ECE 473/573
Cloud Computing and Cloud Native Systems
Lecture 04 Functions and OOP in Go

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

August 28, 2024

Outline

More on Functions

Structs, Methods, and Interfaces

Reading Assignment

- This lecture: 2,3
- Next two lectures: Virtualization and Containerization.

Outline

More on Functions

Structs, Methods, and Interfaces

Error Handling

```
func SomeFunc() (int, error) {
  return 0, fmt.Errorf("error %d", 42)
}

func Error() {
  i, err := SomeFunc()
  // i := SomeFunc() // won’t compile
  // i, _ := SomeFunc() // also ok
  if err == nil {
    fmt.Printf("Got %d.\n", i)
  } else {
    fmt.Printf("Error %v.\n", err)
  }
}
```

- Go functions can return multiple results.
  - You are required to use all of them or cannot use any of them.
- This feature is leveraged for error handling in Go.
  - Errors are usually returned as the last result, and you cannot ignore them, unless using the blank identifier `_`.

Variadic Functions

```go
func Sum(a ...int) int {
  sum := 0
  for _, i := range a {
    sum += i
  }
  return sum
}

func Variadic() {
  fmt.Printf("sum(1,3,4)=%d\n", Sum(1, 3, 4))
  fmt.Printf("sum(1,2,3,4,5)=%d\n", Sum(1, 2, 3, 4, 5))
}
```

- Variadic functions allow to take any number of arguments.
  - Of the same type.
  - Must be the last ones in the argument list.
- In the function, the variadic argument is noted by `...` before its type, and is treated as a slice.

Anonymous Functions

```
func SortIndex(names []string) []int {
  indices := make([]int, 0)
  for i := range names {
    indices = append(indices, i)
  }
  sort.Slice(indices, func(l, r int) bool {
    lstr := names[indices[l]]
    rstr := names[indices[r]]
    return lstr < rstr
  })
  return indices
}

func Lambda() {
  names := []string{"Dave", "Bob", "Alice", "Clair"}
  for _, index := range SortIndex(names) {
    fmt.Printf("%s,", names[index])
  }
  fmt.Printf("names=%v\n", names)
}
```

- Functions can be created on the fly and refer to any variables.
  - As supported by most other languages nowadays except C.
  - They are anonymous since they don't have a name.

Defer

```
func Defer() {
  file, err := os.Create("foo.txt")
  if err != nil {
    log.Print(err)
    return
  }
  defer func() {
    file.Close()
    fmt.Println("File closed.")
  }() // the ending () actually calls the function

  for i := 0; i < 100; i++ {
    fmt.Fprintf(file, "%d\n", i)
  }
}
```

- `defer` allows a statement to be executed whenever the function returns.
  - Make it much easier to handle complex resource management logic with error handling (not available for C).
- Note the use of the extra `()` to call the anonymous function.

Outline

More on Functions

Structs, Methods, and Interfaces

Structs

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

func Struct() {
  v := Vertex{X: 1, Y: 2}
  fmt.Printf("%v, ", v)
  v.X, v.Y = 3, 4
  fmt.Printf("%+v\n", v)
}
```

- `struct` aggregates related variables together into an object
  - As a foundation feature to OOP languages like C++ and Java.
- Use `%v` to print values of members.
  - `%+v` prints member names in addition.

Methods

```
func (v *Vertex) Move(dx, dy float64) {
  v.X += dx
  v.Y += dy
}
func (v Vertex) Norm() float64 {
  return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
func Methods() {
  v := Vertex{X: 1, Y: 2}
  v.Move(1, 2)
  fmt.Printf("%+v, norm=%.3f\n", v, v.Norm())
}
```

- Methods are functions attached to types.
  - Via an extra receiver argument before the function name.
- Pointer receivers allow to modify the object.
  - Work as `this` for C++/Java.
  - Methods with poiter receivers behave the same as methods in other OOP languages.
- Value receivers apply to a copy of the object.
  - A very special feature of Go (and C).

Interface

```
type Movable interface {
  Move(dx, dy float64)
}
func MoveAll(dx, dy float64, movables []Movable) {
  for _, m := range movables {
    m.Move(dx, dy)
  }
}
func Interface() {
  ms := []Movable{}
  ms = append(ms, &Vertex{X: 1, Y: 2})
  MoveAll(10, 20, ms)
}
```

- `interface` specifies what methods should be provided for an object to implement it.
- Functions can access those objects via `interface` and only use the methods defined within.
  - No knowledge of the actual type, less couplings!
- `interface` usually works with pointer receivers so need to convert from a pointer to the object.

Duck Typing

```
type Circle struct {
  X, Y, R float64
}
func (c *Circle) Move(dx, dy float64) {
  c.X += dx
  c.Y += dy
}
func Interface() {
  ms := []Movable{}
  ms = append(ms, &Vertex{X: 1, Y: 2})
  ms = append(ms, &Circle{X: 3, Y: 4, R: 5})
  MoveAll(10, 20, ms)
}
```

- ”If it walks like a duck and it quacks like a duck, then it must be a duck”
- A type implements an interface by implementing all require methods in the interface.
  - With the exact name, arguments, and returned results.
  - No need to inherit or to mention the interface explicitly.

Stringer

```
func (c Circle) String() string {
  return fmt.Sprintf("Circle(%.3f,%.3f,r=%.3f)", c.X, c.Y, c.R)
}
func (v Vertex) String() string {
  return fmt.Sprintf("Vertex(%.3f,%.3f)", v.X, v.Y)
}
func Interface() {
  ms := []Movable{}
  ms = append(ms, &Vertex{X: 1, Y: 2})
  ms = append(ms, &Circle{X: 3, Y: 4, R: 5})
  MoveAll(10, 20, ms)
  fmt.Printf("%v\n", ms)
}
```

- `%v` works with the `Stringer` interface.
- A type can implement it by implementing the `String() string` method.

Struct Embedding

```go
type Circle2 struct {
  Vertex
  R float64
}
func (c *Circle2) Move(dx, dy float64) {
  c.Vertex.Move(dx, dy)
}
func (c Circle2) String() string {
  return fmt.Sprintf("Circle(%.3f,%.3f,r=%.3f)", c.X, c.Y, c.R)
}
func Embedding() {
  ms := []Movable{&Circle2{Vertex: Vertex{X: 3, Y: 4}, R: 5}}
  MoveAll(10, 20, ms)
  fmt.Printf("%v\n", ms)
}
```

- `struct` can have other `struct`s as members.
- You don't have to name them.
  - Refer to the anonymous member as a whole by its type.
  - Refer to members of the anonymous member directly.
- Very similar to how base classes work for C++ and Java.
  - Except when implementing a base interface in C++/Java.
  - But Go don't need that for implementing interfaces!

Summary

- Go provides anonymous functions and `defer` that are available for most other languages but not C.
- Go embraces modern OOP practices by separating composition (embedding) and interface-based design, instead of using inheritance for both.
- We will cover other language features like concurrency as the course goes when needed.