GO

BOOTCAMP

Clase en vivo

//Go Bases

IT BOARDING

BOOTCAMP

Objetivos de esta clase

- Conocer que es una estructura en Golang (struct)
- Entender cómo se compone una estructura.
- Entender el uso de tags en una estructura.
- Comprender la creación de métodos de una estructura.
- Comprender el concepto de Composición

Índice

01 Repaso

02 Composición

IT BOARDING
BOOTCAMP

1 | Repaso

IT BOARDING

BOOTCAMP

Estructuras

Definir una estructura

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 Person struct {
    Name        string
    Gender      string
    Age         int
    Profession  string
    Weight      float64
}
```

Instanciar una estructura

Indicar todos los valores que queremos que tengan los campos.

```
{}
p1 := Person{"Celeste", "Woman", 34, "Engineer", 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 := Person{
    Name:       "Nahuel",
    Gender:     "Man",
    Age:        30,
    Profession: "Engineer",
    Weight:     77,
}
```

Asignación en una estructura

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 Preferencias.

```
{}
type Preferences struct {
    Foods    string
    Movies   string
    Series   string
    Animes   string
    Sports   string
}
```

Asignación en una estructura

Asignaremos un campo gustos a nuestra estructura **persona,** la cual será del tipo **Preferencias**.

```
type Person struct {
    Name          string
    Gender        string
    Age           int
    Profession    string
    Weight        float64
    Likes         Preferences
}
```

Hacemos lo siguiente para instanciar nuestra estructura:

```
p1 := Person{"Celeste", "Woman", 34, "Engineer", 65.5, Preferences{"chicken", "titanic", "", "", ""}}
```

Estructuras anidadas

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

```
{}
fmt.Println(p2.Likes.Animes)
p2.Likes.Sports = "soccer"
```

O podríamos agregarle directamente la estructura completa.

```
{}
p3 := Person{}
p3.Name  = "Ulises"
p3.Age   = 15
p3.Likes = Preferences{Foods: "vegetables", Movies: "How to train your dragon"}
```

Etiquetas

Etiquetas en estructuras

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

```
{}
type Person struct {
    FirstName      string `json:"first_name"`
    LastName       string `json:"last_name"`
    Phone          string `json:"phone"`
    Location       string `json:"location"`
}
```

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

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

Métodos

Métodos en estructuras

Comenzamos creando una estructura representativa de un Círculo, con el siguiente campo: radio.

```
{}
type Circle struct {
    radius      float64
}
```

Métodos en estructuras

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

{}

```
func (c Circle) area() float64 {
   return math.Pi * c.radius * c.radius
}
func (c Circle) perimeter() float64 {
   return 2 * math.Pi * c.radius
}
func (c Circle) setRadius(newRadius float64){
   c.radius = newRadius
}
```

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

Llamado de métodos

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

```
{}
```

```go
func main() {
    c := Circle{radius: 5}
    fmt.Println(c.area())
    fmt.Println(c.perimeter())

    c.setRadius(10)

    fmt.Println(c.area())
    fmt.Println(c.perimeter())
}
```

2 | Composición

IT BOARDING

BOOTCAMP

En otros lenguajes (lenguajes orientados a objetos), existe el concepto de *herencia*. Esto consiste en tener una clase padre y clases hijas, donde las clases hijas heredan definiciones, métodos y atributos de la clase padre.

// ¿Qué pasa con la herencia en Go?

El concepto de herencia no existe en Go, pero sí tenemos **composición**, que consiste en extender estructuras a partir de la creación de otras nuevas, que incluyan a la anterior. Esto se conoce como embedding structs.

// ¿Qué pasa con la composición en Go?

El propósito de la composición en Go es crear estructuras más complejas a partir de otras más pequeñas. Esto nos ayuda a diseñar distintos tipos de datos, permitiéndonos crear comportamientos específicos para componentes con diferentes significados, pero que comparten una definición que los generaliza.

// Herencia Vs Composición

- Se expresa a la **herencia** como una relación **es-un/a**. Es decir, una clase que extiende a otra puede ocupar el lugar en una porción de código que requiera una instancia de la clase padre.
- Se expresa a la **composición** como una relación **tiene-un/a**, significando que si una porción de código requiere una instancia del tipo de la estructura que forma parte de la composición, no puedo poner en su lugar una instancia de la estructura que la incluye.

// Composición y Herencia #1

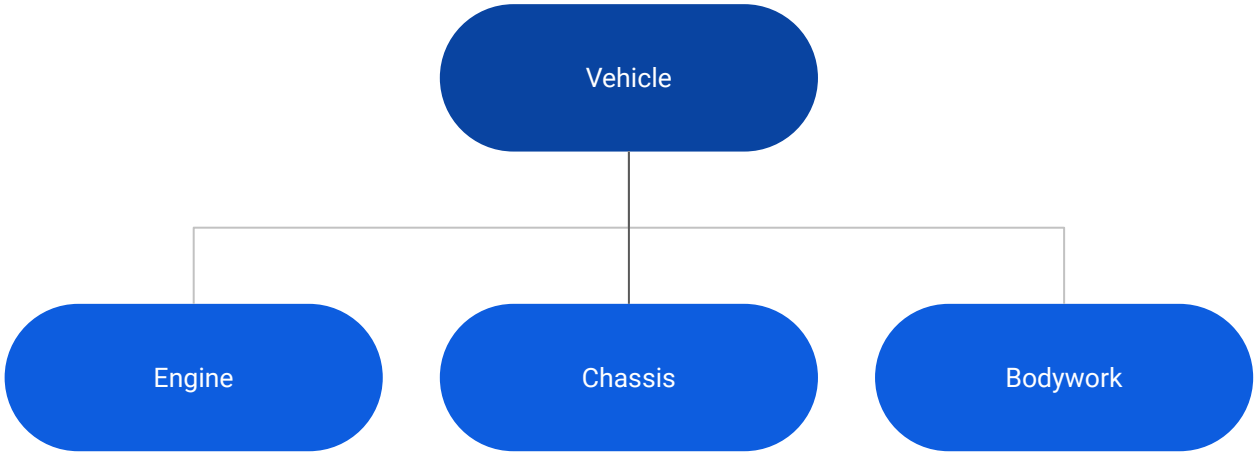
Podemos imaginar a la composición de la siguiente manera:

Supongamos que tenemos un Vehículo cualquiera, el mismo de manera muy simple podemos decir que está compuesto por 3 partes principales:

- ***Motor***
- ***Chásis***
- ***Carrocería***

// Composición y Herencia #2

Podemos entender a la composición de un Vehículo cómo:



// Composición y Herencia #3

Ahora, pensemos que podemos tener vehículos puntuales: Auto, Moto, Camión, etc.

Los cuales pueden diferir en comportamientos, es decir, métodos por ejemplo:

- ***Encender.***
- ***Mostrar Información.***

// Composición y Herencia #4

En un lenguaje de programación que soporte herencia, como Java o C++, se crearía una clase base "Vehicle" y luego tendrías clases "Car" y "Motorcycle" que heredan de "Vehicle". Sin embargo, en Go, la herencia no se maneja de la misma manera. En lugar de la herencia directa, usarías interfaces para definir comportamientos comunes y composición para reutilizar la funcionalidad.

Veamos un ejemplo #1

Declaremos las estructura para el Motor, Chasis y Carrocería

```
{}
type Engine struct {
    Horsepower int
    Type       string
}
type Chassis struct {
    Material string
}
type Bodywork struct {
    Color string
}
```

Veamos un ejemplo #2

Vehicle, es una interfaz que cualquier tipo de vehículo podría implementar

{}

```
type Vehicle interface {
    Start()
    DisplayInfo()
}
```

Veamos un ejemplo #3

Ahora definimos dos estructuras, una para Auto y otra para Moto, las que se componene de las estructurs antes creadas:

```
type Car struct {
    Engine
    Chassis
    Body
    NumberOfDoors int
}
type Motorcycle struct {
    Engine
    Chassis
    Body
}
```

Veamos un ejemplo #4

Veamos a continuación, como Car, implementa los métodos de la interfaz Vehicle.

```
func (c Car) Start() {
    fmt.Println("The car is starting with a roar!")
}
func (c Car) DisplayInfo() {
    fmt.Printf("This car has %d doors, a %s body, a %s chassis, and a %d horsepower %s engine.\n",
        c.NumberOfDoors, c.Body.Color, c.Chassis.Material, c.Engine.Horsepower, c.Engine.Type)
}
```

Veamos un ejemplo #5

Veamos a continuación, como Motorcycle, implementa los métodos de la interfaz Vehicle.

```
{}
func (m Motorcycle) Start() {
    fmt.Println("The motorcycle is starting with a vroom!")
}
func (m Motorcycle) DisplayInfo() {
    fmt.Printf("This motorcycle has a %s body, a %s chassis, and a %d
horsepower %s engine.\n",
        m.Body.Color, m.Chassis.Material, m.Engine.Horsepower,
m.Engine.Type)
}
```

Veamos un ejemplo #6

Creemos una instancia de Car, y otra de Motorcycle:

```
{}
    myCar := Car{
        Engine:        Engine{Horsepower: 250, Type: "V4"},
        Chassis:       Chassis{Material: "Steel"},
        Body:          Body{Color: "Red"},
        NumberOfDoors: 4,
    }

    myMotorcycle := Motorcycle{
        Engine:  Engine{Horsepower: 150, Type: "V2"},
        Chassis: Chassis{Material: "Aluminum"},
        Body:    Body{Color: "Black"},
    }
```

Veamos un ejemplo #7

Hacemos uso de los métodos definidos para cada uno:

```
{}
    myCar.Start()
    myCar.DisplayInfo()

    myMotorcycle.Start()
    myMotorcycle.DisplayInfo()
```

```
{}
go run main.go
The car is starting with a roar!
This car has 4 doors, a Red body, a Steel chassis, and a250 horsepower V4 engine.
The motorcycle is starting with a vroom!
This motorcycle has a Black body, a Aluminum chassis, and a150 horsepower V2 engine.
```

Veamos un ejemplo #8

Podemos tratar a un Car como así tambien a un Motorcylce como un Vehicle, ya que implementan los mismos métodos:

{}

```
var vehicle Vehicle
vehicle = myCar // myCar "is a" Vehicle
vehicle.Start()
vehicle.DisplayInfo()
vehicle = myMotorcycle // myBike "is a" Vehicle
vehicle.Start()
vehicle.DisplayInfo()
```

{}

```
go run main.go
The car is starting with a roar!
This car has 4 doors, a Red body, a Steel chassis, and a250 horsepower V4 engine.
The motorcycle is starting with a vroom!
This motorcycle has a Black body, a Aluminum chassis, and a150 horsepower V2 engine.
```

A codear...

Conclusiones

Hoy conocimos las estructuras en GO. Aprendimos a asignarles métodos y etiquetas.

Además reforzamos la composición y sus distintas aplicaciones dentro del código, las cuales nos permiten darle rigidez y estructura a nuestros programas y su comportamiento.

Gracias.

IT BOARDING

BOOTCAMP