INTRODUCCIÓN A LA CLASE

Go Bases

Objetivos de esta clase

Los objetivos de esta clase son:

- Conocer y aplicar las estructuras en Go.
- Comprender y utilizar qué son los métodos dentro de nuestras estructuras.
- Comprender y aplicar las etiquetas de las estructuras.
- Conocer y utilizar interfaces en Go.

¡Vamos por ello!

ESTRUCTURAS EN GO

Go Bases

// ¿Qué es una estructura?

Una estructura es una colección de campos de datos.

Por ejemplo, podemos definir una estructura “persona” y en ella tener valores como edad, peso, género, profesión, etc...

Definimos una estructura de la siguiente manera: determinamos sus campos seguido de un espacio y el tipo de dato.

Para separar cada campo utilizamos un salto de línea.

```
type Persona struct {
   Nombre    string
   Genero    string
   Edad      int
   Profesión string
   Peso      float64
}
```

Para instanciar una estructura, podemos utilizar distintas formas:

Indicar todos los valores que queremos que tengan los campos.

```
{}  p1 := Persona{"Celeste", 34, "Ingeniera", 65.5}
```

O definir los valores para el campo que corresponda. De esta manera podemos no asignar valores a todos los campos, y de ser así, los valores quedarán por defecto según el tipo de dato.

```
{}  p2 := Persona{
        Nombre:    "Nahuel",
        Edad:      30,
        Profesión: "Ingeniero",
        Peso:      77,
    }
```

Para acceder a un campo de la estructura procedemos de la siguiente manera:

```
p2.Peso
```

Para asignar o modificar un valor a una campo de la estructura procedemos de la siguiente manera:

```
p2.Peso = 70
```

También podemos definir una estructura vacía e ir asignando los valores.

```
{}
var p3 Persona
p3.Nombre = "Ulises"
p3.Edad = 15
```

Podemos utilizar las estructuras como un tipo de dato, por ende, podríamos tener estructuras como campos dentro de otra estructura.
Por ejemplo, podemos tener una estructura **Preferencias** dentro de nuestra estructura **persona**.
Para eso debemos declarar nuestra estructura.

```
{}
type Preferencias struct {
    Comidas   string
    Peliculas string
    Series    string
    Animes    string
    Deportes  string
}
```

Y asignaremos un campo **Gustos** a nuestra estructura **persona**.

```
type Persona struct {
   Nombre    string
   Edad      int
   Profesión string
   Peso      float64
   Gustos    Preferencias
}
```

Hacemos lo siguiente para instanciar nuestra estructura:

```
p1 := Persona{"Celeste", 34, "Ingeniera", 65.5, Preferencias{"pollo", "titanic", "", "", ""}}
```

Podemos instanciarla haciendo referencia a cada campo.

{}

```
p2 := Persona{
    Nombre:    "Nahuel",
    Edad:      30,
    Profesión: "Ingeniero",
    Peso:      77,
    Gustos: Preferencias{
        Comidas:   "asado, pollo",
        Películas: "coco",
        Animes:    "shingeki no kyojin",
    },
}
```

De la misma forma, para acceder a un valor o modificarlo dentro de la estructura “gustos” desde “persona”.

```
{}
fmt.Println(p2.Gustos.Animes)
p2.Gustos.Deportes = "fútbol"
```

O podríamos agregarle directamente la estructura completa.

```
{}
p3 := Persona{}
p3.Nombre = "Ulises"
p3.Edad = 15
p3.Gustos = Preferencias{Comidas: "verduras", Películas: "Entrenando a mi dragón"}
```

ETIQUETAS DE ESTRUCTURAS

Go Bases

Dentro de nuestras estructuras podemos definir etiquetas o anotaciones que hagan referencia a cada uno de los campos que aparecen luego de declarar el tipo de dato.

{}

```
type MiEstructura struct {
    Campo1  string `miEtiqueta:"valor"`
    Campo2  string `miEtiqueta:"valor"`
    Campo3  string `miEtiqueta:"valor"`
}
```

Por ejemplo, cuando trabajamos con aplicaciones REST, podemos, mediante las etiquetas, especificar el nombre de cada campo en formato JSON.

```
type Persona struct {
   PrimerNombre string `json:"primer_nombre"`
   Apellido     string `json:"apellido"`
   Teléfono     string `json:"teléfono"`
   Dirección    string `json:"dirección"`
}
```

Para hacer esta conversión, Go nos proporciona una paquete llamado **encoding/json**

```
import (
   "encoding/json"
)
```

Declaramos la estructura y utilizamos la función Marshal, que nos devuelve los bytes de la representación de nuestra estructura codificada en JSON y un error, en caso que haya habido algún problema al realizar esa conversión.

```
{}
```

```go
p := Persona{"Celeste", "Rodriguez", "43434343", "Calle falsa 123"}
miJSON, err := json.Marshal(p)

fmt.Println(string(miJSON))
fmt.Println(err)
```

El JSON nos quedará de la siguiente manera:

{}

```
{
   "primer_nombre":"Celeste",
   "apellido":"Rodriguez",
   "teléfono":"43434343",
   "dirección":"Calle falsa 123"
}
```

También podemos definir una estructura con etiquetas personalizadas. Por ejemplo, una etiqueta bd con el nombre que queramos utilizar para una base de datos.

```
type Persona struct {
   PrimerNombre string `bd:"primer_nombre"`
   Apellido     string `bd:"apellido"`
   Telefono     string `bd:"teléfono"`
   Dirección    string `bd:"dirección"`
}
```

Para acceder a ella vamos a utilizar **reflection**, este paquete nos proporciona funcionalidades para poder obtener información de los objetos en tiempo de ejecución.

```
import (
   "reflect"
)
```

Para obtener el tipo de reflection sobre nuestra estructura lo hacemos de la siguiente manera:

```
persona := Persona{}
p := reflect.TypeOf(persona)
```

Incluso podemos ver información sobre nuestra estructura como el nombre que le definimos y el tipo.

```
fmt.Println("Type: ", p.Name())
fmt.Println("Kind: ", p.Kind())
```

Con el método NumField podemos obtener el número de campos que tenemos en nuestra estructura, esto nos va a servir para poder recorrerla.

```
{}
for i := 0; i < p.NumField(); i++ {
}
```

Con el método Field podemos obtener el campo de nuestra estructura pasándole como parámetro el índice.

```
{}
for i := 0; i < p.NumField(); i++ {
   field := p.Field(i)
}
```

Además, si queremos acceder al valor de la etiqueta definida lo haríamos de la siguiente manera:

```
for i := 0; i < p.NumField(); i++ {
    field := p.Field(i)
    tag := field.Tag.Get("bd")
}
```

MÉTODOS

Go Bases

// ¿Qué son los Métodos?

Go no tiene clases. Sin embargo, se pueden definir métodos en los tipos de datos.
Un método es una función con un argumento de receptor especial. El receptor aparece en su propia lista de argumentos entre la palabra clave func y el nombre del método.

Declararemos una estructura `circulo` y en ella agregaremos un campo que utilizaremos para almacenar el radio.

```
type circulo struct {
    radio float64
}
```

Para definir un método de nuestra estructura lo hacemos de la misma forma que declarando una función, pero debemos especificar que es un método de nuestra estructura Circulo

```
{}  func metodo(){ }
```

Para especificar que es un método, debemos agregar entre la palabra reservada func y el nombre del método a qué estructura corresponde, de la siguiente manera:

```
{}  func (v MiEstructura) metodo(){ }
```

Definiendo la variable que vamos a utilizar para manipular nuestra estructura desde el método (en el ejemplo, la variable v), y la estructura a la que corresponde.

Definimos nuestro primer método de la estructura Circulo, con la variable c de la estructura Circulo podemos acceder a sus variables.

```
func (c circulo) area() float64 {
   return math.Pi * c.radio * c.radio
}
```

Declaremos el método perímetro

```
func (c circulo) perimetro() float64 {
   return 2 * math.Pi * c.radio
}
```

* math es un paquete que nos proporciona Go para realizar cálculos matemáticos más complejos, en este caso para obtener el valor de Pi.

// ¿Qué son los punteros?

Cuando creamos una función y le pasamos una variable como argumento, lo que hace la función es una *copia* del valor de la variable y trabaja con ese valor. Por lo que la variable que pasamos como argumento no se modifica.
Los punteros son muy útiles en los casos en los que queremos pasar una variable como argumento a una función para que su valor sea modificado, esto es lo que se conoce como pasar valores por *referencia* a una función.

Para modificar variables de nuestra estructura en el método debemos pasar el valor como referencia e indicarlo como un **puntero**.
De no hacerlo, la variable no será modificada al salir del scope del método.

```
{}
func (c *circulo) setRadio(r float64) {
    c.radio = r
}
```

Para ejecutar nuestros métodos, debemos hacerlo de la siguiente manera:

{}

```
func main() {
    c := circulo{radio: 5}
    fmt.Println(c.area())
    fmt.Println(c.perimetro())
    c.setRadio(10)
    fmt.Println(c.area())
    fmt.Println(c.perimetro())
}
```

COMPOSICIÓN

Go Bases

En otros lenguajes existe el concepto de herencia, esto consiste en tener una clase padre y su/s clase/s hija/s. La clase padre es la que transmite su código a las clases hijas.

// ¿Qué pasa con la herencia en Go?

El concepto de herencia no existe en Go, pero tenemos una composición que utiliza la estructura padre como campo en nuestras estructuras hijas, esto se conoce como (embedding structs).

El propósito de la composición en Go es el de poder crear programas más grandes a partir de piezas más pequeñas. Esto nos ayuda a diseñar distintos tipos de datos sobre los cuales implementar distintos comportamientos.

Veamos un ejemplo...

Declaramos nuestra estructura base **Vehículo**, y en ella agregaremos los campos kilómetros y tiempo.

```
type Vehiculo struct {
   km     float64
   tiempo float64
}
```

Declaremos un método para nuestra estructura Vehículo que nos imprima en pantalla el valor de sus campos.

```
func (v Vehiculo) detalle() {
   fmt.Printf("km:\t%f\ntiempo:\t%f\n", v.km, v.tiempo)
}
```

Declaremos una de nuestras estructuras compuestas, la estructura Auto. En ella agreguemos un campo de tipo Vehículo:

```
type Auto struct {
    v Vehiculo
}
```

Agreguemos un método que reciba tiempo en minutos y se encargue de realizar el cálculo de distancia en base a 100 km/h

```
func (a *Auto) correr(minutos int) {
    a.v.tiempo = float64(minutos) / 60
    a.v.km = a.v.tiempo * 100
}
```

y el método detalle que llame al método de la estructura “padre”

```
func (a *Auto) detalle() {
    fmt.Println("\nV:\tAuto")
    a.v.detalle()
}
```

Declaramos nuestra otra estructura compuesta Moto

```
type Moto struct {
    v Vehiculo
}
```

Agregamos el método Correr que recibe el tiempo en minutos y hace el cálculo en base a 80 km/h

```
func (m *Moto) correr(minutos int) {
    m.v.tiempo = float64(minutos) / 60
    m.v.km = m.v.tiempo * 80
}
```

y el método detalle

```
func (m *Moto) detalle() {
    fmt.Println("\nV:\tMoto")
    m.v.detalle()
}
```

Por último ejecutamos nuestros métodos en el main del proyecto y vemos los resultados:

```
{}
```

```
auto := Auto{}
auto.correr(360)
auto.detalle()

moto := Moto{}
moto.correr(360)
moto.detalle()
```

INTERFACES

Go Bases

// ¿Qué son las interfaces?

Una interfaz es una forma de definir métodos que deben ser utilizados pero sin definirlos.

¿Para qué se utilizan?

Las interfaces son utilizadas para brindar modularidad al lenguaje.

// Ejemplo #1

Generamos una estructura “circulo” y una función detalle que mostrará el área y el perímetro de la figura.

{}

```
type circulo struct {
  radio float64
}

func (c circulo) area() float64 {
  return math.Pi * c.radio * c.radio
}

func (c circulo) perimetro() float64 {
  return 2 * math.Pi * c.radio
}
```

// Ejemplo #2

Creamos una función que imprima el área y el perímetro que generamos para dicho objeto:

```
func detalle(c circulo) {
   fmt.Println(c)
   fmt.Println(c.area())
   fmt.Println(c.perimetro())
}
```

y ejecutaremos la función:

```
func main() {
   c := circulo{radio: 5}
   detalle(c)
}
```

// ¿Qué pasará si queremos generar más figuras geométricas utilizando nuestra función ‘details’?

Aquí es donde entran en juego las interfaces. Estas nos permiten implementar el mismo comportamiento a diferentes estructuras.

// Ejemplo #1

A continuación, definimos nuestra interfaz “geometria” que contiene dos métodos que adoptarán nuestros objetos.

{}

```
type geometria interface {
    area() float64
    perimetro() float64
}
```

// Ejemplo #1

Generamos otro objeto geométrico, en este caso un rectángulo que lógicamente, tenga los mismos métodos:

```
type recta struct {
   ancho, alto float64
}
func (r recta) area() float64 {
   return r.ancho * r.alto
}
func (r recta) perimetro() float64 {
   return 2*r.ancho + 2*r.alto
}
```

Modificaremos nuestra función detaille, para que en lugar de recibir un círculo, reciba una figura geométrica:

```
func details(g geometria) {
   fmt.Println(g)
   fmt.Println(g.area())
   fmt.Println(g.perimetro())
}
```

// Ejemplo #1

De esta forma podemos seguir agregando figuras geométricas sin necesidad de modificar nuestra función:

{}

```
func main() {
    r := recta{ancho: 3, alto: 4}
    c := circulo{radio: 5}
    detalle(r)
    detalle(c)
}
```

// Ejemplo #1

En el siguiente ejemplo, creamos una función que nos genere el objeto:

```
func nuevoCirculo(valores float64) geometria {
   return &circulo{radio: valores}
}
```

Ejecutamos el main del programa:

```
func main() {
   c := nuevoCirculo(3)
   fmt.Println(c.area())
   fmt.Println(c.perimetro())
}
```

// ¿Qué pasará si queremos re utilizar nuestra función para poder implementar varias figuras geométricas?

En este caso tendremos que crear una función que retorne una interface que pueda implementar todos nuestros objetos geométricos.

A continuación un ejemplo.

// Ejemplo

Vamos a reemplazar nuestra función ‘newCircle’ por ‘newGeometry’ y le pasaremos 2 constantes que definimos para especificar cuál es el objeto que generamos:

{}

```
const (
    tipoRecta   = "RECTA"
    tipoCirculo = "CIRCULO"
)
func nuevaGeometria(tipoGeo string, valores ...float64) geometria {
    switch tipoGeo {
    case tipoRecta:
        return recta{ancho: valores[0], alto: valores[1]}
    case tipoCirculo:
        return circulo{radio: valores[0]}
    }
    return nil
}
```

// Ejemplo

Implementación en nuestro main y corremos el programa:

{}

```
func main() {
    r := nuevaGeometria(tipoRecta, 2, 3)
    fmt.Println(r.area())
    fmt.Println(r.perimetro())
    c := nuevaGeometria(tipoCirculo, 2)
    fmt.Println(c.area())
    fmt.Println(c.perimetro())
}
```

// Interfaces vacías

Son aquellas interfaces que no tienen métodos declarados.

¿Para qué se utilizan?

La utilidad de estas interfaces, es proveernos un tipo de datos “comodín”, es decir, almacenar valores que sean de un tipo de datos desconocido, o que pueda variar dependiendo el flujo del programa

// ¿Cómo se declara una variable con este tipo?

```
{}  var miVariable interface{}
```

¿Cómo funcionan?

Como vimos anteriormente, una interfaz define el ***conjunto mínimo de métodos*** que un tipo de datos debe implementar para poder ser considerado como implementador de dicha interfaz.
Por lo tanto, todos los tipos de datos son considerados implementadores de la interfaz vacía, ***porque implementan al menos cero métodos.***

// Ejemplo de un uso de interfaz vacía

{}

```
type ListaHeterogenea struct {
   Dato []interface{}
}

func main() {
   l := ListaHeterogenea{}
   l.Dato = append(l.Dato, 1)
   l.Dato = append(l.Dato, "hola")
   l.Dato = append(l.Dato, true)

   fmt.Printf("%v\n", l.Dato)

}
```

// Resultado del ejemplo

output

```
> $ go run interfaces_vacias.go
[1 hola true]
```

// Type assertion (aserción de tipos)

La aserción de tipos provee acceso al tipo de datos exacto que está abstraído por una interfaz

{}

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

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) // panic
fmt.Println(f)
```

Gracias.

IT BOARDING

BOOTCAMP