8.28 발표

Go는 Class가 없다.

Struct는 메서드도 구조체로부터 분리되는 구성을 가지고 있다.

→ 단일 상속도 없고 당연히 다중 상속도 없다.

하지만, Go도 충분히 객체지향적이라고 한다. 그 이유를 알아보자!

1. 클래스

Java에서는 하나의 클래스에 상태와 동작을 모두 표현하였는데,

Go는 상태를 표현하는 타입과 동작을 표현하는 메서드를 분리하여 정의한다고 한다!

대신 receiver가 구조체와 함수를 연결해준다고 한다.

```
type Rectangle struct {
    Name    string
    Width, Height float64
}

func (r Rectangle) Area() float64 {
    return r.Width * r.Height
}
```

이렇듯, Go는 구조체(struct) 내에 메서드를 포함할 수 없다.

메서드는 구조체 밖에 만들어지며, receiver를 이용해서 어느 구조체의 메서드인지 정의할 수 있다.

```
func (r Rectangle) Area() float64 {
    return r.Width * r.Height
}
```

특정 구조체에서만 함수를 사용하게 지정을 하였다.

→ 객체지향언어세서 멤버함수의 역할

Go언어에서 구조체의 메소드는 따로 접근 제한자 키워드가 없다.

접근제어자? : 패키지 밖에서 해당 패키지를 이용할 때의 접근 제한

- 변수 및 메소드가 소문자로 시작 : 패키지 내부에서만 사용 가능
- 변수 및 메서드가 대문자로 시작 : 패키지 내부/외부에서 사용 가능

```
type Rectangle struct {
    Name    string
    Width, Height float64
}

func (r Rectangle) Area() float64 {
    return r.Width * r.Height
}

func (r Rectangle)Width float64{
  return r.width
}

func (r Rectangle)Height float64{
  return r.height
}

func (r *Rectangle)setName(name string){
  return r.name=name
}
```

따라서 getter/setter도 만들 수 있다.

getter는 구조체의 값 변경이 되지 않도록 값을 복사해서 전달했고,

setter는 구조체의 값 변경을 할 수 있도록 포인터를 주었다.

인터페이스

interface를 이용해서 다형성을 지원함.

Java와 마찬가지로 interface 키워드를 이용해 정의한다.

```
type Rectangle interface {
    Area()
}
```

이렇게 추상화된 메서드의 집합을 만들 수 있다.

```
package main

import "fmt"
import "math"

//인터페이스
type Shaper interface {
    Area() int
}

type Rectangle struct {
    width, height int
}

type Triangle struct {
    width, height int
}

type Circle struct {
    radius float64
}

//인터페이스 구현
func (r Rectangle) Area() int {
    return r.width * r.height
}

func (r Triangle) Area() int {
    return (r.width * r.height) / 2
}

func (r Circle) Area() float64 {
    return (r.radius * r.radius) * math.Pi
}

func main() {
    r := Rectangle{3, 5}
    t := Triangle{3, 6}
    c := Circle{10.0}
    fmt.Println("Area of the Rectangle ", r.Area())
    fmt.Println("Area of the Triangle ", t.Area())
    fmt.Println("Area of the Circle", c.Area())
```

```
    s := Shaper(r)
    fmt.Println("Area of the Spape r is ", s.Area())
}
```

상속

Embedding은 객체지향언어에서 상속 관계의 역할에 해당한다.

특정 구조체가 다른 구조체를 상속한 것과 같은 효과를 낼 수 있다.

```
package main

import "fmt"

type Shape struct {
  name string
}

func (a Shape) introduce() {
  fmt.Println("도형명은 " + a.name + "입니다.")
}

type Trianlge struct {
  // Shape 구조체의 자료형만 선언하면 상속과 같은 역할
  Shape
}

type Circle struct {
  // Shape 구조체의 자료형만 선언하면 상속과 같은 역할
  Shape
}

func main() {
  shape := Shape{"도형"}
  triangle := Triangle{}
  circle := Circle{}

  triangle.Shape.name = "세모"
  circle.Shape.name = "동그라미"

  shape.introduce() //도형명은 도형입니다.
  triangle.introduce() //도형명은 세모입니다.
  circle.introduce() //도형명은 동그라미입니다.

  triangle.Shape.introduce() //도형명은 세모입니다.
  circle.Shape.introduce() //도형명은 동그라미입니다.
}
```

Triangle과 Circle 구조체 안에 Shape 자료형만 선언하면 객체 지향 언어에서의 상속 구조와 같아진다.

`circle.introduce()`와 `circle.Shape.introduce()`가 모두 같은 값을 나타내는 것을 알 수 있다.

다만, Shape**의 변수에 값을 지정할 때에는 반드시** `circle.Shape.name` **형식으로 값을 넣어야 한다.**

구조체 메서드 오버라이딩

```
package main

import "fmt"

type Shape struct {
  name string
}

func (a Shape) introduce() {
  fmt.Println("도형명은 " + a.name + "입니다.")
}

type Trianlge struct {
  // Shape 구조체의 자료형만 선언하면 상속과 같은 역할
  Shape
}

func (t Triangle) introduce() {
  fmt.Println("세모의 이름은 " + c.name + "입니다.")
}

type Circle struct {
  // Shape 구조체의 자료형만 선언하면 상속과 같은 역할
  Shape
}

func (c Circle) introduce() {
  fmt.Println("동그라미의 이름은 " + c.name + "입니다.")
}

func main() {
  shape := Shape{"도형"}
  triangle := Triangle{}
  circle := Circle{}

  triangle.Shape.name = "세모"
  circle.Shape.name = "동그라미"

  shape.introduce() //도형명은 도형입니다.
  triangle.introduce() //세모의 이름은 세모 입니다.
  circle.introduce() //동그라미의 이름은 동그라미 입니다.
```

```
  triangle.Shape.introduce() //도형명은 세모입니다.
  circle.Shape.introduce() //도형명은 동그라미입니다.
}
```

이때 **Embedding 한 구조체와 동일한 함수명을 가진 Receiver를 생성하면 overriding과 같은 기능을 구현할 수 있다.**

`circle.introduce()`와 `circle.Shape.introduce()`가 모두 다른 값을 나타내는 것을 알 수 있다.