GO

BOOTCAMP

Clase en vivo

//Go Bases

IT BOARDING

BOOTCAMP

Objetivos de esta clase

- Profundizar el concepto de composición.
- Entender qué es un puntero.
- Conocer qué es una interfaz.
- Aplicación de interfaces.
- Entender el uso de assertion para acceder a interfaces.

Índice

IT BOARDING
BOOTCAMP

1 | Repaso

IT BOARDING

BOOTCAMP

Composición

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 **Autor**, y en ella agregaremos los campos nombre y apellido.

```
type Autor struct {
    FirstName   string
    LastName    string
}
```

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

```
func (a *Autor) fullName() string {
    return fmt.Sprintf("%s %s", a.FirstName, a.LastName)
}
```

Declaremos una estructura, **Libro**. En ella agreguemos un campo de tipo **Autor**:

```
type Book struct {
    Title    string
    Content  string
    Autor    Autor
}
```

Agreguemos un método que nos permita imprimir los valores de nuestro **libro**.

```
func (b *Book) information() {
    fmt.Println("Title: ", b.Title)
    fmt.Println("Content: ", b.Content)
    fmt.Println("Autor: ", b.Autor.fullName())
}
```

Y de esta manera ya podemos llamar a nuestros métodos.

{}

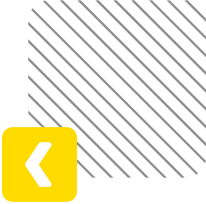
```
func main() {
    autor := Autor{
        "Juan",
        "Lopez",
    }

    book := Book{
        Title:   "Inheritance in Go",
        Content: "Go has composition instead of inheritance",
        Autor:   autor,
    }
    book.information()
}
```

Punteros

Punteros: ejemplo incrementar

{}

```go
package main
import "fmt"
// Increase receives an integer pointer
func Increase(v *int) {
   // We dereference the variable v to obtain
   // its value and increment it by 1
   *v++
}
func main() {
   var v int = 19
   // Increase function receives a pointer
   // we use the address operator &
   // to pass the memory address of v
   Increase(&v)
   fmt.Println("The value of v now reads:", v)
}
```



En el ejemplo podemos ver que se pueden pasar punteros como parámetros a una función utilizando los operadores `*` y `&`. También vemos el comportamiento de estos al incrementar el valor de la variable `v`. El código produce el siguiente resultado:

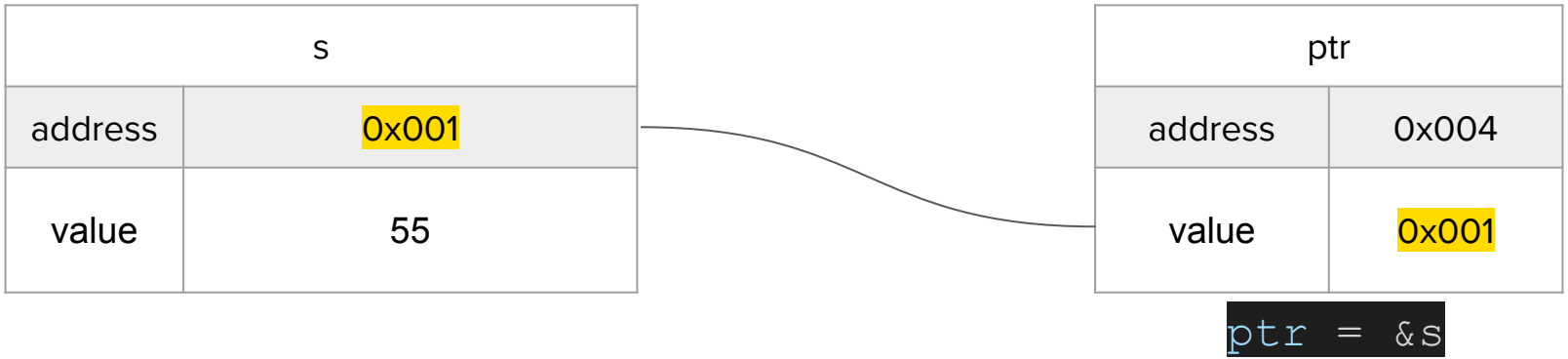
TERMINAL

```
go run main.go

The value of v now reads: 20
```

¿Dónde y qué almacena el puntero?

Suponiendo que tenemos una variable de tipo int y un puntero a int, como en la siguiente tabla:

name	type	address
s	int	0x001
ptr	*int	0x004



2 | Interfaces

IT BOARDING

BOOTCAMP

// ¿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.

diagrama

// Ejemplo #1

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

{}

```
type circle struct {
  radius float64
}

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

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

// Ejemplo #2

Creamos una función que imprima el área y el perímetro que generamos para dicha estructura:

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

y ejecutaremos la función:

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

// Ejemplo #1

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

{}

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

// Ejemplo #1

Generamos otra estructura del tipo 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) perimeter() float64 {
   return 2*r.width + 2*r.height
}
```

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

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

// Ejemplo #1

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)
}
```

// Ejemplo #1

En el siguiente ejemplo, creamos una función que nos cree una variable del tipo de la estructura:

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

Ejecutamos el main del programa:

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

// ¿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 la estructura que generamos:

{}

```
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
}
```

// Ejemplo

Implementación en nuestro main y corremos el programa:

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

    c := newGeometry(circleType, 2)
    fmt.Println(c.area())
    fmt.Println(c.perimeter())
}
```

Interfaces Vacías

// Interfaces vacías

Son aquellas interfaces que no tienen métodos declarados.

¿Para qué se utilizan?

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

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

```
{}  var myVariable interface{}  // or var myVariable any
```

¿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 SliceAny struct {
   Data []interface{}
}

func main() {
   l := SliceAny{}
   l.Data = append(l.Data, 1)
   l.Data = append(l.Data, "hello")
   l.Data = append(l.Data, true)

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

}
```

// Resultado del ejemplo

output

```
> $ go run empty_interfaces.go
[1 hello true]
```

Type assertion

// 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{} = "hello"

s := i.(string)
fmt.Println(s)

s, ok := i.(string)
fmt.Println(s, ok)

f, ok := i.(float64)
fmt.Println(f, ok)
```

A codear...

Conclusiones

En esta clase aprendimos los conceptos de composición y punteros, los cuales podremos usar al momento de desarrollar para construir nuestros programas.

También trabajamos sobre el concepto de Interfaces y su aplicación dentro de GO. Las mismas nos proveen modularidad al momento de desarrollar nuestras aplicaciones.

Gracias.

IT BOARDING

BOOTCAMP