

## 210515 A Tour of Go - Methods and interfaces & Concurrency

Notebook: GoLang

Created: 3/8/2021 4:06 PM

Updated: 5/19/2021 1:28 AM

Author: bill0412

---

- [Methods](#)
  - [Value Receivers](#)
  - [Pointer Recivers](#)
  - [Pointers and functions](#)
  - [Methods and pointer indirection](#)
- [Interfaces](#)
  - [Interfaces are implemented implicitly](#)
  - [Interface Values](#)
  - [Nil Interface Values](#)
  - [The empty interface](#)
  - [Type assertions](#)
  - [Type switches](#)
  - [Stringers](#)
    - [Exercise: Stringers](#)
    - [Errors](#)
    - [Exercise: Errors](#)
    - [Readers](#)
    - [Exercise: Readers](#)
    - [Exercise: rot13Reader](#)
  - [Images](#)
  - [Exercise: Images](#)
- [Concurrency](#)
  - [Goroutines](#)
  - [Channels](#)
    - [Buffered Channels](#)
  - [Range and Close](#)
  - [Select](#)
  - [Default Selection](#)
  - [Exercise: Equivalent Binary Trees](#)
  - [sync.Mutex](#)

# 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 struct {
    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 it 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.

```

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

```

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

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 represented 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 long 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.

```
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 that handles values of unknown type. For example, `fmt.Print` takes any number of arguments of type `interface{}`.

```
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:
    // no 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
}

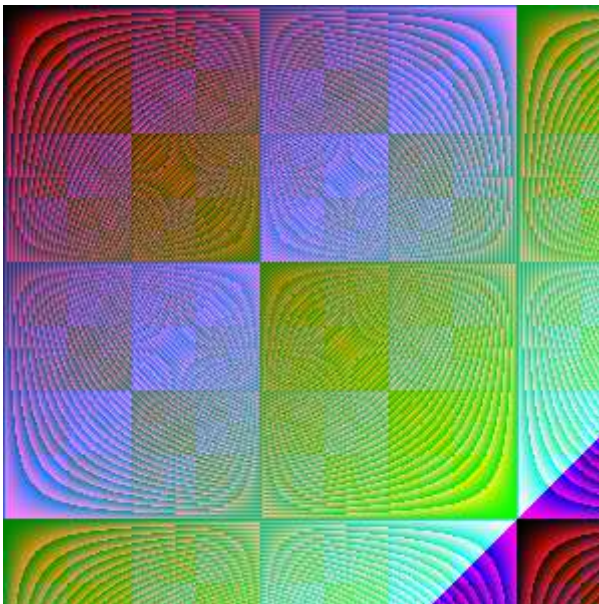
func (img Image) ColorModel() color.Model {
    return color.RGBAModel
}

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)
}

```



## Concurrency

---

# Goroutines

A goroutine is a lightweight thread managed by the Go runtime.

```
go f(x, y, z)
```

starts a new goroutine.

The evaluation of the function name and the arguments happen in the new goroutine while the execution of the function happens in the new goroutine.

`sync` package provides useful primitives for synchronization.

```
package main

import (
    "fmt"
    "time"
)

func say(s string) {
    for i := 0; i < 5; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(s)
    }
}

func main() {
    go say("world")
    say("hello")
}
```

# Channels

Channels are a typed conduit through which you can send and receive values with the channel operator `<-`.

```
ch <- v    // send v to channel ch
v := <-    // Receive from ch, and
           // assign value to v.
```

create a channel

```
ch := make(chan int)
```

sends and receives block until the other side is ready, it allows goroutine to synchronize without explicit locks or condition variables.

```
package main

import "fmt"

func sum(s []int, c chan int) {
    sum := 0
    for _, v := range s {
        sum += v
    }
    c <- sum // send sum to c
}

func main() {
    s := []int{7, 2, 8, -9, 4, 0}

    c := make(chan int)
    go sum(s[:len(s)/2], c)
    go sum(s[len(s)/2:], c)
    x, y := <-c, <-c // receive from c

    fmt.Println(x, y, x+y)
}
```

```
-5 17 12
```

## Buffered Channels

```
ch := make(chan int, 100)
```

example

```
package main

import "fmt"
```

```
func main() {
    ch := make(chan int, 2)
    ch <- 1
    ch <- 2
    ch <- 3
    fmt.Println(<-ch)
    fmt.Println(<-ch)
    fmt.Println(<-ch)
}
```

fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:

main.main()

/tmp/sandbox396107363/prog.go:9 +0x9f

Program exited: status 2.

## Range and Close

Only the sender should close a channel, never the receiver. Sending on a closed channel will cause a panic.

`for i := range c` receives values from the channel repeated until it is closed.

```
package main

import (
    "fmt"
)

func fibonacci(n int, c chan int) {
    x, y := 0, 1
    for i := 0; i < n; ++i {
        c <- x
        x, y = y, x + y
    }
    close(c)
}

func main() {
    c := make(chan int, 10)
    go fibonacci(cap(c), c)
    for i := range c {
```

```
        fmt.Println(i)
    }
}
```

## Select

```
package main

import "fmt"

func fibonacci(c, quit chan int) {
    x, y := 0, 1
    for {
        select {
        case c <- x:
            x, y = y, x + y
        case <- quit:
            fmt.Println("quit")
            return
        }
    }
}

func main() {
    c := make(chan int)
    quit := make(chan int)
    go func() {
        for i := 0; i < 10; i++ {
            fmt.Println(<-c)
        }
    }()
    fibonacci(c, quit)
}
```

## Default Selection

```
select {
case i := <- c:
    // use i
default:
```

```
    // receiving from c world block
}
```

```
package main

import (
    "fmt"
    "time"
)

func main() {
    tick := time.Tick(100 * time.Millisecond)
    boom := time.After(500 * time.Millisecond)
    for {
        select {
        case <-tick:
            fmt.Println("tick.")
        case <-boom:
            fmt.Println("BOOM!")
            return
        default:
            fmt.Println("    .")
            time.Sleep(50 * time.Millisecond)
        }
    }
}
```

## Exercise: Equivalent Binary Trees

```
package main

import "golang.org/x/tour/tree"
import "fmt"

// Walk walks the tree t sending all values
// from the tree to the channel ch.
func Walk(t *tree.Tree, ch chan int) {
    if t == nil {
        return
    }
}
```



```

        Walk(t.Left, ch)
        ch <- t.Value
        Walk(t.Right, ch)
    }

    // Same determines whether the trees
    // t1 and t2 contain the same values.
    func Same(t1, t2 *tree.Tree) bool {
        ch1 := make(chan int)
        ch2 := make(chan int)
        go Walk(t1, ch1)
        go Walk(t2, ch2)

        for i := 1; i <= 10; i++ {
            if <-ch1 != <-ch2 {
                return false
            }
        }

        return true
    }

    func main() {
        fmt.Println(Same(tree.New(1), tree.New(1)))
        fmt.Println(Same(tree.New(1), tree.New(2)))
    }

```

## sync.Mutex

```

package main

import (
    "fmt"
    "sync"
    "time"
)

// SafeCounter is safe to use concurrently.
type SafeCounter struct {
    mu sync.Mutex
    v  map[string]int
}

```

```

// Inc increments the counter for the given key.
func (c *SafeCounter) Inc(key string) {
    c.mu.Lock()
    // Lock so only one goroutine at a time can access the
map c.v.
    c.v[key]++
    c.mu.Unlock()
}

// Value returns the current value of the counter for the given
key.
func (c *SafeCounter) Value(key string) int {
    c.mu.Lock()
    // Lock so only one goroutine at a time can access the
map c.v.
    defer c.mu.Unlock()
    return c.v[key]
}

func main() {
    c := SafeCounter{v: make(map[string]int)}
    for i := 0; i < 1000; i++ {
        go c.Inc("somekey")
    }

    time.Sleep(time.Second)
    fmt.Println(c.Value("somekey"))
}

```

The output should be 1000 exactly