# INTRODUCCIÓN A LA CLASE

Go Bases

# Los objetivos de esta clase son:

- Comprender, crear y utilizar punteros en Go.
- Comprender y utilizar Go Routines
- Comprender y crear canales en Go.

¡Vamos por ello!

# ¿Cómo se guarda una variable?

```
var nombre string = "Esto es una cadena"
```

| | |
|---|---|
| address | 0x001 |
| value | “Esto es una cadena” |

# Punteros

// Aplicar punteros en GO

IT BOARDING

BOOTCAMP

# // ¿Qué es un Puntero?

Un puntero es un **tipo de datos** cuyo valor es una dirección de memoria que se refiere a (o "apunta a”) otro valor almacenado en otra parte de la memoria.

# ¿Cómo creamos un puntero?

Una de las ventajas de Go es que nos permite trabajar con punteros.

Para crear un puntero utilizamos el operador `*` antes del tipo de dato que necesitamos almacenar en esa dirección de memoria.

```
{}    var p *int
```

En la variable `p` tendremos un puntero de un valor de tipo de dato int.

# Otras formas de crear punteros

- Mediante la función `new()` que recibe como argumento un tipo de dato.
- A través del shorthand de declaración de variables `:=`.

```
{}
    var p1 *int
    var p2 = new(int)
    p3 := &v
```

El tipo de datos de las variables `p1`, `p2` y `p3` es puntero a entero `*int`.

# El operador de dirección &

Para obtener la referencia o dirección de memoria de una variable, debemos anteponer a la variable el operador de dirección `&`.

```
{}
var v int = 19
fmt.Println("La dirección de memoria de v es: ", &v)
```

En este ejemplo, podemos ver que la variable `v` de tipo entero tiene el valor 19 y se almacena en la dirección de memoria `0x10414020`.

# El operador de desreferenciación * (1/2)

Desreferenciar un puntero es obtener el valor que está almacenado en la dirección de memoria a donde hace referencia el puntero, para hacerlo debemos anteponer el operador `*` a la variable puntero.

```
var v int = 19
var p *int
// Hacemos que el puntero p, referencie la dirección de memoria de la
// variable v.
p = &v
fmt.Printf("El puntero p referencia a la dirección de memoria: %v \n",p)
fmt.Printf("Al desreferenciar el puntero p obtengo el valor: %d \n",*p)
```

Este código produce el siguiente resultado.

# El operador de desreferenciación * (2/2)

TERMINAL

```
go run main.go

El puntero p referencia a la dirección de memoria: 0x10414020
Al desreferenciar el puntero p obtengo el valor: 19
```

Podemos ver que el puntero `p` referencia a la dirección de memoria `0x10414020` , y para obtener el valor 19 almacenado en esta dirección, hacemos uso del operador de desreferenciación `*p`.

# Punteros: Ejemplo Incrementar (1/2)

{}

```go
package main
import "fmt"
// Incrementar recibe un puntero de tipo entero
func Incrementar(v *int) {
   // Desreferenciamos la variable v para obtener
   // su valor e incrementarlo en 1
   *v++
}
func main() {
   var v int = 19
   // La función Incrementar recibe un puntero
   // utilizamos el operador de dirección &
   // para pasar la dirección de memoria de v
   Incrementar(&v)
   fmt.Println("El valor de v ahora vale:", v)
}
```

# Punteros: Ejemplo Incrementar (2/2)

En el ejemplo podemos ver que se puede pasar punteros como parámetros a una función usando los operadores `*` y `&`, tambien vemos el comportamiento de estos al incrementar el valor de la variable `v`, el código produce el siguiente resultado.

TERMINAL

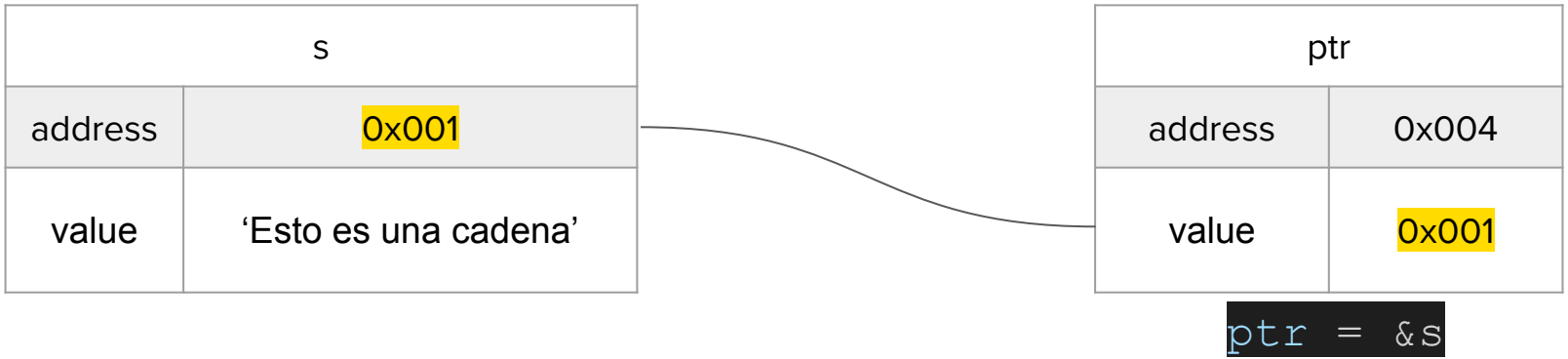
```
go run main.go

El valor de v ahora vale: 20
```

# ¿Dónde y qué almacena el puntero?

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

| name | type | address |
|---|---|---|
| s | string | 0x001 |
| ptr | *string | 0x004 |



# Paso por referencia vs. Paso por valor

## // ¿Qué aprendimos?

Un puntero es una dirección de memoria que hace referencia a otro valor.

¡Sigamos aprendiendo!

IT BOARDING

BOOTCAMP

# Concurrencia y Paralelismo

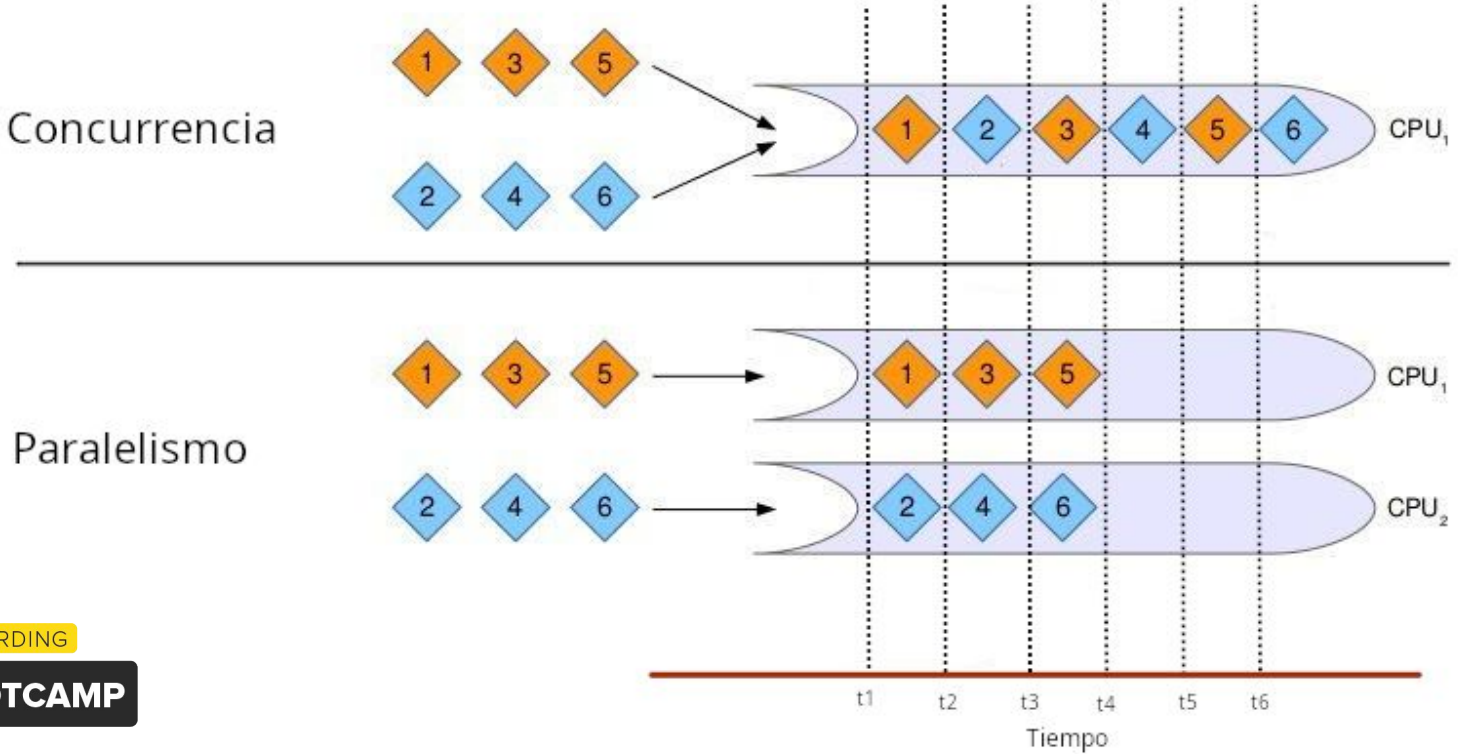
IT BOARDING

BOOTCAMP

# // ¿Qué son la concurrencia y el paralelismo?

- *Concurrencia* es el acto de iniciar, ejecutar y completar dos o más tareas en **períodos de tiempo intercalados.** No significa necesariamente que ambos se estén ejecutando en el mismo instante
- El *paralelismo* implica que dos o más tareas se ejecuten **exactamente al mismo tiempo**.

# // Concurrencia vs. Paralelismo




IT BOARDING

BOOTCAMP

# Go Routines

// lets *go*!

IT BOARDING

BOOTCAMP

# // ¿Qué son las Goroutines?

- Son la solución que nos ofrece Go para implementar la concurrencia.
- ¡NO SON HILOS! Las Goroutines son gestionadas por el Go Runtime y su scheduler, no por el sistema operativo.
- Cuando ejecutamos una función como Goroutine, su ejecución no bloqueara la continuación de aquella función que la invocó, porque correrá de forma concurrente con esta

# Función a procesar

En este caso tenemos una función que se encargará de hacer una determinada tarea. Para ello agregaremos una pausa de un segundo en él, para simular una funcionalidad que accede a una Base de Datos o envía información a una API.

{}

```
func procesar(i int) {
    fmt.Println(i, "-Inicia")
    time.Sleep(1000 * time.Millisecond)
    fmt.Println(i, "-Termina")
}
```

# Procesar en go routine principal

Ejecutaremos nuestra función 10 veces, veremos que para que inicialice el 2do procesamiento el 1ero deberia haber terminado.

```
{}
func main() {
    for i := 0; i < 10; i++{
        procesar(i)
    }
}
```

# Pausar fin ejecución

Agregaremos una pausa de 5 segundos antes de terminar el programa, ya veremos porque...

```
{}
func main() {
    for i := 0; i < 10; i++{
        procesar(i)
    }
    time.Sleep(5000 * time.Millisecond)
    fmt.Println("Termino el programa")
}
```

## // ¿Cómo podríamos ejecutar la función en simultáneo?

**Aquí entran en acción las Goroutines.**

Al ser tareas completamente independientes entre sí, en lugar de ejecutar secuencialmente `procesar()`, podríamos tratar a esas llamadas como goroutines y cada nueva invocación no debería esperar a que la anterior termine.

# Go Routine

Para indicar que queremos ejecutar la función en una Go Routine lo hacemos con la palabra reservada **go**.
Vemos cómo se ejecuta de forma simultánea y no de forma lineal:

```
{}
```

```go
func main() {
    for i := 0; i < 10; i++{
        go procesar(i)
    }
    time.Sleep(5000 * time.Millisecond)
    fmt.Println("Termino el programa")
}
```

# Go Routine

Si quitamos la pausa de 5 segundos que agregamos al final se terminará el programa antes que terminen las Go Routines.
¿Cómo podríamos hacer que el programa espere hasta que terminen todas las rutinas?

```
{}
func main() {
    for i := 0; i < 10; i++{
        go procesar(i)
    }
}
```

Una alternativa para ello es usar Canales.

# Canales

// Func

IT BOARDING

BOOTCAMP

Un canal nos va a permitir enviar valores a las Go Routines y esperar hasta recibir dicho valor.

# Sintaxis

Para definir un canal de tipo entero lo hacemos de la siguiente manera:
utilizando la palabra reservada **chan**

```
{}  c := make(chan int)
```

# Solucionando el problema planteado

Anteriormente propusimos usar canales como solución al intercambio de datos entre rutinas concurrentes

**¿Cómo lo implementamos?**

# Recibir canal como parámetro

Debemos recibir como parámetro el canal en nuestra función, para ello lo definimos como **chan int** al ser un canal de enteros

```
{}
func procesar(i int, c chan int) {
    fmt.Println(i, "-Inicia")
    time.Sleep(1000 * time.Millisecond)
    fmt.Println(i, "-Termina")
}
```

# Enviar valor a canal

Una vez que terminamos de procesar debemos enviarle el valor al canal, en este caso le enviaremos el valor de i

{}

```
func procesar(i int, c chan int) {
    fmt.Println(i, "-Inicia")
    time.Sleep(1000 * time.Millisecond)
    fmt.Println(i, "-Termina")
    c <- i
}
```

# Ejecutar ejemplo

Vamos a correr el programa ejecutando solo una función.
Vemos que sigue terminando el programa antes que termine la rutina.
¿Qué pasó? ¿Qué nos está faltando?

{}

```
func main() {
    c := make(chan int)
    go procesar(1, c)
    fmt.Println("Termino el programa")
}
```

# Recibir valor

Nos está faltando recibir el valor del canal, para que el programa quede esperando hasta recibir ese valor.
Para indicarle al programa que estamos esperando para recibir el valor del canal de la variable **c** lo hacemos con **<-c**

```
<-c // recibimos el valor del canal
variable := <-c // recibimos y lo asignamos a una variable
fmt.Println("Termino el programa en ", <-c) // recibimos y lo imprimimos
```

# Ejecutar ejemplo

De esta manera, el programa terminará una vez que se le asigne el valor en el canal

{}

```
func main() {
    c := make(chan int)
    go procesar(1, c)
    fmt.Println("Termino el programa")
    <-c
}
```

# Ejecutar ejemplo con for

```
{}
```

```go
func main() {
    c := make(chan int)

    for i := 0; i < 10; i++{
        go procesar(i, c)
    }

    for i := 0; i < 10; i++{
        fmt.Println("Termino el programa en ", <-c)
    }
}
```

# ¿Qué aprendimos?

- Hemos visto qué son los punteros y cómo utilizarlos en Go.
- Que un puntero es una dirección de memoria que hace referencia a otro valor.
- Cómo nos pueden ayudar las Go Rutines
- Cómo crear y utilizar Canales en Go.

# Gracias.

IT BOARDING

BOOTCAMP