بسم الله الرحمن الرحیم

تکنولوژی کامپیوتر

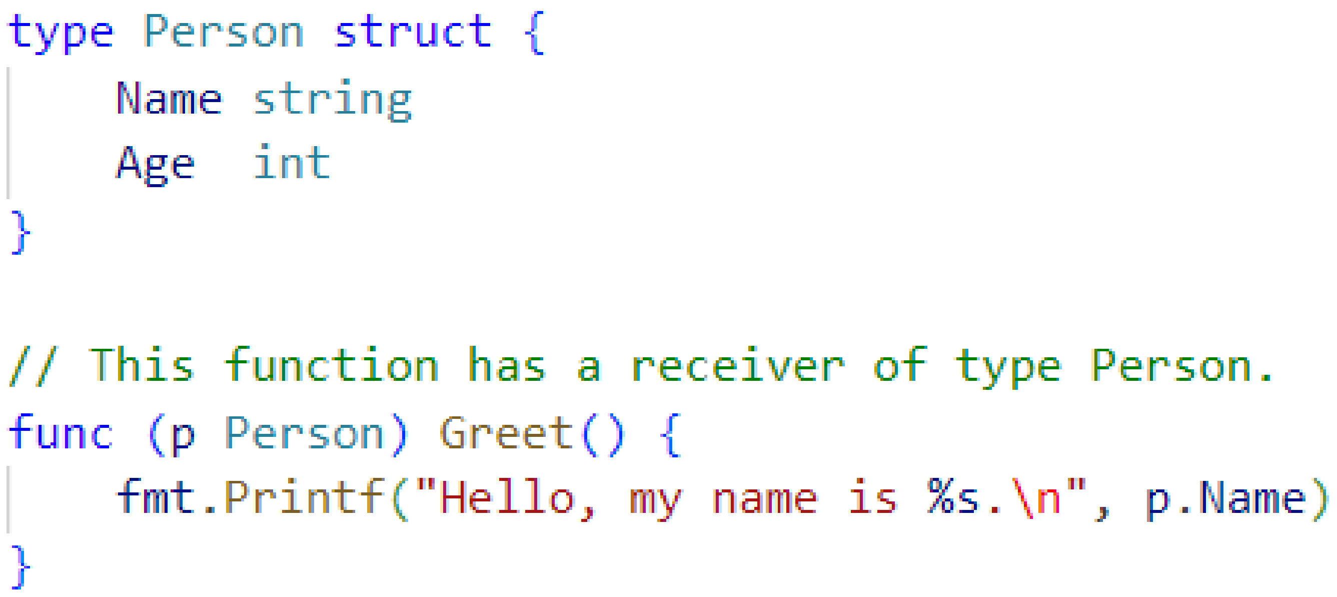
جلسه‌ی چهارم
زبان گولنگ (۴)

جلسه‌ی گذشته

گولنگِ بیشتر

حالا که کلاس نداریم، استراکت داریم، متود چطور؟؟

- function with receivers…

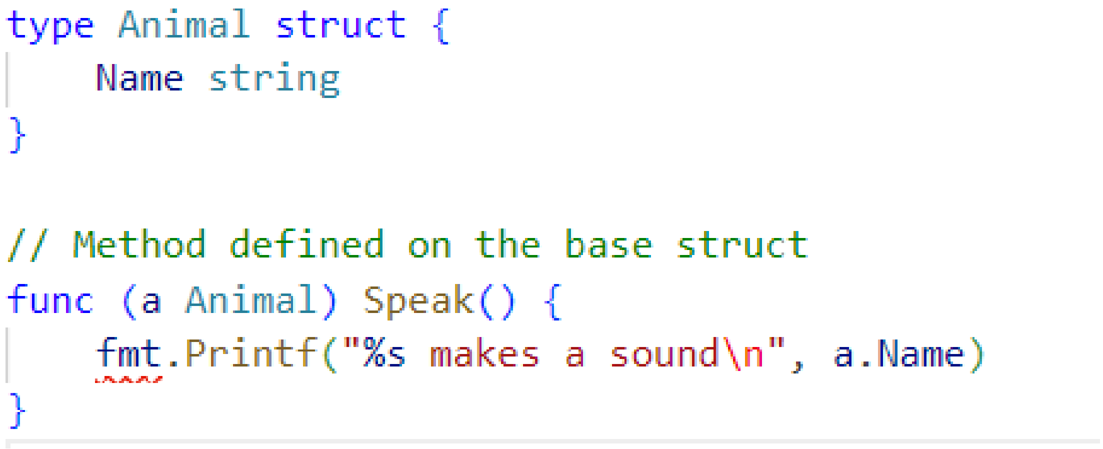


```go
type Person struct {
    Name string
    Age  int
}

// This function has a receiver of type Person.
func (p Person) Greet() {
    fmt.Printf("Hello, my name is %s.\n", p.Name)
}
```

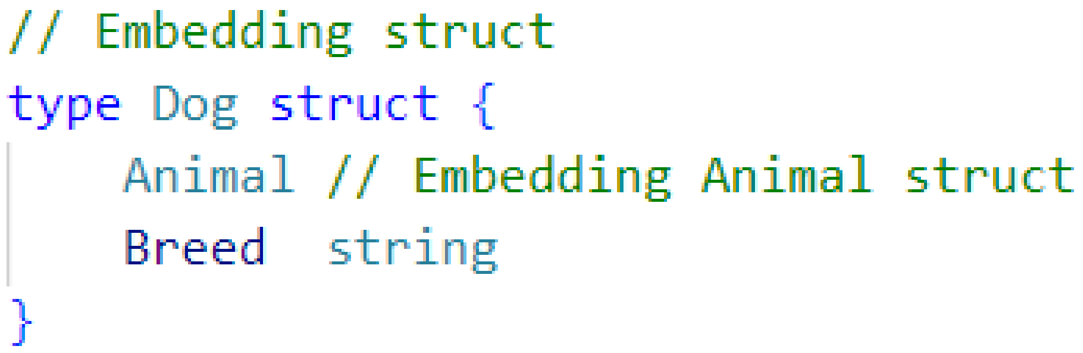
Choosing a value or pointer receiver

- There are two reasons to use a pointer receiver.
  - *The first is so that the method can modify the value that its receiver points to.*
  - *The second is to avoid copying the value on each method call. This can be more efficient if the receiver is a large struct, for example.*
- In general, all methods on a given type should have either value or pointer receivers, but not a mixture of both.

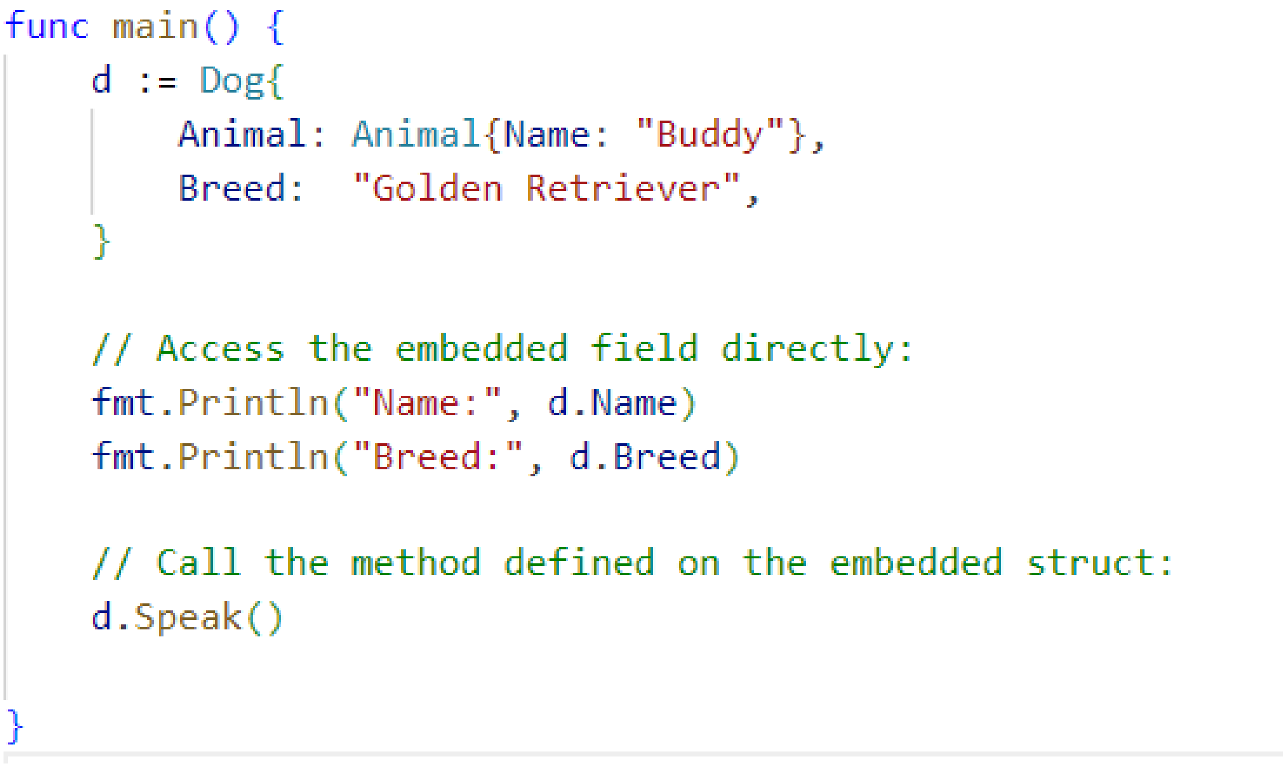
Struct embeding



```go
type Animal struct {
    Name string
}

// Method defined on the base struct
func (a Animal) Speak() {
    fmt.Printf("%s makes a sound\n", a.Name)
}
```

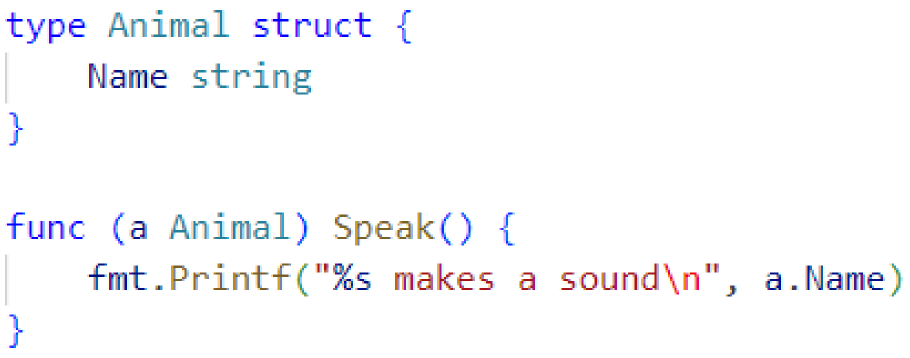


```go
// Embedding struct
type Dog struct {
    Animal // Embedding Animal struct
    Breed  string
}
```

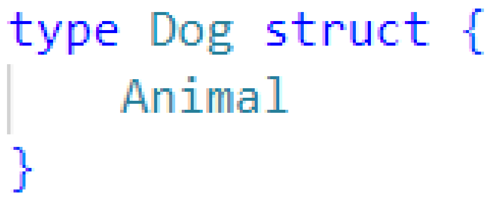


```go
func main() {
    d := Dog{
        Animal: Animal{Name: "Buddy"},
        Breed:  "Golden Retriever",
    }

    // Access the embedded field directly:
    fmt.Println("Name:", d.Name)
    fmt.Println("Breed:", d.Breed)

    // Call the method defined on the embedded struct:
    d.Speak()

}
```

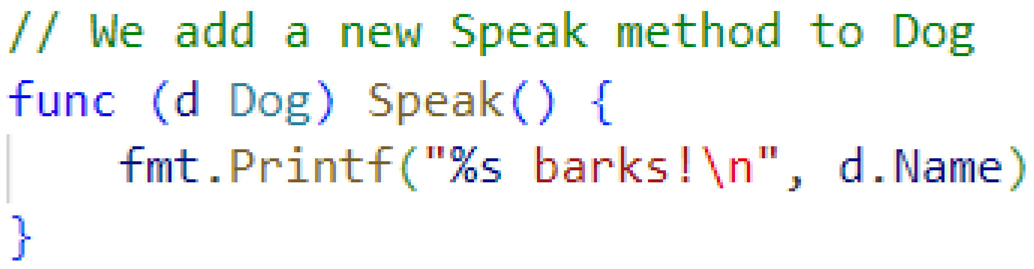
Struct embeding



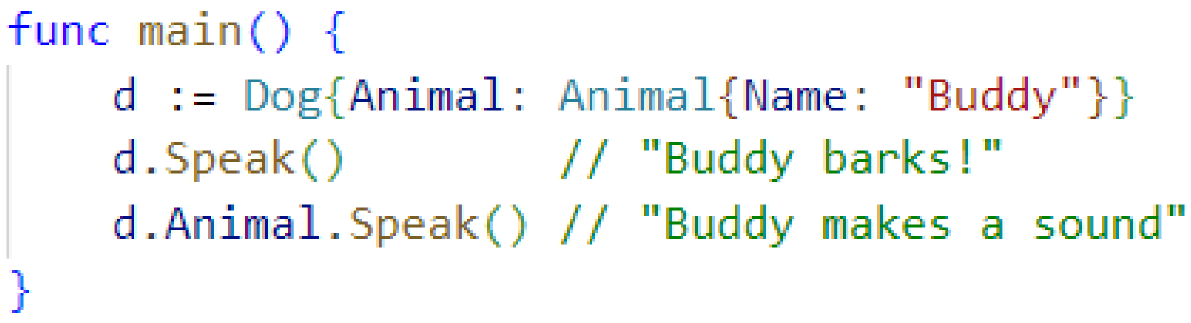
```go
type Animal struct {
    Name string
}

func (a Animal) Speak() {
    fmt.Printf("%s makes a sound\n", a.Name)
}
```



```go
type Dog struct {
    Animal
}
```



```go
// We add a new Speak method to Dog
func (d Dog) Speak() {
    fmt.Printf("%s barks!\n", d.Name)
}
```



```go
func main() {
    d := Dog{Animal: Animal{Name: "Buddy"}}
    d.Speak()          // "Buddy barks!"
    d.Animal.Speak() // "Buddy makes a sound"
}
```

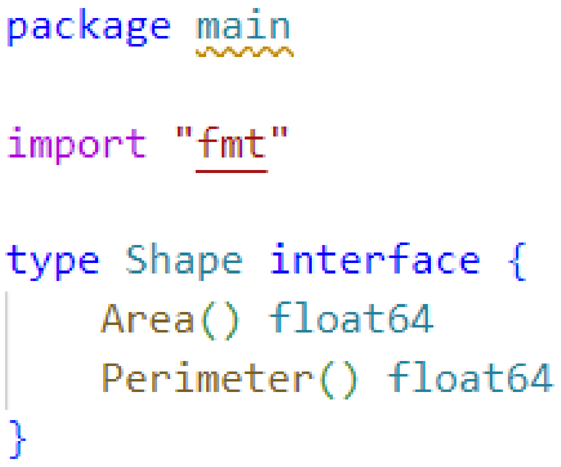
Interface?

- از جاوا، اینترفیس توش یه سری تعریف توابع میومد و هرکی ایمپلمنتش می‌کرد، باید اون توابع رو می‌داشت.
- فرایندی که در جاوا داشتیم:
  - *اینترفیس رو تعریف کنیم*
  - *توی کلاسمون بگیم که اون اینترفیس رو داریم پیاده می‌کنیم.*
  - *متودهای اینترفیس، عینا در کلاسمون بیاد.*

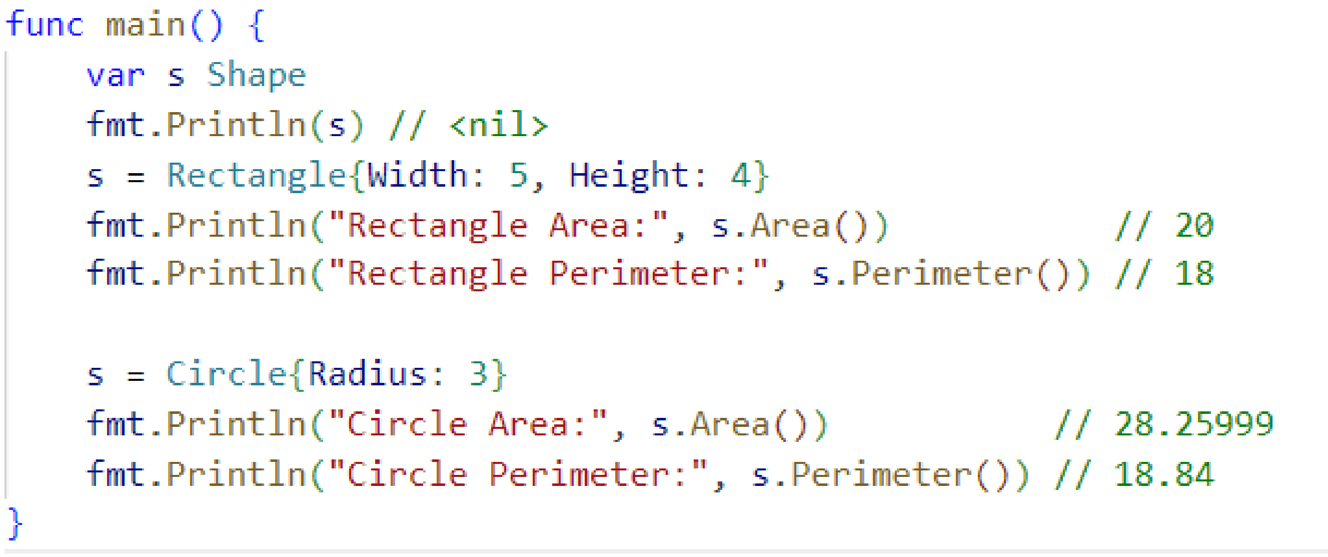
Interface

- در گولنگ
  - اینترفیس رو تعریف کنیم
  - ~~توی کلاسمون بگیم که اون اینترفیس رو داریم پیاده می‌کنیم.~~
  - متودهای اینترفیس، عینا در استراکتمون بیاد.

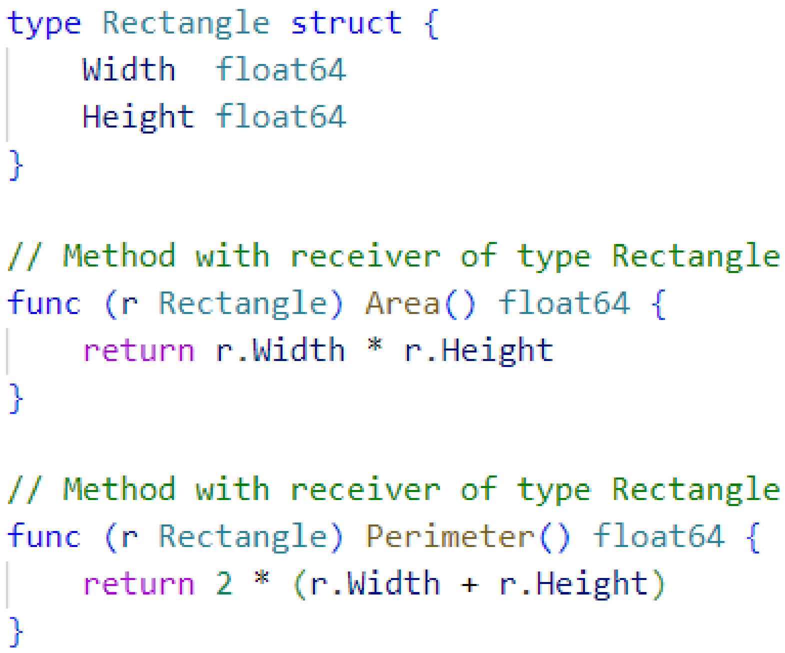
Interface



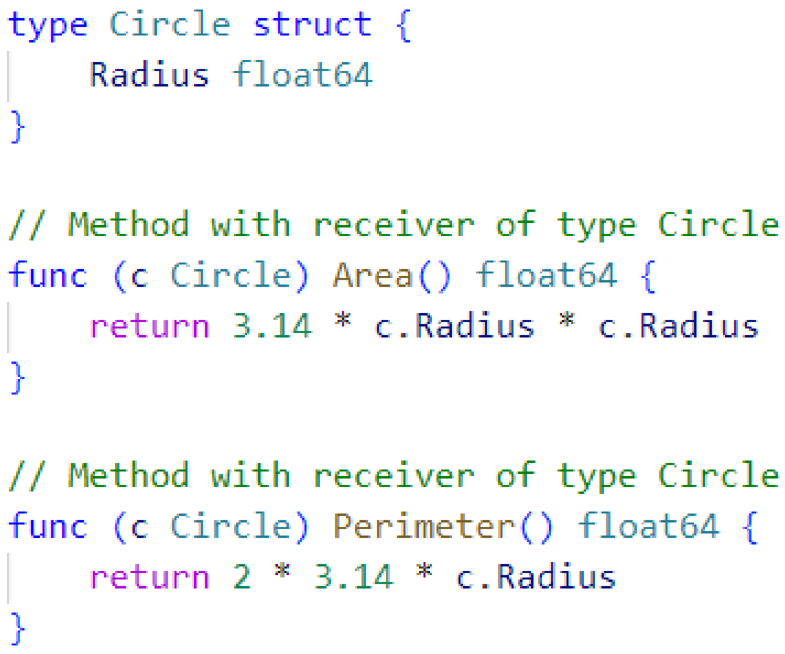
```go
package main

import "fmt"

type Shape interface {
    Area() float64
    Perimeter() float64
}
```



```go
func main() {
    var s Shape
    fmt.Println(s) // <nil>
    s = Rectangle{Width: 5, Height: 4}
    fmt.Println("Rectangle Area:", s.Area())          // 20
    fmt.Println("Rectangle Perimeter:", s.Perimeter()) // 18

    s = Circle{Radius: 3}
    fmt.Println("Circle Area:", s.Area())          // 28.25999
    fmt.Println("Circle Perimeter:", s.Perimeter()) // 18.84
}
```



```go
type Rectangle struct {
    Width  float64
    Height float64
}

// Method with receiver of type Rectangle
func (r Rectangle) Area() float64 {
    return r.Width * r.Height
}

// Method with receiver of type Rectangle
func (r Rectangle) Perimeter() float64 {
    return 2 * (r.Width + r.Height)
}
```



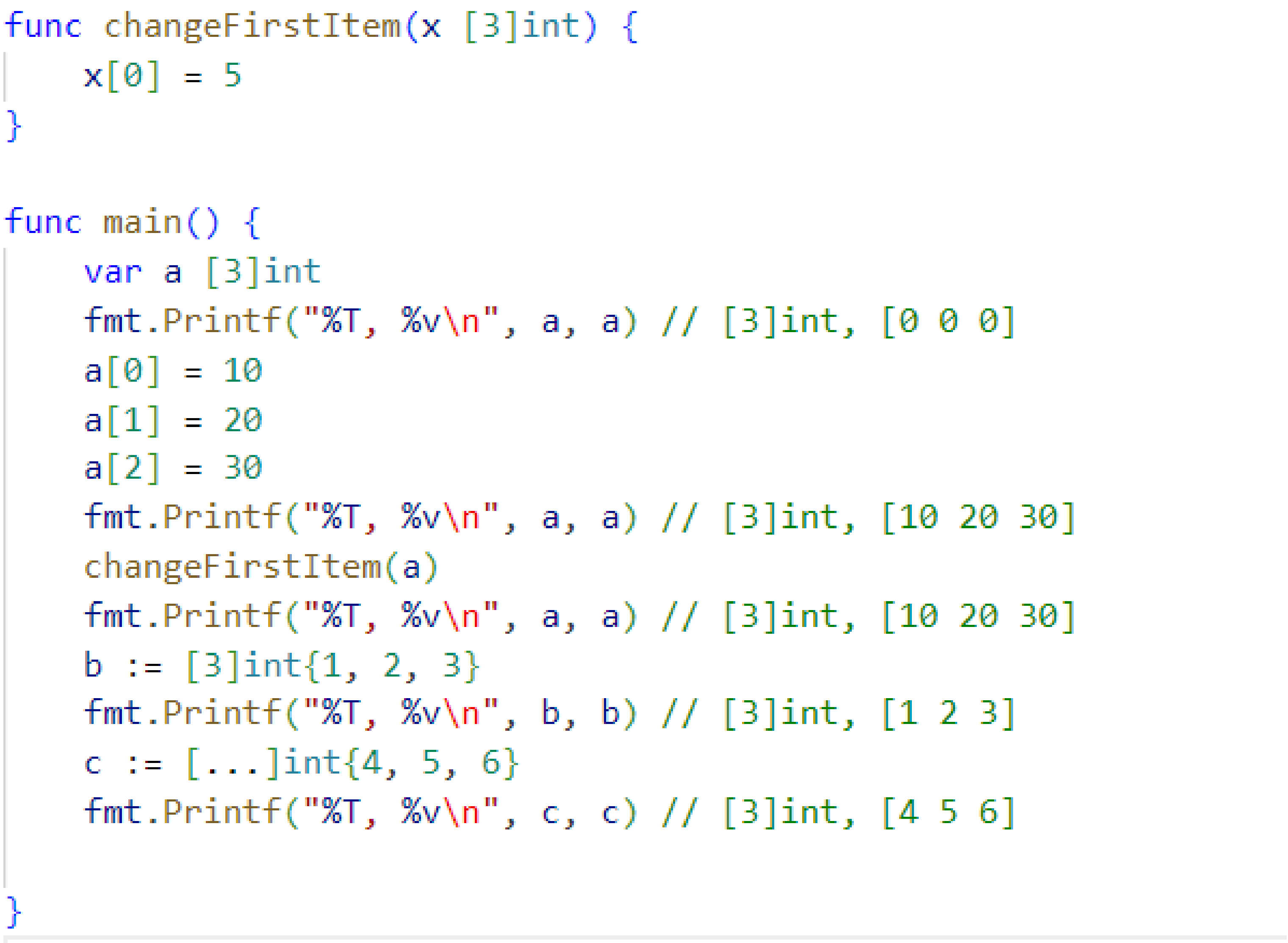
```go
type Circle struct {
    Radius float64
}

// Method with receiver of type Circle
func (c Circle) Area() float64 {
    return 3.14 * c.Radius * c.Radius
}

// Method with receiver of type Circle
func (c Circle) Perimeter() float64 {
    return 2 * 3.14 * c.Radius
}
```

Array

- Fixed Size
  - *An array has a fixed length, and the length is part of its type.*
    - For example, [3]int is an array of exactly 3 integers.
  - *The size cannot change.*
- Value Semantics
  - *When you assign an array to another array variable, it creates a copy of the entire array.*
  - *Similarly, when you pass an array to a function, it copies the array (unless you pass it by reference with a pointer).*

Array

- Declaration and Initialization



```
func changeFirstItem(x [3]int) {
    x[0] = 5
}

func main() {
    var a [3]int
    fmt.Printf("%T, %v\n", a, a) // [3]int, [0 0 0]
    a[0] = 10
    a[1] = 20
    a[2] = 30
    fmt.Printf("%T, %v\n", a, a) // [3]int, [10 20 30]
    changeFirstItem(a)
    fmt.Printf("%T, %v\n", a, a) // [3]int, [10 20 30]
    b := [3]int{1, 2, 3}
    fmt.Printf("%T, %v\n", b, b) // [3]int, [1 2 3]
    c := [...]int{4, 5, 6}
    fmt.Printf("%T, %v\n", c, c) // [3]int, [4 5 6]

}
```

Array

- Rarely Used Directly
  - *In practice, Go developers rarely use arrays for everyday programming because of the fixed size and copying behavior.*
  - *Instead, arrays are usually used internally to build more flexible data structures (like slices).*

Slice

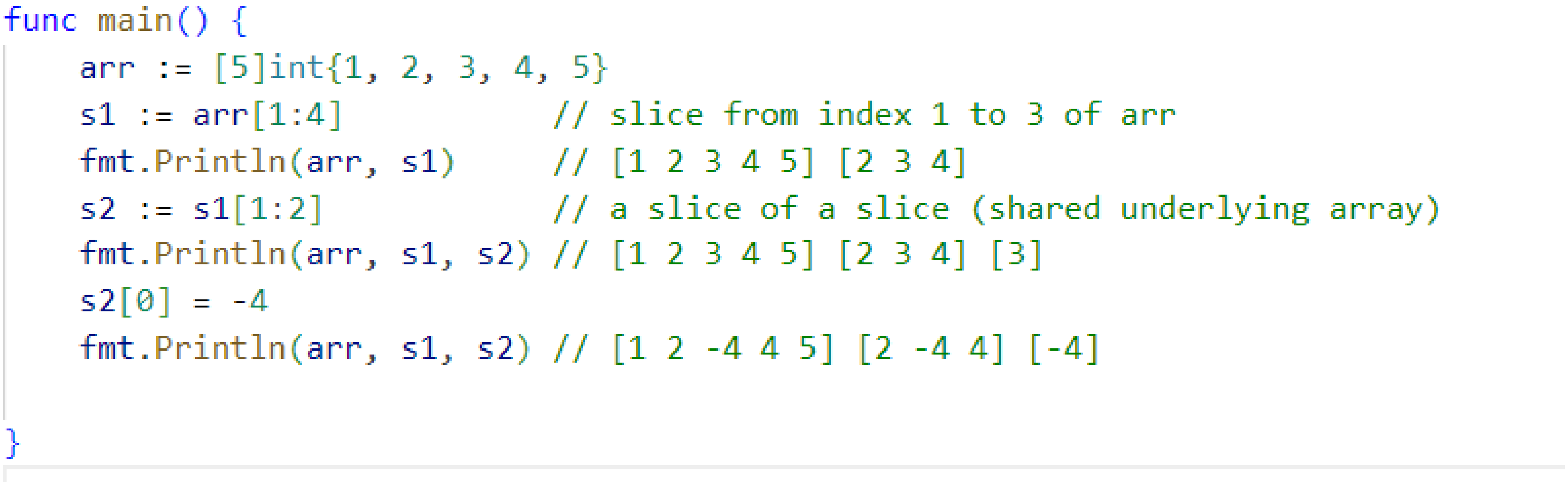
- Reference to an Underlying Array
  - *A slice does not store its own data. Instead, it references a portion of an underlying array.*
  - *Because of this, multiple slices can share the same underlying array.*
- Length and Capacity
  - *A slice has two critical properties:*
    - **Length:** the number of elements it contains.
    - **Capacity:** the total number of elements available in the underlying array from the first element of the slice to the end of that array.
  - *You can retrieve them using the built-in functions len(slice) and cap(slice).*

Slice

- Flexible and Resizable
  - *Slices are much more flexible than arrays because you can grow or shrink them using the built-in append function (as long as the capacity allows it).*
  - *If you exceed the capacity of the existing underlying array, Go will allocate a new array and copy the existing elements over.*

Slice

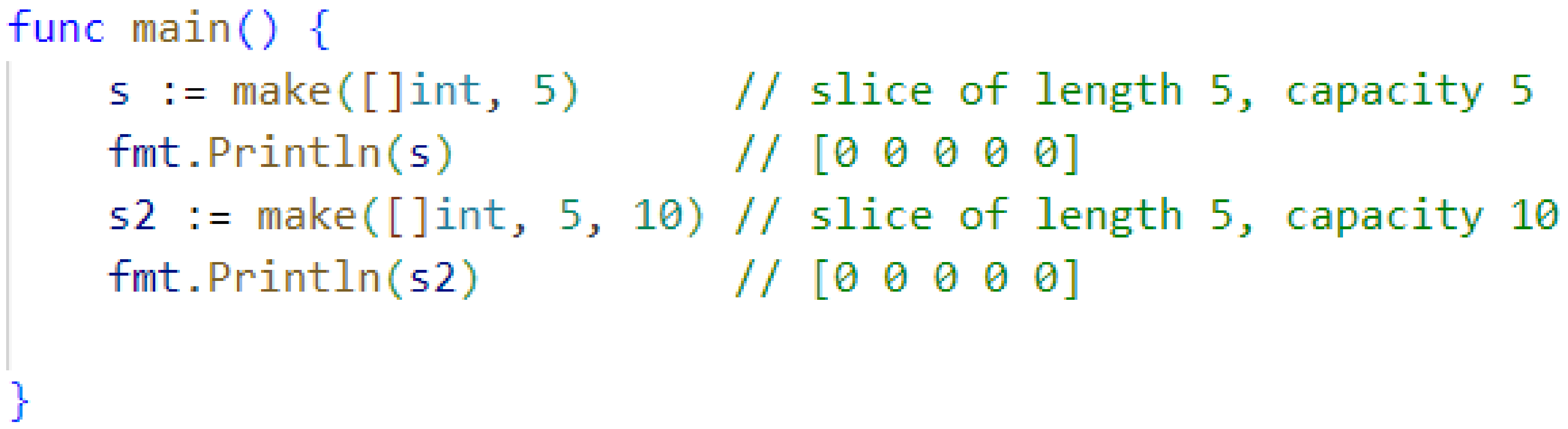
- Creating Slices
  - *From an array or an existing slice:*



```go
func main() {
    arr := [5]int{1, 2, 3, 4, 5}
    s1 := arr[1:4]           // slice from index 1 to 3 of arr
    fmt.Println(arr, s1)     // [1 2 3 4 5] [2 3 4]
    s2 := s1[1:2]            // a slice of a slice (shared underlying array)
    fmt.Println(arr, s1, s2) // [1 2 3 4 5] [2 3 4] [3]
    s2[0] = -4
    fmt.Println(arr, s1, s2) // [1 2 -4 4 5] [2 -4 4] [-4]

}
```

Slice

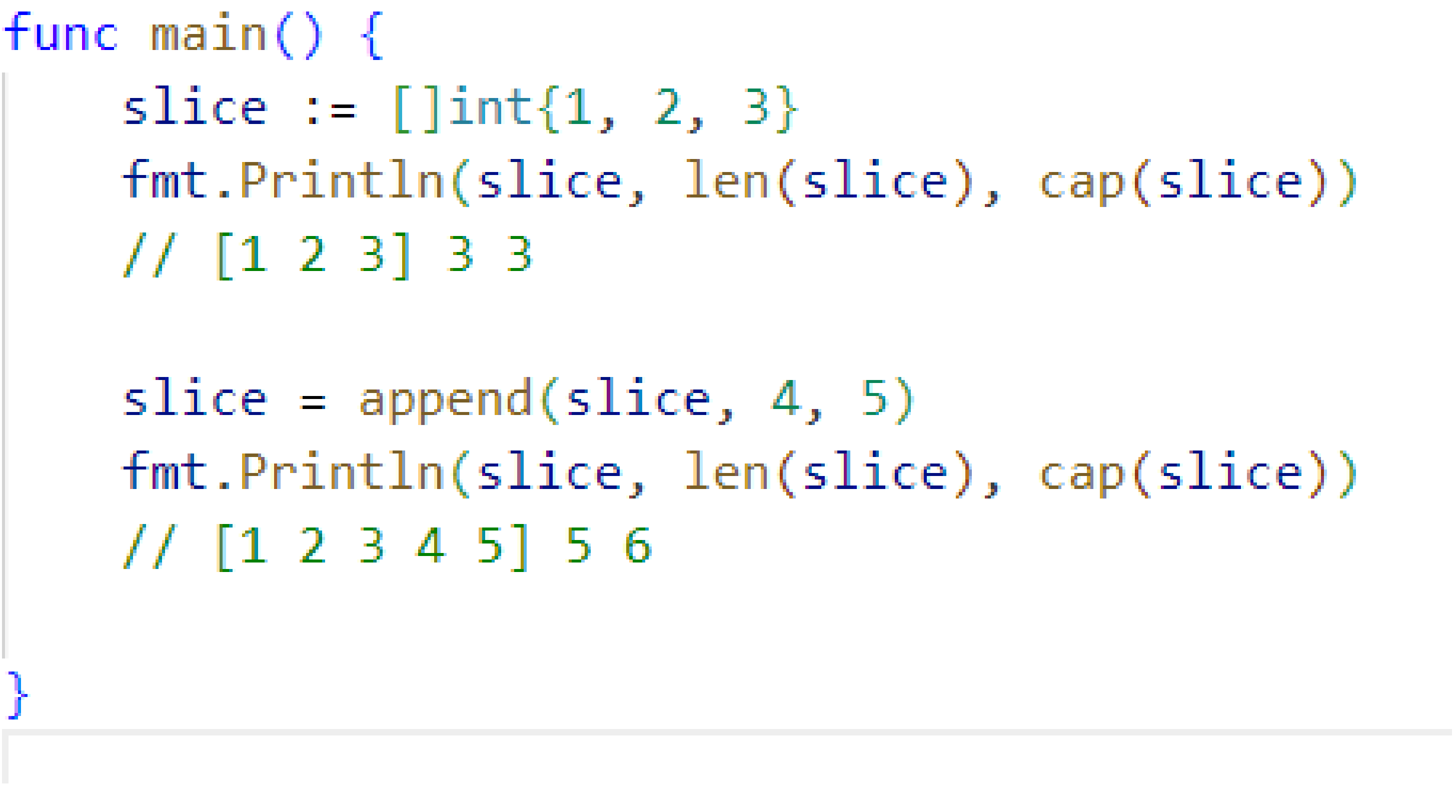
- Creating Slices
  - *Using make:*



```
func main() {
    s := make([]int, 5)      // slice of length 5, capacity 5
    fmt.Println(s)           // [0 0 0 0 0]
    s2 := make([]int, 5, 10) // slice of length 5, capacity 10
    fmt.Println(s2)          // [0 0 0 0 0]

}
```

Slice

- Appending to slice



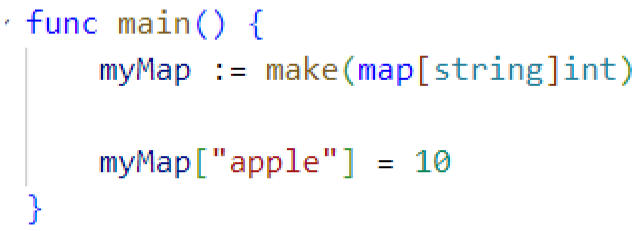
```go
func main() {
    slice := []int{1, 2, 3}
    fmt.Println(slice, len(slice), cap(slice))
    // [1 2 3] 3 3

    slice = append(slice, 4, 5)
    fmt.Println(slice, len(slice), cap(slice))
    // [1 2 3 4 5] 5 6

}
```

Slice

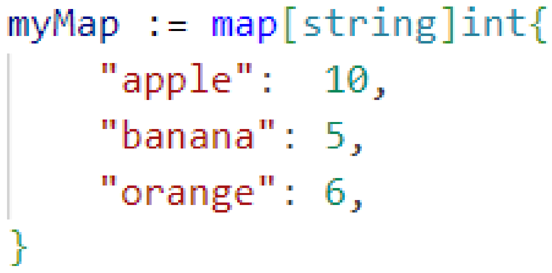
- s[l:r]
  - *0 <= l <= r <= cap(s)*
- s[l:] معادله با s[l:len(s)]
- s[:r] معادله با s[:r]

جلسەی جدید

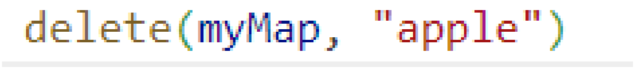
Map



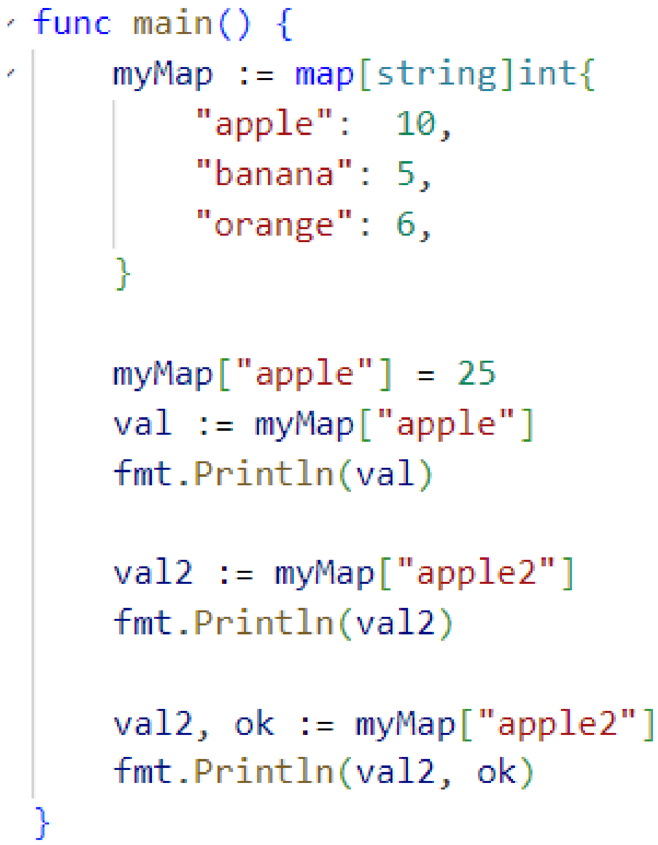
```go
func main() {
    myMap := make(map[string]int)

    myMap["apple"] = 10
}
```



```go
myMap := map[string]int{
    "apple":  10,
    "banana": 5,
    "orange": 6,
}
```

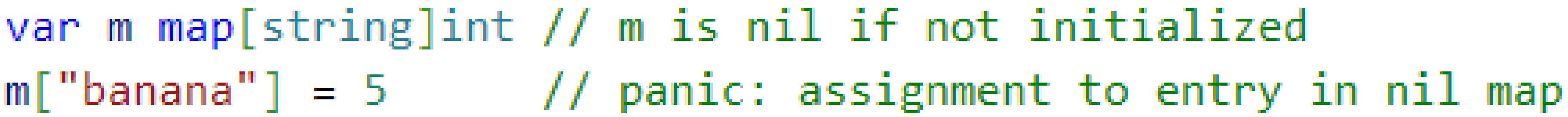


```go
delete(myMap, "apple")
```



```go
func main() {
    myMap := map[string]int{
        "apple":  10,
        "banana": 5,
        "orange": 6,
    }

    myMap["apple"] = 25
    val := myMap["apple"]
    fmt.Println(val)

    val2 := myMap["apple2"]
    fmt.Println(val2)

    val2, ok := myMap["apple2"]
    fmt.Println(val2, ok)
}
```

Map



```go
var m map[string]int // m is nil if not initialized
m["banana"] = 5      // panic: assignment to entry in nil map
```

Range

Closure

همزمانی

Goroutine

- ترد در جاوا
- چرا ترد سنگینه؟؟
- Asyncio در زبان‌هایی مثل جاوا اسکریپت
- Goroutine و lightweight thread...

Goroutine

- نمونه کد استفاده از goroutine

Goroutine

- ارتباط بین چند گوروتین با channelها.
- Sync.Mutex
- Sync.WaitGroup

ماژول در گولنگ

کمی چیزهای پیشرفته‌تر

Struct Tag

- با مثال json