# INTRODUCCIÓN A LA CLASE

GO BASES

# Objetivos de esta clase

- Conocer y comprender qué es un “*panic*” y su estructura.
- Crear “*panics*”.
- Conocer casos típicos generadores de “*panic*”.
- Conocer “*defer*” y “*recover*” y comprender su utilidad y funcionamiento.
- Implementar “*defer*” y “*recover*” para un adecuado manejo de los “*panic*”.
- Adquirir un primer conocimiento introductorio sobre el package “*context*”.

# PANIC

GO BASES

## // ¿Qué es *panic*?

"***Panic*** es una **interrupción de la ejecución** de nuestro programa, con la **indicación de que algo salió mal** de forma inesperada".

# // ¿Para qué son útiles los *panic*?

1.- En general, se usan para fallar rápidamente en errores que no deberían ocurrir durante un funcionamiento normal de un programa.

2.- Debido a que entre los *panics* se incluyen detalles que son útiles para resolver un problema, los desarrolladores normalmente utilizan los *panics* como una indicación de que cometieron un error durante el desarrollo de un programa y como un mapa para encontrar rápidamente la causa de ese error.

3.- Un uso común del pánico es abortar la ejecución de nuestro programa si una función devuelve un valor de error que, por alguna razón, no vamos a manejar.

# ¿Cuándo ocurre un *panic*?

Ciertas operaciones en GO producen *panic* e interrumpen la ejecución del programa de forma automática. Algunas operaciones comunes que lo generan, a modo de ejemplo, no taxativo, son:

- Exceder la capacidad de indexación de una matriz.
- Invocar métodos en punteros nulos.
- Intentar trabajar con canales cerrados.

La mayoría de estas situaciones se deben a equivocaciones que se cometen durante la programación y que el compilador no puede detectar en tiempo de compilación.

# Tener en cuenta:



**Es recomendable realizar un adecuado manejo de los *panic* para evitar que nuestro programa colapse abruptamente.**

Los *panic*:
- terminan la ejecución del programa.
- llaman a las funciones diferidas por la función en pánico.

# PANIC()

## //FUNCIÓN INCORPORADA

IT BOARDING

BOOTCAMP

# Creando nuestros propios *panic*

Además de las operaciones que generan *panic* por defecto, también podemos generar y personalizar nuestros propios *panic*. Para eso GO nos brinda la función incorporada "*panic()*".

Esta función recibe como argumento una *interface*, lo cual podemos utilizar para pasar la información que nos ayude a comprender el *panic* cuando se produzca.

De este modo podemos, por ejemplo, brindar un mensaje con una breve indicación de la causa del *panic*.

```
{}  panic("causa del panic")
```

# Creando nuestros *panic*. Ejemplo #1:

Un uso común de la función “*panic()*”, es abortar si una función devuelve un valor de error que, por algún motivo, no vamos a manejar (o porque aún no sabemos cómo hacerlo, o porque no tenemos interés en hacerlo, etc.).

Veamos un ejemplo, para el cual debemos, previamente, haber definido nuestro pkg “*main*” e importado los pkgs “*fmt*” y “*os*”:

{}

```
func main() {
    fmt.Println("Iniciando... ")
    _, err := os.Open("no-file.txt")
    if err != nil {
        panic(err)
    }
    fmt.Println("Fin")
}
```

# Creando nuestros *panic*. Ejemplo #2:

Al ejecutar nuestro programa obtendremos por consola una salida similar a esta:

```
Iniciando...
panic: open no-file.txt: no such file or directory

goroutine 1 [running]:
main.main()
   /tmp/sandbox920660534/prog.go:12 +0x130

Program exited: status 2.
```

# Creando nuestros *panic*. Ejemplo #3:

Al chequear la salida por consola vemos que:

1. Nuestra función "*main*" comenzó a ejecutarse al imprimir el texto "*Iniciando...*"
2. Luego obtuvimos un "*panic*" seguido de un mensaje que nos indica que no se encontró el archivo especificado que intentamos abrir.
3. Se imprimió una ruta de seguimiento de ejecución.
4. Nuestro programa abortó de forma abrupta con un "*status 2*" (por ello la ejecución no continuó y no se imprimió "*Fin*").

# Estructura de un *panic*. Ejemplo #4:

Normalmente, un *panic* se estructura de ese modo:

```
//la palabra “panic” + la acción intentada + un breve detalle de lo sucedido
panic: open no-file.txt: no such file or directory
//el stack trace en ejecución con un mapa de seguimiento que indica dónde se produjo el panic
goroutine 1 [running]:
main.main()
   /tmp/sandbox920660534/prog.go:12 +0x130
//el status de salida del programa
Program exited: status 2.
```

# CASOS DE PANIC

//EJEMPLOS

IT BOARDING

BOOTCAMP

# Index Out of Bounds Panics #1

Este caso se da cuando intentamos acceder a un índice más allá de la longitud de un slice o la capacidad de un array.

En este caso, GO producirá un *panic* en tiempo de ejecución. Definamos nuestro pkg "*main*" e importemos el pkg "*fmt*" y veamos un ejemplo:

{}

```
func main() {
    animals := []string{
        "vaca",
        "perro",
        "halcon",
    }
    fmt.Println("solo vuela el: ", animals[len(animals)])
}
```

# Index Out of Bounds Panics #2

Al ejecutar nuestro programa obtendremos por consola una salida similar a esta:

```
panic: runtime error: index out of range [3] with length 3

goroutine 1 [running]:
main.main()
   /tmp/sandbox009542944/prog.go:13 +0x1b

Program exited: status 2.
```

# Index Out of Bounds Panics #3

Al chequear la salida por consola vemos que:

1. Se produjo un *panic* al intentar acceder a un índice que excede la longitud del slice creado.
2. GO finalizó en tiempo de ejecución.
3. Las instrucciones posteriores al *panic* no se ejecutaron. Esto dado que el *panic*, al producirse, llama a las funciones diferidas por la función en pánico y aborta nuestro programa.

# Receptores nulos #1

GO da la posibilidad de trabajar con punteros para referenciar a una instancia específica de algún tipo existente en la memoria del equipo en tiempo de ejecución. Los punteros pueden asumir el valor "*nil*" para indicar que no apuntan a nada. Cuando intentamos invocar métodos en un puntero que tenga el valor "*nil*", se producirá *panic*. Veamos un ejemplo:

{}

```
type Dog struct {
   Name string
}
func (s *Dog) WoofWoof() {
   fmt.Println(s.Name, " hace woof woof")
}
func main() {
   s := &Dog{"Sammy"}
   s = nil
   s.WoofWoof()
}
```

# Receptores nulos #2

Al ejecutar nuestro programa obtendremos por consola una salida similar a esta:

```
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x0 pc=0x497783]

goroutine 1 [running]:
main.(*Dog).WoofWoof(...)
   /tmp/sandbox602314289/prog.go:12
main.main()
   /tmp/sandbox602314289/prog.go:18 +0x23

Program exited: status 2.
```

# Receptores nulos #3

Al chequear la salida por consola vemos que:

1. Se produjo un *panic* al intentar acceder a una dirección no válida de memoria o apuntar a un receptor nulo.
2. El *panic* se produce en tiempo de ejecución y aborta la ejecución del programa con *status 2*.

# Recuerda:

Los *panics* generados a partir de los punteros "*nil*" y los accesos fuera de límites, son dos casos típicos de *panics* que se generan en tiempo de ejecución.

# DEFER & RECOVER

GO BASES

## // ¿Cómo manejar un panic?

"Con las sentencias incorporadas "*defer*" y "*recover*" podemos controlar los efectos de un *panic* y evitar que nuestro programa finalice de modo no deseado.

# Recuerda:

*"defer"* y *"recover"* son funciones incorporadas al lenguaje, específicamente diseñadas para evitar o controlar la naturaleza destructiva de un *panic*.

Si bien se presentan como funciones independientes, se requiere un uso complementario entre ambas para lograr resultados de mejor performance.

# DEFER

## //FUNCIONES DIFERIDAS

IT BOARDING

BOOTCAMP

# // ¿Qué es “*defer*”?

Es una sentencia incorporada en GO, que nos permite diferir la ejecución de ciertas funciones y “asegurar” que sean ejecutadas antes de la finalización de la ejecución de un programa.

Puede decirse que es el similar a “*ensure*” o “*finally*” utilizado en otros lenguajes de programación.

# // ¿Para qué es útil y cómo se utiliza?

Es de gran utilidad para asegurarnos de limpiar recursos durante la ejecución de nuestro programa, incluso ante la ocurrencia de un *panic*.

Se utiliza como mecanismo de seguridad para brindar protección contra los cortes de ejecución y salidas abruptas que generan los *panics*.

Las funciones se difieren invocándolas de la forma habitual y añadiendo luego un prefijo a toda la instrucción con la palabra clave “***defer***”.

# *Defer*. Ejemplo #1

Definamos nuestro pkg “*main*”, importemos el pkg “*fmt*” y probemos un ejemplo de función diferida usando “*defer*”:

{}

```
func main() {
   //aplicamos “defer” a la invocación de una función anónima
       defer func() {
           fmt.Println("Esta función se ejecuta a pesar de producirse panic")
       }()
   //creamos un panic con un mensaje de que se produjo
       panic("se produjo panic!!!")
   }
```

# *Defer*. Ejemplo #2

Al ejecutar nuestro programa obtendremos por consola una salida similar a esta:

```
Esta función se ejecuta a pesar de producirse panic
panic: se produjo panic!!!

goroutine 1 [running]:
main.main()
   /tmp/sandbox501206329/prog.go:13 +0x5b

Program exited: status 2.
```

# *Defer*. Ejemplo #3

Al chequear la salida por consola vemos que:

1. La función anónima diferida se ejecuta e imprime su mensaje.
2. Se encuentra el *panic* que se generó en nuestra función “*main*”.
3. Se produjo la salida del programa con *status 2*.

# Tener en cuenta:

Si bien las funciones diferidas se ejecutan, incluso, ante la producción de *panics*, NO se ejecutan ante la ejecución de la función "*log.Fatal()*".

En caso de haberse diferido varias funciones, al ser llamadas, se ejecutarán en orden, iniciando desde la última función diferida hacia la primera de ellas.

# RECOVER

## //RECUPERANDO PANICS

IT BOARDING

BOOTCAMP

# // ¿Qué es “*recover*” y para qué es útil?

Es una función incorporada que permite interceptar un *panic* y evitar que este termine con la ejecución del programa en forma inesperada o no deseada.

Al ser parte del paquete incluido en GO, puede invocarse sin importar paquetes adicionales.

# // ¿Cómo se utiliza?

Lo correcto es utilizar la función incorporada “*recover*” dentro de una declaración “*defer*”. De este modo, al producirse un *panic*, esa función diferida recuperará el control de la rutina en pánico, y el valor establecido en “*panic*”, y evitaremos que nuestro programa termine de forma no deseada.

Si utilizáramos “*recover*” fuera de una declaración “*defer*”, la producción de un panic terminaría con la ejecución del programa antes de que “*recover*” pueda recuperar el valor de “*panic*”. Es decir, en este caso, “*recover*” retornaría “*nil*” y no evitaría la finalización abrupta de la ejecución.

# *Recover*. Ejemplo #1

Definamos nuestro pkg “*main*”, importemos el pkg “*fmt*” y probemos un ejemplo de “*recover*”.

Para esto vamos a declarar una función llamada “*isPair()*” que recibirá como argumento un número entero y analizará si es par o no.

Caso de ser impar, producirá *panic* y llamará a la función anónima diferida que contiene la función “*recover*”.

El *panic* será controlado y su valor recuperado por “*recover*” y asignado a la variable “*err*”.

Al ser “*err*” distinto de “*nil*”, se imprimirá por consola el valor recuperado del *panic* producido. La función diferida finalizará su ejecución y el programa continuará la suya.

# *Recover*. Ejemplo #2

{}

```
func isPair(num int) {
    defer func() {
        err := recover()

        if err != nil {
            fmt.Println(err)
        }

    }()

    if (num % 2) != 0 {
        panic("no es un número par")
    }

    fmt.Println(num, " es un número par!")
}
```

# *Recover*. Ejemplo #3

En nuestra función “*main()*”, llamaremos a la función “*isPair()*” y le pasaremos como argumento un número impar para que genere *panic*.

Veremos que el *panic* generado en la función “*isPair()*” es controlado por “*recover*”, y no se aborta la ejecución de nuestra “*main()*”.

{}

```
func main() {
    num := 3

    isPair(num)

    fmt.Println("Ejecución completada!")

}
```

# *Recover*. Ejemplo #4

Al ejecutar nuestro programa obtendremos por consola una salida similar a esta:

```
no es un número par
Ejecución completada!

Program exited.
```

Observa como el valor de *panic* fue recuperado por "*recover*" y el programa completó su ejecución hasta el final.

# Tener en cuenta:

Es muy importante verificar que el argumento para la función *builtin panic* no sea "*nil*".

En caso contrario, "*recover*" recuperará ese valor "*nil*", pasará nuestra estructura de validación "*if err != nil*" y un *panic* producido podría pasar inadvertido, generando efectos no deseados en nuestro programa.

Puedes hacer la prueba estableciendo el valor del argumento de panic como "*nil*"

# PACKAGE CONTEXT

GO BASES

## // ¿Qué es el package Context?

El package *context* es una de las librerías más útiles que provee GO.
Sirve para definir un contexto que puede ser pasado a través del código, para que las diferentes partes interesadas reaccionen a él.

# Context type

El tipo *context* es una interfaz que define los métodos “*Deadline()*”, “*Done()*”, “*Err()*”, y “*Value()*”.

Estos métodos se usan según los distintos tipos de *context* que creamos.

{}

```
type Context interface {
   Deadline() (deadline time.Time, ok bool)
      Done() <- chan struct{}
      Err() error
      Value(key interface{}) interface{}
   }
```

# Convenciones

Es convención al momento de definir una función que reciba un contexto, que el mismo sea el primer argumento y sea nombrado como "*ctx*".

```
{}
func funcionConContexto(ctx context.Context, ...args) {
    ...
}
```



**Esto es una buena práctica en go**

# ¿Cuál es la utilidad del pkg *context*?

Algunas veces, puede que queramos llamar a una función que puede tardar mucho en retornar. Sería bueno tener alguna herramienta para cancelar dicha función en caso de que tarde demasiado. Por suerte existe el paquete *Context*.

Este paquete nos permite crear un contexto, que puede pasarse a distintas funciones y a sus funciones “hijas” subsiguientes.

Cuando se cancela el contexto desde alguna parte del código, esta señal de cancelación se propaga a todas las funciones que reciben el contexto como parámetro.

# Background

//Func

IT BOARDING

BOOTCAMP

# *.Background()*

La función “*Background( )*”, que define el package *context*, nos permite crear un contexto vacío. Generalmente, es usada para crear el contexto base, sobre el cual después podemos expandir según sea necesario. Veamos un ejemplo:

{}

```go
package main

import (
    "context"
    "fmt"
)

func main() {
    ctx := context.Background()

    fmt.Println(ctx)
}
```

# WithValue

//Func

IT BOARDING

BOOTCAMP

# *.WithValue()*

La función “*WithValue(ctx context.Context, key, val interface{ } )*” toma como argumentos un contexto padre (por ejemplo el creado con “*.Background( )”* ) y un par “*clave -> valor”*.

Retorna un nuevo contexto.

Este contexto sirve para pasar este par “*clave -> valor*” a través de las distintas funciones que tomen este contexto como argumento.

# *.WithValue()*. Ejemplo:

{}

```go
package main

import (
    "context"
    "fmt"
)

func main() {
    ctx := context.Background()
    ctx = context.WithValue(ctx, "saludo", "hola digital house!!")
    saludoWrapper(ctx)
}

func saludoWrapper(ctx context.Context) {
    saludo(ctx)
}

func saludo(ctx context.Context) {
    fmt.Println(ctx.Value("saludo"))
}
```

# WithDeadline

//Func

IT BOARDING

BOOTCAMP

# *.WithDeadline()*

La función “*WithDeadline(ctx context.Context, d time.Time)*”, toma como argumentos: un contexto padre (por ejemplo, el creado con “*.Background( )”* ) y una fecha.

Retorna un nuevo contexto, y una función que sirve para cancelar el contexto manualmente.

Cuando la fecha actual sea menor a la fecha especificada en la función, el contexto se cancelará automáticamente.

# *.WithDeadline()*

En el siguiente ejemplo, crearemos una fecha límite de 5 segundos posterior a la ejecución del programa, y la pasaremos como argumento a “*context.WithDeadline( )*”.

Cuando pasen esos 5 segundos, el canal “*ctx.Done( )*” recibirá un mensaje indicando que el contexto fue cancelado y en la variable “*Err( )*” indicará el motivo correspondiente.

# *.WithDeadline()*.Ejemplo #1:

{}

```go
package main

import (
    "context"
    "fmt"
    "time"
)

func main() {
    ctx := context.Background()
    deadline := time.Now().Add(time.Second * 5)

    ctx, _ = context.WithDeadline(ctx, deadline)

    <-ctx.Done()
    fmt.Println(ctx.Err().Error())
}
```

# *.WithDeadline()*.Ejemplo #2:

Al ejecutar nuestro programa, obtendremos una salida por consola similar a la siguiente:

```
context deadline exceeded
{}
Program exited.
```

# WithTimeout

//Func

IT BOARDING

BOOTCAMP

# *.WithTimeout()*

Es tán común la funcionalidad de

“***context.WithDeadline( ctx, time.Now( ).Add( time.x * y ) )***”,

que el package *context* nos provee de un wrapper para no tener que escribir

**time.Now( ).Add( ).**

La función “***WithTimeout(ctx context.Context, timeout time.Duration)***” toma como argumentos: un contexto padre (por ejemplo el creado con “***.Background( )***”) y un timeout.

Esta función retorna “***context.WithDeadline( ctx, time.Now().Add( timeout ) )***” por lo que es un wrapper de “***context.WithDeadline( )***”.

# *.WithTimeout()*

En este ejemplo, sólo modificamos nuestra función *“main()”* del ejemplo anterior, reemplazando “*.WithDeadline( )*” por “*.WithTimeout( )*”.

La funcionalidad es exactamente la misma (obtendrás el mismo resultado), pero el código se reduce.

{}

```
func main() {

   ctx := context.Background()

   ctx, _ = context.WithTimeout(ctx, time.Second * 5)

   <-ctx.Done()
   fmt.Println(ctx.Err().Error())
}
```

Los invitamos a completar la siguiente encuesta sobre el módulo GO Bases.
**¡Es muy muy importante para nosotros contar con su feedback!**
Solamente les tomará unos minutos completarla :)

Link a la encuesta

# Gracias.

IT BOARDING

BOOTCAMP