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 que 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, podríamos 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
   Profesion string
   Peso      float64
}
```

Para instanciar una estructura, podemos utilizar distintas formas:

Indicar todos los valores que queremos que tengan los campos.

```
{}  p1 := Persona{"Celeste", "Mujer", 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.

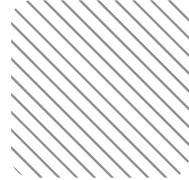
```
{}  p2 := Persona{
        Nombre:    "Nahuel",
        Genero:    "Hombre",
        Edad:      30,
        Profesion: "Ingeniero",
        Peso:      77
    }
```

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

```
{}  p2.Peso
```

Para asignar/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.

```
{}
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, podríamos tener una estructura **gustos** dentro de nuestra estructura **persona**.
Para eso debemos declarar nuestra estructura gustos.

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

Y asignaremos un campo de tipo gustos a nuestra estructura **persona**.

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

Hacemos lo siguiente para instanciar nuestra estructura.

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

Podríamos instanciarla haciendo referencia a cada campo.

{}

```
p2 := Persona{
    Nombre:    "Nahuel",
    Genero:    "Hombre",
    Edad:      30,
    Profesion: "Ingeniero",
    Peso:      77,
    Gustos: Preferencias{
        Comidas:   "asado, pollo",
        Peliculas: "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 = "futbol"
```

O podríamos agregarle directamente la estructura completa.

```
{}
p3 := Persona{}
p3.Nombre = "Ulises"
p3.Edad = 15
p3.Gustos = Preferencias{Comidas: "verduras", Peliculas: "entrenando a mi dragon"}
```

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, especificarle el nombre de cada campo en formato JSON.

```go
type Persona struct {
    PrimerNombre string `json:"primer_nombre"`
    Apellido     string `json:"apellido"`
    Telefono     string `json:"telefono"`
    Direccion    string `json:"direccion"`
}
```

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

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

Instanciamos la estructura y utilizamos la función Marshal, esta función nos devolverá una conversión a JSON de nuestra estructura en formato byte y un error en caso que haya habido algún problema en realizar esa conversión.

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

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

El JSON nos quedaría de la siguiente manera:

```
{}
{
   "primer_nombre":"Celeste",
   "apellido":"Rodriguez",
   "telefono":"43434343",
   "direccion":"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:"telefono"`
   Direccion    string `bd:"direccion"`
}
```

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")
}
```

Go orientado a objetos - Métodos

// ¿Qué son los Métodos?

“Los métodos de una clase son funciones que sirven para manipular las variables de dicha clase”.

IT BOARDING

BOOTCAMP

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.

Definiremos 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
}
```

Declararemos el método perimetro

```
func (c Circulo) perim() 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

Para modificar variables de nuestra estructura en el método debemos indicarlo como puntero.

De no indicarlo la variable no será modificada al salir del scope del método.

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

Para ejecutar nuestros métodos lo haríamos de la siguiente manera:

{}

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

Go orientado a objetos - Herencia

// ¿Cuál es el concepto de herencia?

“La herencia 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”.

// ¿Cuál es el concepto de herencia?

El concepto de herencia no existe en Go, pero podemos replicando utilizando una clase Padre como campo en nuestras clases hijas,

vamos a ver un ejemplo...

Declaramos nuestra clase padre **vehículos**, y en ella agregaremos los campos kilómetros y tiempo

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

declararemos un método a nuestra clase 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)
}
```

Declararemos una de nuestras clases hijas, la clase Auto. En ella agregaremos un campo de tipo Vehiculo:

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

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

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

Declaramos nuestra otra clase hija Moto

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

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

// ¿Qué son las interfaces?

“Las interfaces nos permiten definir los métodos de las estructuras que queramos representar y de esta manera implementar el mismo comportamiento a objetos que no tienen relación entre sí, esto nos permite utilizar polimorfismo en Go y hacer que nuestro código sea escalable”.

IT BOARDING

BOOTCAMP

En el siguiente ejemplo, generamos una estructura circle y una función details que nos mostrará el Área y el Perímetro de la figura.

{}

```go
package main

import (
  "fmt"
  "math"
)

type circle struct {
  radius float64
}

func (c circle) area() float64 {
  return math.Pi * c.radius * c.radius
}
func (c circle) perim() float64 {
  return 2 * math.Pi * c.radius
}
```

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

```
{}
func details(c circle) {
    fmt.Println(c)
    fmt.Println(c.area())
    fmt.Println(c.perim())
}
```

y ejecutaremos la función:

```
{}
func main() {
    c := circle{radius: 5}
    details(c)
}
```

// **¿Qué pasaría si quisiéramos generar más figuras geométricas y quisiéramos re utilizar nuestra función details?**

Aquí es donde entran en juego las interfaces, que nos van a permitir implementar el mismo comportamiento a diferentes objetos.

Vamos a generar una interface geometry y definiendo en ellas los métodos que nuestros objetos tendrán:

{}

```
type geometry interface {
    area() float64
    perim() float64
}
```

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

```
type rect struct {
   width, height float64
}
func (r rect) area() float64 {
   return r.width * r.height
}
func (r rect) perim() float64 {
   return 2*r.width + 2*r.height
}
```

Modificaremos nuestra función details, para que el lugar de recibir un círculo, reciba una figura geométrica

```
func details(g geometry) {
   fmt.Println(g)
   fmt.Println(g.area())
   fmt.Println(g.perim())
}
```

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

{}

```
func main() {
    r := rect{width: 3, height: 4}
    c := circle{radius: 5}
    details(r)
    details(c)
}
```

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

```
func newCircle(values float64) circle {
   return circle{radius: values}
}
```

Ejecutamos el main del programa:

```
func main() {
   c := newCircle(2)
   fmt.Println(c.area())
   fmt.Println(c.perim())
}
```

// **¿Qué pasaría si quisiéramos reutilizar nuestra función para poder implementar varias figuras geométricas?**

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

Vamos a reemplazar nuestra funcion newCircle por newGeometry, y les pasaremos 2 constantes que definimos para especificar cuál es el objeto que generaramos:

{}

```
const (
    rectType   = "RECT"
    circleType = "CIRCLE"
)
func newGeometry(geoType string, values ...float64) geometry {
    switch geoType {
    case rectType:
        return rect{width: values[0], height: values[1]}
    case circleType:
        return circle{radius: values[0]}
    }
    return nil
}
```

Implementamos nuestro main y corremos el programa

{}

```
func main() {
    r := newGeometry(rectType, 2, 3)
    fmt.Println(r.area())
    fmt.Println(r.perim())
    c := newGeometry(circleType, 2)
    fmt.Println(c.area())
    fmt.Println(c.perim())
}
```

EXTRAS

Algunos temas para profundizar

// The “empty interface” (interface vacía)

Es un tipo de interface especial que no tiene métodos especificados.

Se representa así: `interface{}`

Una interface vacía puede almacenar valores de cualquier tipo porque todas las interfaces implementan por lo menos cero métodos.

IT BOARDING

BOOTCAMP

// Typecasting (encasillamiento de tipos)

Es forzar que un dato de un tipo sea interpretado como un dato de un tipo diferente.

// Type assertion (aserción de tipos)

Proveen acceso al tipo exacto de variable de una interface. Si el tipo de dato está presente en la interface, entonces recuperará el tipo de dato real que está siendo albergado por la interface.

// Para concluir

Hemos visto cómo diseñar y utilizar Estructuras e Interfaces en Go.

¡A seguir aprendiendo!

IT BOARDING

BOOTCAMP

Gracias.

IT BOARDING

BOOTCAMP