INTRODUCCIÓN A LA CLASE

GO BASES

Objetivos de esta clase

- Comprender por qué es importante implementar un adecuado manejo de errores en nuestro código.
- Conocer qué es un error en GO y sus diferencias con otros lenguajes.
- Saber cómo crear errores personalizados y aplicarlo en la práctica.
- Conocer, comprender y utilizar correctamente las funciones del package "*errors*".
- Comprender la utilidad de los retornos de valor tipo "*error*" en las funciones que retornan más de un valor y realizar correctamente las validaciones de esos errores.

MANEJO DE ERRORES

GO BASES

// ¿Por qué es importante manejar errores?

Normalmente, el programa que desarrollamos puede arrojar uno o más errores. Manejar adecuadamente esos errores nos brinda al menos dos ventajas:

- **Ahorrar tiempo:** al poder encontrar más fácilmente qué es lo que falla y dónde.
- **Evitar que la ejecución finalice de modo o en tiempo no deseado:** al incorporar funciones que capturen los errores y permitan al programa continuar su ejecución.

Es conveniente (o necesario) que nuestro código siempre posea un adecuado manejo de errores.

Definiendo el tipo error y diferencias con otros lenguajes

IT BOARDING

BOOTCAMP

// ¿Qué es un error en GO?

En GO, “*error*” es un tipo interface. Es un tipo incorporado como cualquier otro. Por eso, no se requieren estructuras de control rígidas, ni especiales, para el manejo de errores.

Error type

Un “*error*” es un tipo “*interface*”. Una función de valor “*error*”, representa cualquier valor que pueda describirse a sí mismo como un “*string*”.

Esta es la declaración de la interface “*error*”:

```
type error interface {
    Error() string
}
```

Entonces, “*error*” es un tipo interface cuyo método “*Error()*” devuelve un “*string*”.

Ejemplo:

Una implementación típica del método “*Error()*” del tipo incluido “*error*” interface es el siguiente:

```
{}
func (e *MyError) Error() string {
    return "my error info"
}
```

Diferencias con otros lenguajes

A diferencia de otros lenguajes (como por ejemplo Java, C# y JavaScript), GO permite manejar los errores como cualquier otro tipo de dato del lenguaje. Sin requerir estructuras especiales ni rígidas. Es decir, como cualquier otra tarea que no se relacione exclusivamente con manejar errores.

GO	OTROS LENGUAJES
Funciones con retorno de un valor "*error*" para poder ser manejado mediante las mismas construcciones o estructuras de control propias de cualquier tarea.	Estructuras especiales y rígidas para el control de errores (Ej.: try-catch-finally)
Funciones propias del lenguaje para el manejo de errores.	Excepciones

CREANDO ERRORES PERSONALIZADOS

GO BASES

// ¿Cómo crear y personalizar nuestros errores?

"GO nos brinda algunas funciones principales para crear y personalizar nuestros errores:

- *Error()*
- *errors.New()*
- *fmt.Errorf()*

IT BOARDING
BOOTCAMP

Error()

//Método de interface error

IT BOARDING

BOOTCAMP

Error()

Para dotar a un error de información más compleja, y ampliar la funcionalidad, podemos implementar el tipo de interfaz de biblioteca estándar: "*error*".

```
type error interface {
   Error() string
   }
```

El paquete "builtin" de GO, define "*error*" como un tipo interface que implementa el método "*Error()*" para devolver un mensaje de error de tipo "*string*". Al implementarlo, podemos crear un error propio, personalizado con la información que necesitemos.

Ejemplo implementando "*Error()*": #1

Veamos un ejemplo paso a paso de cómo implementar el método "*Error()*":
1.- Primero lo primero: definir nuestro package main e importar los packages "*fmt*" y "*os*".

```
{}
package main

import (
    "fmt"
    "os"
)
```

Ejemplo implementando Error(): #2

2.- Luego definimos un *struct* (al que nombraremos *myCustomError*) para crear y personalizar nuestro error, y hacemos que implemente la interface ("*Error()*").

{}

```
// sólo se requiere crear un tipo que implemente el método Error()
type myCustomError struct {
   status int
   msg string
}
//hacemos que nuestro tipo struct implemente el método Error()
func (e *myCustomError) Error() string {
   return fmt.Sprintf("%d - %v", e.status, e.msg)
}
```

Ejemplo implementando "*Error()*": #3

3.- Ahora creamos una función para llenar nuestro custom error con los datos que precisemos para describirlo.

```
{}
```

```
func myCustomErrorTest (status int) (int, error) {
   if status >= 300 {
      return 400, &myCustomError {
          status : status,
          msg : "algo salió mal",
      }
   }
   return 200, nil
}
```

Ejemplo implementando "*Error()*": #4

4.- Finalmente generamos nuestra función "*main*" para practicar el manejo de nuestro custom error.

{}

```
func main () {
   status, err := myCustomErrorTest( 300 ) //llamamos a nuestra func
   if err != nil { //hacemos una validación del valor de err
      fmt.Println(err) //si err no es nil, imprimimos el error y...
      os.Exit(1) //utilizamos este método para salir del programa
   }
   fmt.Printf("Status %d, Funciona!", status)
}
```

Para tener en cuenta...

Recuerda que este es sólo un ejemplo ilustrativo de cómo podemos implementar el método "*Error()*" para crear nuestros propios errores y dotarlos de los datos que precisemos.

En la práctica diaria, puedes crear tus propios tipos con los atributos que consideres necesarios, y hacer que esos tipos implementen el método "*Error()*".

Así, nuestros errores pueden contener información más descriptiva de lo que sucedió.

errors.New()

//Función de pkg errors

IT BOARDING

BOOTCAMP

"errors.New()"

La función "*New()*" del paquete "*errors*" nos permite crear un nuevo error básico con un mensaje (*string*) dado.

errors.New("string") recibe un único argumento: un mensaje de error de tipo *string* que puedes personalizar para informar cuál fue el problema que generó el error.

Package "*errors*"

Importado dentro de nuestro package "*main*"

Argumento tipo string

Un mensaje dado de error para describir lo que sucedió.

Función "*New()*"

Pertenece al package "*errors*" importado.

Ejemplo implementando “*errors.New()*” #1

Veamos un ejemplo paso a paso de cómo implementar la función “*New()*” del paquete “*errors*”:
1.- Primero lo primero: definir nuestro package *main* e importar los packages “*fmt*” y “*errors*”.

```
{}
package main

import (
    “fmt”
    “errors”
)
```

Ejemplo implementando “*errors.New()*” #2

2.- Declaramos nuestra función “*main()*”. Definimos una variable llamada “*statusCode*” con un valor de tipo “*int*”. Luego realizamos una validación para comprobar si “*statusCode*” es mayor a 399. En cuyo caso utilizamos “*errors.New()*” para generar un mensaje de error.

{}

```
func main() {
   statusCode := 404;
   if statusCode > 399 {
      fmt.Println(errors.New("La petición ha fallado."))
      return
   }
   fmt.Println("El programa finalizó correctamente.")
}
```

fmt.Errorf()

//Función de pkg fmt

IT BOARDING

BOOTCAMP

"fmt.Errorf()"

La función "*Errorf()*" del paquete "*fmt*" nos permite crear un mensaje de error de forma dinámica.

Como argumentos recibe, primero, un *string* que contiene el mensaje de error con valores de marcadores de posición, como *"%s"* para cadenas y *"%d"* para enteros. Luego, recibe los argumentos que se interpolan en esos marcadores de posición en orden.

Package "*fmt*"

Importado dentro de nuestro package "*main*"

Argumento tipo string

Un mensaje de error creado en forma dinámica que contiene un marcador de posición

Argumento tipo int

Se interpola en dónde indica el marcador de posición

Función "*Errorf()*"

Pertenece al package "fmt" importado.

Ejemplo implementando “*fmt.Errorf()*” #1

Veamos un ejemplo paso a paso de cómo implementar la función “*Errorf()*” del paquete “*fmt*”:
1.- Primero lo primero: definir nuestro package *main* e importar los paquetes “*fmt*” y “*time*”.

```
{}
package main

import (
    “fmt”
    “time”
)
```

Ejemplo implementando “*fmt.Errorf()*” #2

2.- Declaramos nuestra función “*main()*”.
Creamos una variable a la que llamamos “*err*”
Implementamos “*fmt.Errorf()*” para crear un mensaje de error interpolando, en su primer argumento, un marcador de posición “*%v*” que indica que será reemplazado por el dato devuelto por el método “*Now()*” del paquete “*time*”, que pasamos como segundo argumento.
Luego imprimimos “*err*”.

{}

```
func main() {
    err := fmt.Errorf(“momento del error: %v”, time.Now())
    fmt.Println(“error ocurrido: ”, err)
}
```

Recuerda...

Por convención de estilos, tus mensajes de error no deben contener mayúsculas (a menos que comiencen con nombres propios o siglas).

PACKAGE ERRORS

GO BASES

// ¿Para qué sirve el paquete “*errors*”?

“El package “*errors*” implementa funciones para manipular errores: *New()*, *Is()*, *As()* y *Unwrap()*”.

IT BOARDING

BOOTCAMP

Aclaración:

La función “*New()*” crea errores que sólo contienen un texto de mensaje tipo string. De modo que vamos a concentrarnos en las restantes funciones del package “*errors*”: *“Is()”*, *“As()”* y “*Unwrap()”*.

As()

//PKG ERRORS

IT BOARDING

BOOTCAMP

"As()"

La función "*As()*" comprueba si un error es de un tipo específico. Si encuentra coincidencia devuelve "*true*", caso contrario devuelve "*false*".

La función se comporta como una aserción o afirmación de tipo.

```
{}  func As(err error, target interface{}) bool
```

Un error coincide con el objetivo, o target, si el valor concreto del error es asignable al valor señalado por el objetivo.

Ejemplo implementando “*As()*”: #1

Veamos un ejemplo paso a paso de cómo implementar la función “*As()*”:
1.- Primero lo primero: definir nuestro package *main* e importar los packages “*fmt*” y “*errors*”.

```
{}
package main

import (
    "fmt"
    "errors"
)
```

Ejemplo implementando "*As()*": #2

2.- Luego definimos un *struct* (al que nombraremos "*myError*") y hacemos que implemente el método "*Error()*" de la interface "*error*".

{}

```
// definimos un type struct
type myError struct {
   msg string
   x string
}
//hacemos que nuestro type struct implemente el método Error()
func (e *myError) Error() string {
   return fmt.Sprintf("ha ocurrido un error: %s, %s", e.msg, e.x)
}
```

Ejemplo implementando "*As()*": #3

4.- Finalmente generamos nuestra función "*main()*".
Dentro definiremos una variable llamada "e" que será de tipo "*myError*" y asignaremos un valor a sus atributos de tipo *string*.
Y declararemos otra variable que apuntaremos a nuestro tipo "*myError*".

{}

```
func main() {
   e := &myError{"nuevo error", "404"}

   var err *myError

   isMyError := errors.As(e, &err)// compara los errores

   fmt.Println(isMyError)//imprime true porque los errores coinciden
}
```

Para tener en cuenta...

"*errors.As()*" sólo funciona con una interface o con un tipo que implementa *error* (no con *error* en forma directa). El código siguiente no funcionará:

```
var err *error
 if ok := errors.As( e,  &err ); ok {
  fmt.Println( err )
}
```

ls()

//PKG ERRORS

IT BOARDING

BOOTCAMP

“errors.Is()”

La función “*Is()*” compara un error con un valor. Si encuentra coincidencia devuelve “*true*”, caso contrario, devuelve “*false*”.

Se comporta como una comparación con un error centinela.

```
{}  func Is(err, target error) bool
```

Un error coincide con un objetivo si es igual a ese objetivo.

Ejemplo implementando “*errors.Is()*” #1

Veamos un ejemplo paso a paso de cómo implementar la función “*Is()*” del paquete “*errors*”:
1.- Primero lo primero: definir nuestro package *main* e importar los packages “*fmt*” y “*errors*”.

```
{}
package main

import (
    “fmt”
    “errors”
)
```

Ejemplo implementando “*errors.Is()*” #2

2.- Creamos un nuevo error con la función “*New()*” y lo asignamos a una variable que definimos como “*err1*”. Luego declaramos una función “*x*” que retorna “*err1*”.

```
var err1 = errors.New("error número 1")

func x() error {
   return fmt.Errorf("información extra del error: %w", err1)
}
```

Ejemplo implementando “*errors.Is()*” #3

3.- Declaramos nuestra función “*main()*”.
Definimos una variable llamada “e”, a la que le asignamos el retorno de nuestra función “*x*”.
Implementamos la función “*Is()*” del paquete “*errors*” para comparar “*e*” con “*err1*” y asignamos su retorno a la variable “*coincidence*”.
Finalmente imprimimos el valor de “*coincidence*”.

```
{}
func main() {
    e := x()
    coincidence := errors.Is(e, err1)
    fmt.Println(coincidence) //imprime true
}
```

Unwrap()

//PKG ERRORS

IT BOARDING

BOOTCAMP

“errors.Unwrap()”

Un error que contiene a otro puede implementar “*Unwrap()*”. Esta función devuelve el error subyacente. Si *“e1.Unwrap()”* regresa “*e2*”, entonces, decimos que “*e1*” envuelve a “*e2*” y que se puede desenvolver “*e1*” para obtener “*e2*”.

```
{}  func Unwrap(err error) error
```

Si tenemos un error llamado “*e1*” y otro llamado “*e2*”, y “*e1*” envuelve a “*e2*”, al ejecutar la función “*errors.Unwrap(e1)*” devolverá “*e2*”, caso contrario devolverá “*nil*”.

Ejemplo implementando “*errors.Unwrap()*” #1

Veamos un ejemplo paso a paso de cómo implementar la función “*Unwrap()*” del paquete “*errors*”:
1.- Primero lo primero: definir nuestro package *main* e importar los paquetes “*fmt*” y “*errors*”.

```
{}
package main

import (
    “fmt”
    “errors”
)
```

Ejemplo implementando “*errors.Unwrap()*” #2

2.- Creamos una estructura “*errorTwo*” que tiene un método “*Error()*”, por lo que implementa la interfaz de “*error*”.

```
{}
type errorTwo struct{}

func (e errorTwo) Error() string {
    return "error two happened"
}
```

Ejemplo implementando “*errors.Unwrap()*” #3

3.- Luego declaramos nuestra función “*main*” y dentro creamos una instancia de la estructura “*errorTwo*” llamada “*e2*”.
Envolvemos esa instancia “*e2*” en otro error “*e1*”.
Finalmente, utilizando “*errors.Unwrap()*”, desenvolvemos “*e1*” para obtener “*e2*”.
Pero si intentamos desenvolver “*e2*” obtendremos “*nil*”.

{}

```
func main() {
    e2 := errorTwo{}
    e1 := fmt.Errorf("e2: %w", e2)
    fmt.Println(errors.Unwrap(e1))//imprime e2
    fmt.Println(errors.Unwrap(e2))//imprime nil
}
```

RETORNO DE ERRORES

GO BASES

// ¿Cúal es el rol de las funciones multi-retorno en el manejo de errores?

“GO permite que las funciones devuelvan más de un valor. Entre los valores retornados, puede haber uno de tipo error”.

IT BOARDING
BOOTCAMP

Para tener en cuenta:

Las funciones multi-retorno de GO nos permiten retornar, entre otros valores, un error.
Este error puede ser uno que nosotros mismos hayamos creado, o bien, alguno de los errores que dichas funciones retornan por defecto.

A diferencia de otros lenguajes que usan excepciones para el manejo de muchos errores, en Go es idiomático usar valores de retorno que indican errores siempre que sea posible.

Buenas prácticas:

En la práctica, usualmente, no creamos un error para utilizarlo de forma directa. Por lo general, lo creamos para utilizarlo como retorno de una función.

Por convención, cuando utilices funciones que retornan más de un valor, el error debe ser devuelto en última posición.

SINTAXIS

//Funciones con retorno “*error*”

IT BOARDING

BOOTCAMP

Sintaxis

Para crear una función que devuelva más de un valor, incluyendo el retorno de un error, enumeramos los tipos de cada valor devuelto dentro de paréntesis en la firma de la función. Por ejemplo:

```
{}
func FuncName() (string, error) {
   //return string, error
}
```

Recuerda que el “error” debe ser el último tipo retornado por nuestra función.

Ejemplo #1

Veamos un ejemplo paso a paso de cómo utilizar una función que retorna un tipo error junto con otro valor:
1.- Primero lo primero, definimos nuestro package “*main*” e importamos los packages “*fmt*” y “*errors*”.

```
{}
package main

import (
    "fmt"
    "errors"
)
```

Ejemplo #2

2.- Luego declaramos nuestra función “*SayHello*” que recibirá como argumento un “*string*” y retornará dos valores: un “*string*” y un “*error*”.

{}

```
func SayHello(name string) (string, error) {
     if name == "" {
          return "", errors.New("no name provided")
     }
     return fmt.Sprintf("Hola %s ", name), nil
}
```

Ejemplo #3

4.- Finalmente generamos nuestra función “*main()*”. Dentro llamamos a nuestra función “*SayHello*”:

```
{}
func main() {
    name := “Meli”
    greeting, err := SayHello(name)
    if err != nil {
        fmt.Println(“error: ”, err)
}
    fmt.Println(greeting)
}
```

Para tener en cuenta...

Al devolver un error de una función con varios valores de retorno, el código idiomático GO también establecerá un “valor cero” para cada valor “non-error”. Los valores cero son, por ejemplo, una cadena vacía para *strings*, “0” para enteros, una estructura vacía para tipos de estructura y “*nil*” para tipos de punteros e interfaces, entre otros.

Manejo del error retornado

// Retornos “*error*”

IT BOARDING

BOOTCAMP

Manejando el retorno “*error*”

Cuando una función devuelve muchos valores, GO requiere que asignemos una variable a cada uno de ellos. Como vimos en nuestro ejemplo anterior, lo hacemos al proporcionar nombres para los dos valores que devuelve la función “*SayHello*”.

```
{}  greeting, err := SayHello(name)
```

El primer valor devuelto se asignará a la variable “*greeting*”, y el segundo valor (el error) se asignará a la variable “*err*”.

A veces, solo nos interesa el valor de error. En tal caso, podemos descartar cualquier valor no deseado que devuelva una función utilizando el identificador blank “_”.

```
{}  _, err := SayHello(name)
```

Validaciones

// Retornos “*error*”

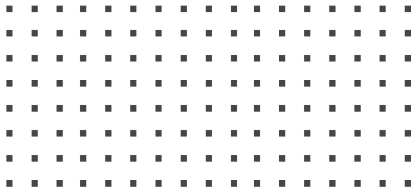
IT BOARDING

BOOTCAMP

Validando el retorno “*error*” #1

Al retornar un tipo *error* junto con otros valores en una función, es posible que dicho error efectivamente ocurra y, por ende, contenga algo. O bien, puede que no ocurra ningún problema y nuestro error se retorne como “*nil*”.

En cualquier caso, es necesario realizar una validación de lo ocurrido, antes de continuar con nuestro código.



Validando el retorno “*error*” #2

Retomando el ejemplo anterior, dentro de la función “*main()*”, utilizamos un condicional “*if*” para verificar si nuestro error ocurrió o retornó con valor cero (“*nil*”).

{}

```
func main() {
   name := “Meli”
   greeting, err := SayHello(name)
   //validamos si hubo error
   if err != nil {
      fmt.Println(“no se puede saludar ”, err)
   }
   //si no hubo error, continuamos con nuestra ejecución
   fmt.Println(greeting)
}
```

Validando el retorno “*error*” #3

Es recomendable contar con casos lógicos de manipulación en los que el error no esté presente (“*nil*”) y casos en los que sí lo esté.

La construcción “*if err ! = nil*” que se muestra en el último ejemplo es el caballo de batalla de la manipulación de errores en GO.
Cuando una función pueda producir un error y retornarlo, es importante utilizar una instrucción “*if*” para corroborar su presencia. De esta manera, el código idiomático GO tiene, naturalmente, su lógica “*happy path*” en el primer nivel de indentación, y toda la lógica “*sad path*”, en el segundo.

Gracias.

IT BOARDING

BOOTCAMP