INTRODUCCIÓN A LA CLASE

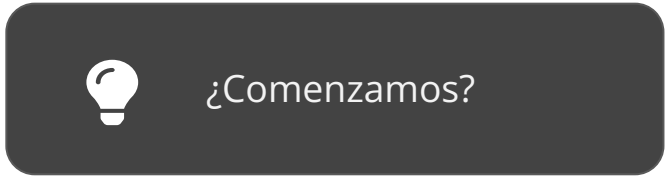
Go Bases

¿Qué aprenderemos hoy?

¡Llegó el momento de arrancar con nuestra segunda clase de Go Bases!

Los objetivos para esta clase son:

- Comprender y Generar Funciones en Go
- Conocer cómo retornar funciones
- Aprender qué es y para qué sirven los Elipsis
- Comprender los multi retornos de funciones



FUNCIONES

Go Bases

// ¿Qué es una función?

“La función es una porcion de codigo que ejecuta una tarea en específica.
Todos los programas tienen al menos una función main. Las funciones se diferencian entre sí por medio de un identificador que las hace únicas”

Composición de una función

Una función recibe uno, ninguno o muchos parámetros y puede retornar o no un valor.

```
{}
func miFuncion(parametros) {

}
```

Dentro de cada función, debemos generar un scope. Esto significa que todas las variables que declaremos aquí, solo vivirán dentro de nuestra función.

Composición de una función

Ahora, supongamos que tenemos cuatro variables de tipo integer y queremos saber, por cada una, si el valor es positivo, negativo o cero. Si tenemos que hacer la validación por cada una de estas variables terminamos repitiendo mucho código que podríamos reutilizar.

{}

```
func main() {
   a, b, c, d := 1, 0, 5, -3

   if a < 0 {
       fmt.Println("El numero es negativo")
   } else if a > 0{
       fmt.Println("El numero es positivo")
   } else {
       fmt.Println("El numero es cero")
   }
  ...
}
```

Composición de una función

Para poder reutilizar código vamos a crear una función que se encargue de imprimir si la variable es positiva, negativa o es cero.

Para definir una función tenemos que especificarlo con la palabra reservada **func** seguido del nombre de la función, en este caso mi función se va a llamar **inspeccionarVariable**

```
{}  func inspeccionarVariable()
```

Parámetros de una Función

Luego debemos definir los parámetros que va a recibir. En caso de no recibir parámetros simplemente dejamos los paréntesis vacíos.

En nuestro caso vamos a recibir un parámetro que va a ser la variable a analizar, para eso debemos agregar el nombre de parámetro y el tipo de dato.

```
{}  func inspeccionarVariable(numero int)
```

Parámetros de una Función

Después debemos definir el scope de la función mediante llaves, y todo lo que esté dentro de las llaves es lo que se ejecutará dentro de nuestra función.

```
{}
```

```
func inspeccionarVariable(numero int) {

}
```

Parámetros de una Función

Ahora vamos a usar la misma lógica que utilizamos anteriormente para analizar cada variable, pero lo haremos con el parámetro que definimos en nuestra función.

{}

```
func inspeccionarVariable(numero int) {
    if numero < 0 {
        fmt.Println("El numero es negativo")
    } else if numero > 0 {
        fmt.Println("El numero es positivo")
    } else{
        fmt.Println("El numero es cero")
    }
}
```

Parámetros de una Función

Luego llamaremos a nuestra función por cada variable generada, pasándole la variable como parámetro.

Vemos como el código queda más limpio, reducido y legible.

{}

```
func main() {
    a, b, c, d := 1, 0, 5, -3

    inspeccionarVariable(a)
    inspeccionarVariable(b)
    inspeccionarVariable(c)
    inspeccionarVariable(d)
}
```

Parámetros de una Función

Ahora, ¿qué sucede con una función que posee **dos** parámetros del mismo tipo de dato?

```
{}  func miFuncion(valor1 float64, valor2 float64)
```

Cuando tenemos varios parámetros en simultáneo con el mismo tipo de dato, podemos solamente definir el nombre e indicarle el tipo de dato al final.

```
{}  func miFuncion(valor1, valor2 float64)
```

Retornar un valor

Para definir a una función que esperamos que nos devuelva un valor, tenemos que indicarle el tipo de dato que esperamos al final de la función. En este caso, creamos una función que reciba dos parámetros y nos devuelva la sumatoria de ellos.

```
{}
func suma(valor1, valor2 float64) float64 {
    return valor1 + valor2
}
```

En este ejemplo obtenemos el valor que retorna la función, se lo asignamos a una variable, y luego mostramos el valor devuelto.

```
{}
func main() {
    s := suma(4, 5)
    fmt.Println(s)
}
```

Ejemplo

Vamos a realizar un ejemplo un poquito más complejo, una función a la cual le pasaremos dos valores y le indicaremos qué operación queremos realizar.
Para ello, definiremos cuatro constantes con las operaciones que queremos realizar, que son las cuatro variables matemáticas principales.

{}

```
const (
    Suma   = "+"
    Resta  = "-"
    Multip = "*"
    Divis  = "/"
)
```

Ejemplo

Generamos una función que se encargará de orquestar el tipo de operación que le definiremos por parámetro.

{}

```
func operacionAritmetica(valor1, valor2 float64, operador string) float64 {
   switch operador {
   case Suma:
       return valor1 + valor2
   case Resta:
       return valor1 - valor2
   case Multip:
       return valor1 * valor2
   case Divis:
       if valor2 != 0 {
           return valor1 / valor2
       }
   }
   return 0
}
```

Ejemplo

Y por último llamaremos a esa función y le indicaremos con las constantes que definimos, qué operación queremos realizar.

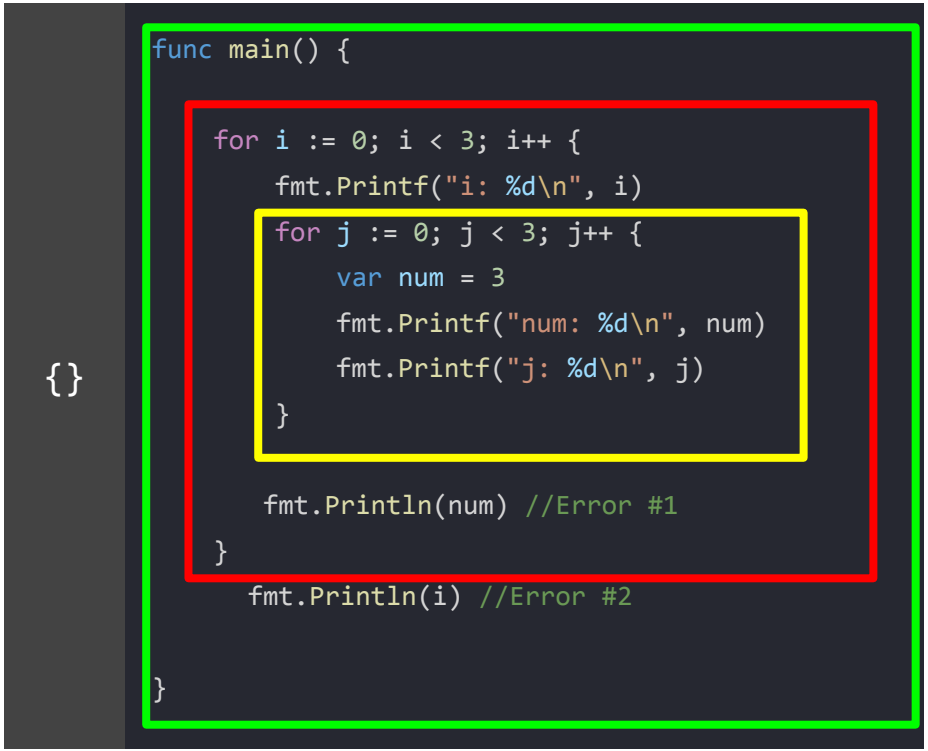
```
{}
func main() {
    fmt.Println(operacionAritmetica(6, 2, Suma))
    fmt.Println(operacionAritmetica(6, 2, Resta))
    fmt.Println(operacionAritmetica(6, 2, Multip))
    fmt.Println(operacionAritmetica(6, 2, Divis))
}
```

¿Qué es el Scope? ¿Scope?

Si, así es... el ‘scope’ o también conocido como ámbito de la función hace referencia a la porcion de codigo en la que la variable va a existir durante el programa.

¡A continuación veremos un ejemplo!

Ejemplo de Scope



- Scope verde: Puede ser utilizado por el scope rojo y amarillo
- Scope rojo y amarillo: No pueden ser utilizados por verde
- Scope rojo: Puede ser utilizado por el scope amarillo
- Scope amarillo: No puede ser utilizado por el scope rojo
- Errores: Intentar utilizar variables de un scope al que no tienen acceso
- Otro: El scope amarillo utiliza una variable que pertenece al scope rojo (*var i*)

Ellipsis

// ¿Qué es?

IT BOARDING

BOOTCAMP

// ¿Qué es Ellipsis?

“Go nos proporciona la notación de puntos suspensivos (Ellipsis). Que nos permite que nuestras funciones reciban una cantidad dinámica de parámetros.”

Notación de puntos suspensivos (Ellipsis)

Para utilizar esta notación, vamos a definir una función de la siguiente manera:

```
{}  func miFuncion(valores ...float64) float64
```

Al momento de llamar a esta función, podremos pasarle la cantidad de valores que queramos, siempre del mismo tipo de dato. Y nuestra función recibirá los parámetros como si fueran un array.

```
{}  miFuncion(2, 3, 2, 1, 2, 3, 4, 5, 6)
```

Notación de puntos suspensivos (Ellipsis)

Vamos a crear una función que reciba, mediante la notación de puntos suspensivos, un número variable de valores numéricos, y devolveremos la sumatoria de todos ellos.

```
{}
func suma(values ...float64) float64 {
    var resultado float64
    for _, value := range values {
        resultado += value
    }
    return resultado
}
```

Al llamar a esta función le podemos pasar todos los valores que queramos sumar.

```
{}
suma(2, 3, 2, 1, 2, 3, 4, 5, 6)
```

Notación de puntos suspensivos (Ellipsis)

También podemos pasar otros parámetros adicionales pero, en ese caso, el parámetro de notación de puntos suspensivos siempre tiene que estar al final

```
{}  func miFuncion(valor1 string, valor2 string, valores ...float64)
```

Notación de puntos suspensivos (Ellipsis)

Vamos a realizar un ejemplo más interesante que el anterior, crearemos una función a la cual le indicaremos la operación a realizar y todos los números a los que se le realizará dicha operación.

Por ejemplo: le indicaremos a la función que queremos realizar una suma y le pasaremos todos los valores que queramos sumar.

```
{}  operacionAritmetica(Suma, 2, 3, 2, 1, 2, 3, 4, 5, 6)
```

Notación de puntos suspensivos (Ellipsis)

Primero declararemos las constantes con las operaciones a realizar:

```
{}
const (
    Suma   = "+"
    Resta  = "-"
    Multip = "*"
    Divis  = "/"
)
```

Notación de puntos suspensivos (Ellipsis)

Luego creamos las funciones que realizarán las operaciones.

{}

```
func opSuma(valor1, valor2 float64) float64 {
   return valor1 + valor2
}

func opResta(valor1, valor2 float64) float64 {
   return valor1 - valor2
}

func opMultip(valor1, valor2 float64) float64 {
   return valor1 * valor2
}

func opDivis(valor1, valor2 float64) float64 {

   if valor2 == 0 {
       return 0
   }
   return valor1 / valor2
}
```

Funciones

También crearemos la función que se encargará de recibir la operación a realizar y los valores a los cuales se le aplicará la operación.
Por cada operación llamaremos a una función que reciba los valores y la función que vamos a ejecutar por ese operador.

{}

```
func operacionAritmetica(operador string, valores ...float64) float64 {
   switch operador {
   case Suma:
       return orquestadorOperaciones(valores, opSuma)
   case Resta:
       return orquestadorOperaciones(valores, opResta)
   case Multip:
       return orquestadorOperaciones(valores, opMultip)
   case Divis:
       return orquestadorOperaciones(valores, opDivis)
   }

   return 0
}
```

Notación de puntos suspensivos (Ellipsis)

Crearemos esa función que se encargará de orquestar las operaciones

{}

```
func orquestadorOperaciones(valores []float64, operacion func(value1, value2 float64)
float64) float64 {
    var resultado float64
    for i, valor := range valores {
        if i == 0 {
            resultado = valor
        } else {
            resultado = operacion(resultado, valor)
        }
    }

    return resultado
}
```

Notación de puntos suspensivos (Ellipsis)

Probamos nuestra aplicación pasandole la operación que queramos realizar.

```
{}
func main() {
    fmt.Println(operacionAritmetica(Suma, 2, 3, 2, 1, 2, 3, 4, 5, 6))
}
```

Multi retorno

// ¿Cómo se retornan varios valores?

IT BOARDING

BOOTCAMP

// Multi retorno

“Una de las características que tiene Go es que podemos crear funciones que retornan más de un valor.”

IT BOARDING

BOOTCAMP

Funciones de múltiples retornos

Para empezar tenemos que indicar los tipos de datos de los valores que retornarán, separados por coma y entre paréntesis.

```
{}  func miFuncion(valor1, valor2 float64) (float64, string, int, bool)
```

Funciones de múltiples retornos

Luego, vamos a generar una función que nos devuelva los cuatro resultados de las operaciones aritméticas: suma, resta, multiplicación, y división.

{}

```
func operaciones(valor1, valor2 float64) (float64, float64, float64, float64) {
   suma := valor1 + valor2
   resta := valor1 - valor2
   multip := valor1 * valor2
   var divis float64

   if valor2 != 0 {
       divis = valor1 / valor2
   }

   return suma, resta, multip, divis
}
```

Funciones de múltiples retornos

Al llamar a nuestra función, debemos recibir todos los valores que retorna.

{}

```
func main() {
    s, r, m, d := operaciones(6, 2)

    fmt.Println("Suma:\t\t", s)
    fmt.Println("Resta:\t\t", r)
    fmt.Println("Multiplicación:\t", m)
    fmt.Println("Division:\t", d)
}
```

Funciones de múltiples retornos

En Go el retorno de multivalores se utiliza por lo general cuando necesitamos retornar un valor y un error, y necesitemos validar si se produjo un error o no. Para ello vamos a realizar un ejemplo de una división y nos retorna error en caso que el divisor sea cero.

Utilizaremos el paquete errors que nos permite trabajar con la interfaz error.

```
{}
import (
    "errors"
)
```

Funciones de múltiples retornos

Implementamos nuestra función división y validamos si el divisor es cero, en caso que lo sea retornará un error, de lo contrario realizará la división.

```
{}
func division(dividendo,divisor float64) (float64, error) {

    if divisor == 0 {
        return 0, errors.New("El divisor no puede ser cero")
    }

    return dividendo/divisor, nil
}
```

Funciones de múltiples retornos

Ejecutamos nuestra función main y validamos si la operación fue realizada correctamente.

```
{}
func main() {
    res, err := division(2, 0)

    if err != nil {
        // Si hubo error
    } else {
        // Si termino correctamente
    }
}
```

Retorno de valores nombrados

// ¿Cómo lo hacemos?

IT BOARDING

BOOTCAMP

Retorno de valores nombrados

Podemos, también, retornar valores **nombrados**. Para esto, debemos definir en la función no solo el tipo de dato a retornar sino también el nombre de la variable.

```
{}
func operaciones(valor1, valor2 float64) (suma float64, resta
float64, multip float64, divis float64)
```

Retorno de valores nombrados

Dentro de la función, tenemos que almacenar el resultado de las operaciones en dichas variables y luego hacer un return.

De este modo, Go retornará los valores que guardamos en las variables que definimos en la función.

{}

```
func operaciones(valor1, valor2 float64) (suma float64, resta float64, multip float64,
divis float64) {
   suma = valor1 + valor2
   resta = valor1 - valor2
   multip = valor1 * valor2

   if valor2 != 0 {
       divis = valor1 / valor2
   }

   return
}
```

Retorno de funciones

// ¿Cómo una función retorna otra función?

IT BOARDING

BOOTCAMP

Retorno de función

Podemos implementar una función que devuelva otra función, para ello debemos indicarle los parámetros y los tipos de datos que retorne dicha función.
En este caso **“miFuncion”** nos devolverá otra función que recibe 2 parámetros y devuelve un valor en punto flotante.

```
{}  func miFuncion(valor string) func(valor1, valor2 float64) float64
```

Veamos un ejemplo de una función a la cual le indicaremos una operación y nos devolverá una función que realice la operación pasándole dos valores numéricos como parámetros.

Retorno de función

Crearemos una función para cada operación; Y cada una de ellas se encargará de una de las operaciones aritméticas: suma, resta, multiplicación, y división.

{}

```
func opSuma(valor1, valor2 float64) float64 {
   return valor1 + valor2
}

func opResta(valor1, valor2 float64) float64 {
   return valor1 - valor2
}

func opMultip(valor1, valor2 float64) float64 {
   return valor1 * valor2
}

func opDivis(valor1, valor2 float64) float64 {
   if valor2 == 0 {
       return 0
   }
   return valor1 / valor2
}
```

Retorno de función

Generamos una función que se encargue de orquestar las funciones que realizarán las operaciones.

{}

```
func operacionAritmetica(operador string) func(valor1, valor2 float64) float64 {
   switch operador {
   case “Suma”:
       return opSuma
   case “Resta”:
       return opResta
   case “Multip”:
       return opMultip
   case “Divis”:
       return opDivis
   }

   return nil
}
```

Retorno de función

Instanciamos la función indicando la operación a realizar.
Nos devolverá una función a la que le pasaremos los dos valores con los cuales queremos realizar la operación.

{}

```
func main() {
    oper := operacionAritmetica("Suma")
    r := oper(2, 5)
    fmt.Println(r)
}
```

Gracias.

IT BOARDING

BOOTCAMP