DIGITALHOUSE

Composición

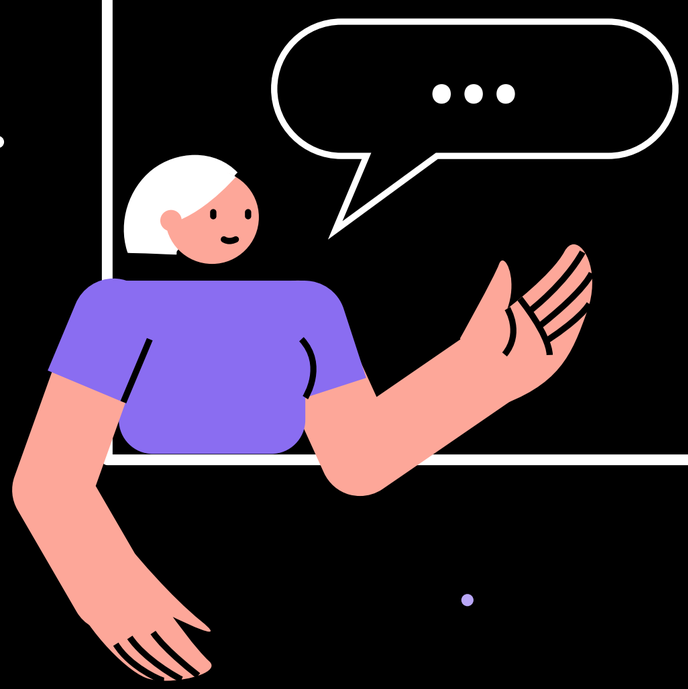
Certified Tech Developer
The Ultimate Degree

En otros lenguajes existe el concepto de **herencia**. Este consiste en tener una clase padre y sus clases hijas. 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 poder crear programas más grandes a partir de piezas más pequeñas. Esto nos ayuda a diseñar diversos tipos de datos sobre los cuales implementar distintos comportamientos.



Composición

Podemos imaginar la composición como una receta de cocina, que está compuesta por otras recetas.

Supongamos que queremos cocinar un pollo frito a la barbacoa. Entonces, deberemos seguir tres pasos:

Receta de pollo frito a la barbacoa

1	2	3
Receta: Cómo trocear el pollo.	Receta: Cómo preparar el empanizado.	Receta: Cómo preparar la barbacoa.

Esto no solo nos permite ordenar y estructurar nuestras recetas de mejor manera, sino que también nos permite reutilizar nuestros pasos para construir otras recetas. Por ejemplo:

Receta de pollo troceado a la barbacoa

1	2
Receta: Cómo trocear el pollo.	Receta: Cómo preparar la barbacoa.

Veamos un ejemplo

Declaramos nuestra clase padre **Vehiculo** y en ella agregaremos los campos **km** y **tiempo**:

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

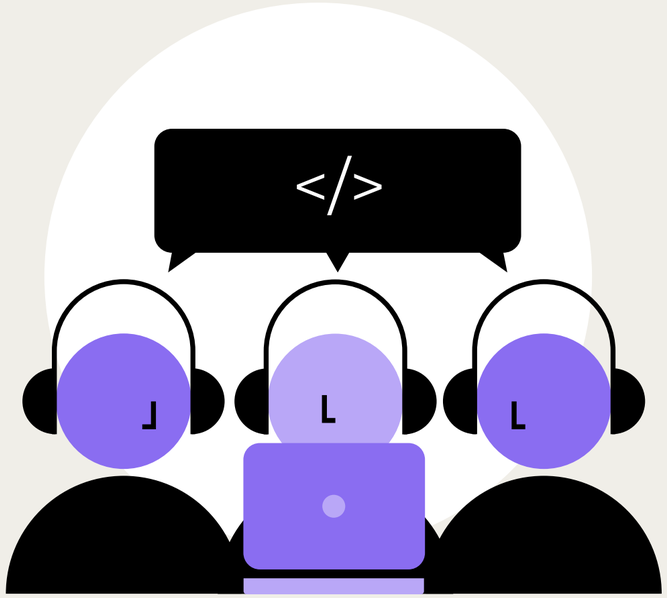
Declaremos un método para nuestra clase **Vehiculo** que nos imprima en pantalla el valor de sus campos:

```
$
func (v Vehiculo) detalle() {

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

Declaremos la estructura **Auto**. En ella, agreguemos un campo de tipo **Vehiculo**. Aquí estamos embebiendo la estructura.

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



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

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

Ahora, el método **Detalle** que llame al método de la clase padre:

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

Declaramos la estructura **Moto**, la cual tiene embebida a la estructura **Vehiculo**:

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

Agregamos el método **Correr** que recibe el tiempo en minutos y hace el cálculo basándose en 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()
```

DIGITALHOUSE

¡Muchas gracias!