Seminario de Lenguajes opción Go

Raúl Champredonde

Seminario de Lenguajes opción Go

- Pointers
- Declaración de Tipos
- Methods
- Structs
- Embedded Types
- Interfaces

Pointers

- Un puntero es la dirección de memoria de un contenido de cierto tipo.
- `*T` es un puntero a un valor de tipo `T`
- Zero value es `nil`

```
var p *int
// fmt.Println(*p) // error en ejecución
if p != nil {
   fmt.Println(*p)
} else {
    fmt.Println("nil") // nil
}
i := 42
p = &i
fmt.Println(*p) // 42
```

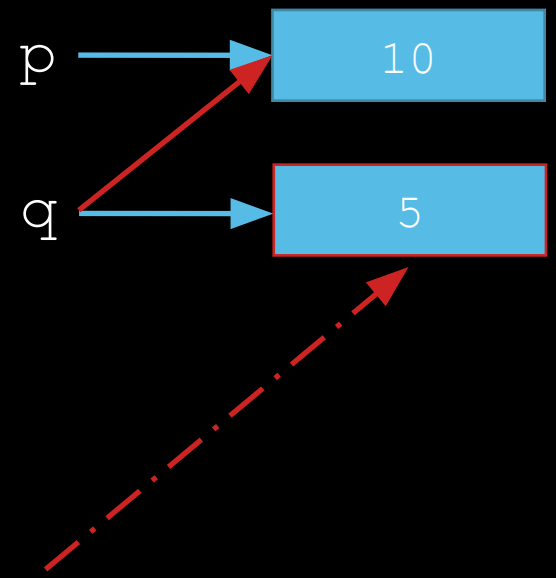
Pointers

- Alocador `new(T)`

```
p := new(int)
q := new(int)

*p = 10
*q = 5
fmt.Println(*p, *q) // 10 5

q = p
fmt.Println(*p, *q) // 10 10
```

Garbage collector

```
p := new(int)
q := new(int)

*p = 10
*q = 5
fmt.Println(*p, *q)

q = p   // garbage collector
fmt.Println(*p, *q)
```



Pointer parameters

- Las parámetros a funciones son “por valor”.
  Incluso los punteros, pero no sus contenidos:

```
func zero(xPtr *int) {
   *xPtr = 0
}

func main() {
   x := 5
   zero(&x)
   fmt.Println(x) // 0
}
```

```
func zero(xPtr *int) {
   *xPtr = 0
}

func main() {
   xPtr := new(int)
   zero(xPtr)
   fmt.Println(*xPtr) // 0
}
```

Declaración de Tipos

```
type MyInt int

type MyFloat float64
type MyFloat2 float64

type MyString string
type MyPString *string

type MyArray [5]int
type MySlice []int
```

- Tipo “nombrado”
- Igual al tipo subyacente, con los mismo valores y operaciones, pero incompatible

```
var f float64 = 1.5
var mf MyFloat = 2.5
var mf2 MyFloat2 = 3.5

f = mf // type mismatch
mf = f // type mismatch
// tampoco f > mf ni f == mf
```

Casting

```
type MyString *string
type MyString2 *string

func main() {
   var s string = "PE"
   var ms MyString = new(string)
   var ms2 MyString2 = new(string)

   *ms = s
   ms2 = ms // Error: incompatibles
   ms2 = MyString2(ms)

   fmt.Println(s + *ms)  // PEPE
   fmt.Println(s + *ms2) // PEPE
}
```

```
package tempconv

type Celsius float64
type Fahrenheit float64

func CToF(c Celsius) Fahrenheit {
   return Fahrenheit(c * 9 / 5 + 32)
}

func FToC(f Fahrenheit) Celsius {
   return Celsius((f - 32) * 5 / 9)
}
```

Casting

- `T(x)`
- Una conversión es permitida si ambos tipos (tipo origen y tipo destino) tienen el mismo tipo subyacente

Métodos

- Go (no tiene clases pero) permite definir métodos de tipos nombrados

```
type Celsius float64
type Fahrenheit float64
```

Función con un argumento "**receiver**"

```
func main() {
    var c Celsius = 35.0
    var f Fahrenheit = 95.0

    fmt.Println(c, f) // 35 95
}
```

```
func (c Celsius) String() string {
    return fmt.Sprintf("%g°C", c)
}

func (f Fahrenheit) String() string {
    return fmt.Sprintf("%g°F", f)
}

func main() {
    var c Celsius = 35.0
    var f Fahrenheit = 95.0
    fmt.Println(c.String(), f.String()) // 35°C 95°F
    fmt.Println(c, f)                   // 35°C 95°F
}
```

Métodos

```
type Celsius float64
type Fahrenheit float64
```

```
func CToF(c Celsius) Fahrenheit {
   return Fahrenheit(c * 9 / 5 +
32)
}

func FToC(f Fahrenheit) Celsius {
   return Celsius((f - 32) * 5 / 9)
}
```

```
func (c Celsius) CToF() Fahrenheit {
   return Fahrenheit(c * 9 / 5 + 32)
}

func (f Fahrenheit) FToC() Celsius {
   return Celsius((f - 32) * 5 / 9)
}

func main() {
   var c Celsius = 35.0
   f := c.CToF()
   fmt.Println(c, f, f.FToC()) // 35 95 35
}
```

Métodos

- El "receiver" actúa como si fuera un parámetro por valor

```
func (c Celsius) CToF() Fahrenheit { ... }
```

... es equivalente a ...

```
func CToF(c Celsius) Fahrenheit { ... }
```

Entonces no se puede modificar el receiver ...

```
type MyFloat float64

func (f MyFloat) Scale(s float64) {
   f = f * MyFloat(s)
}

func main() {
   f := MyFloat(-math.Sqrt2)
   fmt.Println(f, f.Abs()) // -1.4142... 1.4142...
   f.Scale(2)
   fmt.Println(f, f.Abs()) // -1.4142... 1.4142...
}
```

Métodos

- Para poder modificar el receiver …

```
type MyFloat float64

func (f *MyFloat) Scale(s float64) {
   *f = *f * MyFloat(s)
}
```

```
var ptrf *MyFloat = MyFloat(math.Sqrt2)
fmt.Println(*ptrf)
ptrf.Scale(2)
fmt.Println(*ptrf)
```

```
f := MyFloat(math.Sqrt2)
fmt.Println(f)
ptrf := &f
ptrf.Scale(2)
fmt.Println(f)
```

```
f := MyFloat(math.Sqrt2)
fmt.Println(f) f.Scale(2)
fmt.Println(f)
```

```
// 1.4142135623730951
// 2.8284271247461903
```

Métodos

- El receiver puede ser `nil`

```
type MySlice []int

func (mv *MySlice) Add() (res int) {
   res = 0
   if *mv == nil {
      return
   }
   for _, e := range *mv {
      res += e
   }
   return
}
```

```
func main() {
   ms1 := MySlice{1,2,3,4,5}
   var ms2 MySlice
   fmt.Println(ms1.Add()) // 15
   fmt.Println(ms2.Add()) // 0
}
```

Structs

- Conjunto de campos de distintos tipo:

```
type Person struct {
   firstname string
   lastname string
   age int
}

var p1 Person
p2 := Person{"Pepe", "Sargento", 25}
p3 := new(Person)
p4 := Person{lastname: "Larralde",
             firstname: "José"}

p1.firstname = "John"
p1.lastname = "Doe"

p3 = &p2
```

```
fmt.Println(p1)  // {John Doe 0}

fmt.Println(p2)  // {Pepe Sargento 25}
fmt.Println(p3)  // &{Pepe Sargento 25}
fmt.Println(*p3) // {Pepe Sargento 25}
fmt.Println(p4)  // {José Larralde 0}

p3.age = 28

fmt.Println(p2)  // {Pepe Sargento 28}
fmt.Println(p3)  // &{Pepe Sargento 28}
fmt.Println(*p3) // {Pepe Sargento 28}
```

Structs

- Conjunto de campos de distintos tipo:

```
type Date struct {
    day int
    month int
    year int
}
type Person struct {
    firstname string
    lastname string
    birthdate Date
}

var p1 Person
p2 := Person{"Pepe", "Sargento",
             Date{13, 4, 1968}}
p3 := new(Person)
p4 := Person{lastname: "Larralde",
             firstname: "José"}
```

```
p1.firstname = "John"
p1.lastname = "Doe"

p3 = &p2

fmt.Println(p1)  // {John Doe {0 0 0}}
fmt.Println(p2)  // {Pepe Sargento {13 4 1968}}
fmt.Println(p3)  // &{Pepe Sargento {13 4 1968}}
fmt.Println(*p3) // {Pepe Sargento {13 4 1968}}
fmt.Println(p4)  // {José Larralde {0 0 0}}

p3.birthdate.day = 28 // Notación punto también
                      // para punteros
p4.birthdate = Date{1, 1, 1910}

fmt.Println(p2)  // {Pepe Sargento {28 4 1968}}
fmt.Println(p3)  // &{Pepe Sargento {28 4 1968}}
fmt.Println(*p3) // {Pepe Sargento {28 4 1968}}
fmt.Println(p4)  // {José Larralde {1 1 1910}}
```

Structs

- Ejemplo:

```
type Circle struct {
   x, y, r float64 // Observar declaración
                   // con ","
}

func circleArea(c Circle) float64 {
   return math.Pi * c.r * c.r
}

func main() {
   c := Circle{0, 0, 5}
   fmt.Println(circleArea(c)) // 78.539…
}
```

Parámetros por copia …

```
type Circle struct {
   x, y, r float64
}

func circleArea(c Circle) float64 {
   c.r++
   fmt.Println(c.r)          // 6
   return math.Pi * c.r * c.r
}

func main() {
   c := Circle{0, 0, 5}
   fmt.Println(circleArea(c)) // 113.097…
   fmt.Println(c)             // {0, 0, 5}
}
```

Structs

- Los structs son comparables si todos sus campos lo son:

```
type Point struct{ X, Y int }
p := Point{1, 2}
q := Point{2, 1}
fmt.Println(p.X == q.X && p.Y == q.Y) // "false"
fmt.Println(p == q)                   // "false"
```

- Los tipos structs comparables se pueden usar como clave de maps

```
type address struct {
    hostname string
    port int
}
hits := make(map[address]int)

hits[address{"golang.org", 443}]++
```

Struct embedding

```
type Circle struct {
    X, Y, Radius int
}

type Cylinder struct {
    X, Y, Radius, Height int
}
```

```
type Point struct {
    X, Y int
}

type Circle struct {
    Center Point
    Radius int
}

type Cylinder struct {
    Circle Circle
    Height int
}
```

Struct embedding

```
type Circle struct {
   X, Y, Radius int
}

type Cylinder struct {
   X, Y, Radius, Height int
}
```

```
type Point struct {
   X, Y int
}

type Circle struct {
   Center Point
   Radius int
}

type Cylinder struct {
   Circle Circle
   Height int
}
```

```
var c Cylinder

c.Circle.Center.X = 1
                  // {{{1 0} 0} 0}

c.Circle.Center = Point{2, 3}
                  // {{{2 3} 0} 0}

c.Circle = Circle{Point{2, 3}, 10}
                  // {{{2 3} 10} 0}

d := Cylinder{???}
                  // {{{2 3} 10} 20}
```

Struct embedding

```
type Point struct {
   X, Y int
}

type Circle struct {
   Center Point
   Radius int
}

type Cylinder struct {
   Circle Circle
   Height int
}
```

```
type Point struct {
   X, Y int
}

type Circle struct {
   Point  // campo anónimo
   Radius int
}

type Cylinder struct {
   Circle // campo anónimo
   Height int
}
```

```
var ci Circle

ci.Point.X = 1          // {{1 0} 0}
ci.X = 2                // {{2 0} 0}



var cy Cylinder

cy.Circle.Point.X = 1 // {{{1 0} 0} 0}
cy.Circle.X = 2       // {{{2 0} 0} 0}
cy.X = 3              // {{{3 0} 0} 0}
```

Interfaces

- Un tipo `interface` está definido por un conjunto de firmas (encabezamiento) de métodos
- Un valor de un tipo `interface` puede ser cualquiera que implemente esos métodos
- Un tipo "implementa" un tipo `interface`, implementando sus métodos. No hay una declaración explícita de intención (`implements` keyword).

Interfaces

```
type Abser interface {
   Abs() float64
}
```

```
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)
}
```

```
var a Abser
f := MyFloat(123.45)
v := Vertex{3, 4}

a = f  // MyFloat implements Abser
fmt.Println(a.Abs())      // 123.45
a = &v // *Vertex implements Abser
fmt.Println(a.Abs())      // 5
```

```
var a Abser

a = MyFloat(123.45)
fmt.Println(a.Abs())      // 123.45
a = &Vertex{3, 4}
fmt.Println(a.Abs())      // 5
```

Interfaces

- El valor de un tipo interface se puede pensar como un par `(value, type)`
  donde `type` es un tipo concreto
  y `value`, un valor de ese tipo.
- La invocación de un método de la interface, ejecuta el método del mismo nombre del tipo subyacente.
- El valor del tipo concreto puede ser `nil`.
- Un valor de interface `nil` no tiene ni un tipo concreto ni un valor de dicho tipo.

Interfaces

```
type I interface {
   M()
}
```

```
type T struct {
   S string
}
func (t *T) M() {
   if t == nil {
      fmt.Println("nil!")
      return
   }
   fmt.Println(t.S)
}
```

```
type S string
func (s S) M() {
   fmt.Println(s)
}
```

```
type F float64
func (f F) M() {
   fmt.Println(f)
}
```

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

Interfaces

- Empty interface:
  - No especifica métodos
  - Puede mantener valores de cualquier tipo
  - Se puede escribir código que se aplique a cualquier tipo.
    Por ejemplo, `fmt.Print` tiene una cantidad indeterminada de parámetros `interface{}`.

```
func describe(i interface{}) {
    fmt.Printf("(%v, %T)\n", i, i)
}
type Point {
    X, Y: int
}
```

```
var i interface{}
describe(i) // (<nil>, <nil>)

i = 42
describe(i) // (42, int)

i = "hello"
describe(i) // (hello, string)

p := Point{1, 2}
describe(p) // ({1 2}, main.Point)
```

Interfaces

- Type assertion
  - Una type assertion `x.(T)` proporciona acceso al valor concreto subyacente de un valor de interfaz. Se aplica a un valor interface donde `x` es una expresión de un tipo interface y `T` es un tipo concreto (asserted type)

```
var i interface{} = "hello"

s := i.(string)
fmt.Println(s)

s, ok := i.(string)
fmt.Println(s, ok)

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

f = i.(float64) // runtime error
fmt.Println(f)
```

Interfaces

- Type switches

```
func do(i interface{}) {
   switch v := i.(type) {
   case nil:
      fmt.Printf("Nil: %v\n", v)
   case int:
      fmt.Printf("Twice %v is %v\n", v, v*2)
   case string:
      fmt.Printf("%q is %v bytes long\n", v, len(v))
   default:
      fmt.Printf("I don't know about type %T!\n", v)
   }
}
```

```
func main() {
   do(21)       // Twice 21 is 42
   do("hello") // "hello" is 5 bytes long
   do(true)    // I don't know about type bool!
   var p *int  // p es nil
   do(p)       // I don't know about type *int!
   var i interface{}
   do(i)       // Nil: <nil>
}
```

Interfaces

- Stringer interface

```
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)
}
```