4, 3}
    fmt.Println(p.Abs())
    fmt.Println(AbsFunc(*p))
}
```

Interfaces

An interface type is defined as a set of method signatures.

A value of interface type can hold any value that implements those methods.

```
package main
```

```go
import (
    "fmt"
    "math"
)

type Abser interface {
    Abs() float64
}

func main() {
    var a Abser
    f := MyFloat(-math.Sqrt2)
    v := Vertex{3, 4}

    a = f  // a MyFloat implements Abser
    a = &v // a *Vertex implements Abser

    // In the following line, v is a Vertex (not *Vertex)
    // and does NOT implement Abser.
    a = v

    fmt.Println(a.Abs())
}

type MyFloat float64

func (f MyFloat) Abs() float64 {
    if f < 0 {
        return float64(-f)
    }
    return float64(f)
}

type Vertex struct {
    X, Y float64
}
```

```
func (v *Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```

Interfaces are implemented implicitly

Implicit interfaces decouple the definition of an interface from its implementation, which could then appear in any package without prearrangement.

```
package main

import "fmt"

type I interface {
    M()
}

type T struct {
    S string
}

// This method means type T implements the interface I,
// but we don't need to explicitly declare that it does so.
func (t T) M() {
    fmt.Println(t.S)
}

func main() {
    var i I = T{"hello"}
    i.M()
}
```

Interface Values

(value type)

```
package main

import (
    "fmt"
    "math"
)

type I interface {
    M()
}

type T struct {
    S string
}

func (t *T) M() {
    fmt.Println(t.S)
}

type F float64

func (f F) M() {
    fmt.Println(f)
}

func main() {
    var i I

    i = &T{"Hello"}
    describe(i)
    i.M()

    i = F(math.Pi)
```

```
    describe(i)
    i.M()
}

func describe(i I) {
    fmt.Printf("(%v, %T)\n", i, i)
}
```

interfaces are reresented as tuples of value and underlying type.

```
(&{Hello}, *main.T)
Hello
(3.141592653589793, main.F)
3.141592653589793
```

interface value cannot be nil, but the value inside the tuple can be nil, so lang as not defined. It is still safe to call methods of interface values with nil value in tuple.

```
package main

import "fmt"

type I interface {
    M()
}

type T struct {
    S string
}

func (t *T) M() {
```

```
    if t == nil {
        fmt.Println("<nil>")
        return
    }
    fmt.Println(t.S)
}

func main() {
    var i I

    var t *T
    i = t
    describe(i)
    i.M()

    i = &T{"hello"}
    describe(i)
    i.M()
}

func describe(i I) {
    fmt.Printf("(%v, %T)\n", i, i)
}
```

```
(<nil>, *main.T)
<nil>
(&{hello}, *main.T)
hello
```

Nil Interface Values

Calling a method on a nil interface is a run-time error.

```go
package main

import "fmt"

type I interface {
    M()
}

func main() {
    var i I
    describe(i)
    i.M()
}

func describe(i I) {
    fmt.Printf("(%v, %T)\n", i, i)
}
```

```
(<nil>, <nil>)
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x0 pc=0x498fcf]

goroutine 1 [running]:
main.main()
        /tmp/sandbox856238245/prog.go:12 +0x8f
```

The empty interface

Empty interfaces are used by code than handles values of unknown type. For example, `fmt.Print` takes amuy mumber of arguments of type `interface()` .

```go
package main
```

```
import "fmt"

func main() {
    var i interface {}
    describe(i)

    i = 42
    describe(i)

    i = "hello"
    describe(i)
}

func describe(i interface{}) {
    fmt.Printf("{(%v, %T)}", i, i)
}
```

```
(<nil>, <nil>)
(42, int)
(hello, string)
```

Type assertions

```
package main

import "fmt"

func main() {
    var i interface{} = "hello"

    s := i.(string)
```

```
    fmt.Print(s)

    s, ok := i.(float64)
    fmt.Println(f, ok)

    f = i.(float64) // panic
    fmt.Println(f)
}
```

```
hello
hello true
0 false
panic: interface conversion: interface {} is string, not float64

goroutine 1 [running]:
main.main()
        /tmp/sandbox322485079/prog.go:17 +0x1fe
```

Type switches

Tests whether the interface value `i` holds a value of type `T` or `S`.

```
switch v := i.(type) {
case T:
    // here v has type T
case S:
    // here v has type S
default:
    // mo match; here v has the same type as i
}
```

Stringers

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

example,

```
package main

import "fmt"

type Person struct {
    Name string
    Age  int
}

func (p Person) String() string {
    return fmt.Sprintf("%v (%v years)", p.Name, p.Age)
}

func main() {
    a := Person{"Arthur Dent", 42}
    z := Person{"Zaphod Beeblebrox", 9001}
    fmt.Println(a, z)
}
```

```
Arthur Dent (42 years) Zaphod Beeblebrox (9001 years)
```

Exercise: Stringers

```
package main

import "fmt"
```

```
type IPAddr [4]byte

// TODO: Add a "String() string" method to IPAddr.

func (ip IPAddr) String() string {
    return fmt.Sprintf("%v.%v.%v.%v", ip[0], ip[1], ip[2], ip[3])
}

func main() {
    hosts := map[string]IPAddr{
        "loopback":  {127, 0, 0, 1},
        "googleDNS": {8, 8, 8, 8},
    }
    for name, ip := range hosts {
        fmt.Printf("%v: %v\n", name, ip)
    }
}
```

Errors

```
package main

import (
    "fmt"
    "time"
)

type MyError struct {
    When time.Time
    What string
}

func (e *MyError) Error() string {
    return fmt.Sprintf("at %v, %s",
```

```
        e.When, e.What)
}

func run() error {
    return &MyError{
        time.Now(),
        "it didn't work",
    }
}

func main() {
    if err := run(); err != nil {
        fmt.Println(err)
    }
}
```

Exercise: Errors

```
package main

import (
    "fmt"
)

type ErrNegativeSqrt float64

func (e ErrNegativeSqrt) Error() string {
    return fmt.Sprintf("cannot Sqrt negative number: %v", float64(e))
}

func Sqrt(x float64) (float64, error) {
    if x < 0 {
        return 0, ErrNegativeSqrt(x)
    }
```

```
    z := 1.0

    for i := 0; i != 10; i++ {
        z -= (z*z - x) / (2 * z)
    }

    return z, nil
}

func main() {
    fmt.Println(Sqrt(2))

    fmt.Println(Sqrt(-2))
}
```

```
1.414213562373095 <nil>
0 cannot Sqrt negative number: -2
```

Readers

```
package main

import (
    "fmt"
    "io"
    "strings"
)

func main() {
    r := strings.NewReader("Hello, Reader!")

    b := make([]byte, 8)
    for {
```

```
        n, err := r.Read(b)
        fmt.Printf("n = %v err = %v b = %v\n", n, err, b)
        fmt.Printf("b[:n] = %q\n", b[:n])
        if err == io.EOF {
            break
        }
    }
}
```

```
n = 8 err = <nil> b = [72 101 108 108 111 44 32 82]
b[:n] = "Hello, R"
n = 6 err = <nil> b = [101 97 100 101 114 33 32 82]
b[:n] = "eader!"
n = 0 err = EOF b = [101 97 100 101 114 33 32 82]
b[:n] = ""
```

Exercise: Readers

```
package main

import "golang.org/x/tour/reader"

type MyReader struct{}

// TODO: Add a Read([]byte) (int, error) method to MyReader.
func (e MyReader) Read(bytes []byte) (int, error) {
    for i := 0; i < len(bytes); i++ {
        bytes[i] = 'A'
    }
    return len(bytes), nil
}


func main() {
    reader.Validate(MyReader{})
```

```
}
```

```
OK!


Program exited.
```

Exercise: rot13Reader

```
package main

import (
    "io"
    "os"
    "strings"
)

type rot13Reader struct {
    r io.Reader
}

func (r rot13Reader) Read(b []byte) (int, error) {
    l, err := r.r.Read(b)

    if err != nil {
        return l, err
    }

    for i := 0; i != l; i++ {
        if b[i] <= 'z' && b[i] >= 'a' {
            b[i] = (b[i] - 'a' + 13) % 26 + 'a'
```

```
        }

        if b[i] <= 'Z' && b[i] >= 'A' {
            b[i] = (b[i] - 'A' + 13) % 26 + 'A'
        }
    }

    return l, nil
}

func main() {
    s := strings.NewReader("Lbh penpxrq gur pbqr!")
    r := rot13Reader{s}
    io.Copy(os.Stdout, &r)
}
```

```
You cracked the code!
Program exited.
```

Images

```
package main

import (
    "fmt"
    "image"
)

func main() {
    m := image.NewRGBA(image.Rect(0, 0, 100, 100))
    fmt.Println(m.Bounds())
    fmt.Println(m.At(0, 0).RGBA())
```

```
}
```

```
(0,0)-(100,100)
0 0 0 0
```

Exercise: Images

```
package main

import (
    "image"
    "image/color"
    "golang.org/x/tour/pic"
)

type Image struct{
    w int
    h int
}


func (img Image) ColorModel() color.Model {
    return color.RGBAModel
}

func (img Image) Bounds() image.Rectangle {
    return image.Rect(0, 0, img.w, img.h)
}
```

```
func (img Image) At(x, y int) color.Color {
    return color.RGBA{uint8(x*y), uint8((x+y)/2), uint8(x^y), 255}
}


func main() {
    m := &Image{300, 300}
    pic.ShowImage(m)
}
```