GO

BOOTCAMP

Clase en vivo

//Go Bases

IT BOARDING

BOOTCAMP

Objetivos de esta clase

- Conocer cómo escribir funciones.
- Entender las distintas formas de enviar parámetros a una función.
- Aplicar elipsis como parámetro de entrada
- Aplicar el concepto de multi retorno de objetos en una función

Índice

01 Repaso

02 Variadic Functions - Elipsis

03 Retorno de valores

04 Retorno de valores nombrados

05 Retorno de funciones

IT BOARDING
BOOTCAMP

1 | Repaso

IT BOARDING

BOOTCAMP

Parámetros de una Función #1

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 checkNumber(number int) {
    if number < 0 {
        fmt.Println("The number is negative")
    } else if number > 0 {
        fmt.Println("The number is positive")
    } else{
        fmt.Println("The number is zero")
    }
}
```

Parámetros de una Función #2

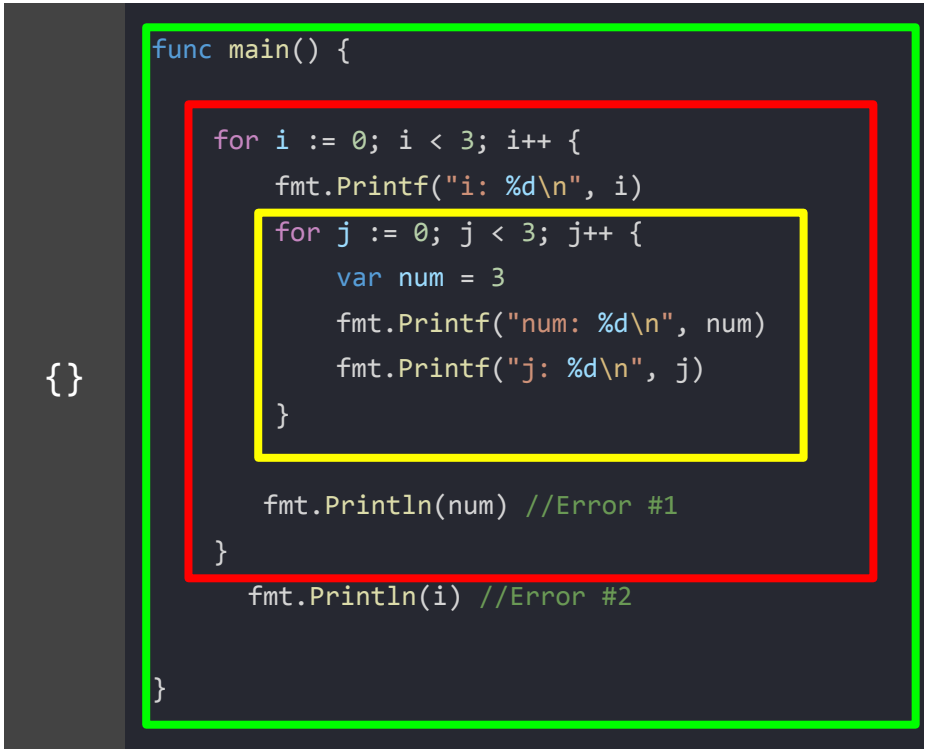
Luego llamaremos a nuestra función por cada variable generada, pasándole la variable como parámetro.

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

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

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

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*)

2 | Variadic Functions (Elipsis)

IT BOARDING

BOOTCAMP

Notación de puntos suspensivos

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

```
{}  func myFunction(values ...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.

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

Notación de puntos suspensivos

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

```
{}  func myFunction(value1 string, value2 string, others ...float64)
```

Notación de puntos suspensivos

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 addition(values ...float64) float64 {
    var result float64
    for _, value := range values {
        result += value
    }
    return result
}
```

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

{}

```
addition(2, 3, 2, 1, 2, 3, 4, 5, 6)
```

Notación de puntos suspensivos

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.

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

Notación de puntos suspensivos

Primero declararemos las constantes con las operaciones a realizar:

```
{}
const (
    Sum= "+"
    Sub  = "-"
    Mult = "*"
    Div  = "/"
)
```

Notación de puntos suspensivos

Luego creamos las funciones que realizarán las operaciones.

{}

```
func opSummatory(value1, value2 float64) float64 {
   return value1 + value2
}

func opSubtraction(value1, value2 float64) float64 {
   return value1 - value2
}

func opMultiplication(value1, value2 float64) float64 {
   return value1 * value2
}

func opDivision(value1, value2 float64) float64 {

   if value2 == 0 {
       return 0
   }
   return value1 / value2
}
```

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 calculate(operator string, values ...float64) float64 {
   switch operador {
   case Sum:
       return operationsOrchestrator(values, opSummatory)
   case Sub:
       return operationsOrchestrator(values, opSubtraction)
   case Mult:
       return operationsOrchestrator(values, opMultiplication)
   case Div:
       return operationsOrchestrator(values, opDivision)
   }

   return 0
}
```

Notación de puntos suspensivos

Crearemos esa función que se encargará de orquestar las operaciones

{}

```
func operationsOrchestrator(values []float64, operation func(value1, value2 float64)
float64) float64 {
    var result float64
    for i, value := range values {
        result = operation(result, value)
    }

    return result
}
```

Notación de puntos suspensivos

Probamos nuestra aplicación pasandole la operación que queramos realizar.

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

A codear...

3 | Multi Retorno

IT BOARDING

BOOTCAMP

// Multi retorno

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

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 myFunction(value1, value2 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 operations(value1, value2 float64) (float64, float64, float64, float64) {
    summatory := value1 + value2
    subtraction := value1 - value2
    multiplication := value1 * value2

    var division float64
    if value2 != 0 {
        division = value1 / value2
    }

    return summatory, subtraction, multiplication, division
}
```

Funciones de múltiples retornos

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

```
{}
func main() {
    sum, sub, mult, div := operations(6, 2)

    fmt.Println("Summatory:\t\t", sum)
    fmt.Println("Subtraction:\t\t", sub)
    fmt.Println("Multiplication:\t", mult)
    fmt.Println("Division:\t", div)
}
```

Funciones de múltiples retornos y errores

Veamos otro ejemplo de aplicación más convencional:

```
{}
func division(numerator, denominator float64) (float64, error) {
    if denominator == 0 {
        return 0, errors.New("can not divide by zero")
    }
    return numerator / denominator, nil
}
```

Cómo buena práctica, el uso de funciones multiretorno es implementado cuando durante el proceso se produce un error, el mismo es retornado cómo último parámetro.

Funciones de múltiples retornos y errores

De esta forma, podemos validar la existencia del error.

```
{}
func main() {
    num, den := 5.0, 0.0
    result, err := division(num, den)

    if err != nil{
        panic(err)
    }
    fmt.Println("Division result: ", result)
}
```

A codear...

4 Retorno de valores nombrados

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 operations(value1, value2 float64) (sum float64, sub float64,
mult float64, div 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 operations(value1, value2 float64) (sum float64, sub float64, mult float64, div
float64) {
   sum  = value1 + value2
   sub  = value1 - value2
   mult = value1 * value2

   if value2 != 0 {
       div = value1 / value2
   }

   return
}
```

A codear...

5 Retorno de Funciones

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 **“myFunction”** nos devolverá otra función que recibe dos parámetros y devuelve un valor en punto flotante.

```
{}  func myFunction(value string) func(value1, value2 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 opSummatory(value1, value2 float64) float64 {
   return value1 + value2
}

func opSubtraction(value1, value2 float64) float64 {
   return value1 - value2
}

func opMultiplication(value1, value2 float64) float64 {
   return value1 * value2
}

func opDivision(value1, value2 float64) float64 {
   if value2 == 0 {
       return 0
   }
   return value1 / value2
}
```

Retorno de función

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

{}

```
func calculate(operator string) func(value1, value2 float64) float64 {
   switch operator {
   case “Sum”:
       return opSummatory
   case “Sub”:
       return opSubtraction
   case “Mult”:
       return opMultiplication
   case “Div”:
       return opDivision
   }

   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 := calculate(“Sum”)
   r := oper(2, 5)
   fmt.Println(r)
}
```

A codear...

Conclusiones

Hoy aprendimos cómo utilizar las funciones en GO.

Aprendimos también sobre parámetros y elipsis y cerramos la clase viendo las distintas formas que tienen las funciones de devolvernos valores.

Gracias.

IT BOARDING

BOOTCAMP