INTRODUCCIÓN A LA CLASE

GO BASES

¿Qué aprenderemos en esta clase?

En esta clase vamos a:

- Conocer qué operadores tenemos en Go.
- Entender un Array & Slices y sus diferencias.
- Comprender que son los Maps.
- Conocer sobre los Condicionales & bucles.
- Aplicar a través diferentes comandos: operadores, array & slices, maps, condicionales & bucles.

OPERADORES

GO BASES

// ¿Qué es un Operador?

“Un operador es un símbolo que le dice al compilador que realice operaciones matemáticas o lógicas específicas.”

¿Qué tipos de operadores tenemos?

Estos son los distintos tipos de operadores que tenemos en GO

- Operadores aritméticos
- Operadores de asignación
- Operadores de comparación
- Operadores lógicos
- Operadores lógicos a nivel de bits (bit a bit)
- Operadores de dirección

Operadores Aritméticos

Los operadores aritméticos se utilizan para realizar operaciones aritméticas comunes, como suma, resta, multiplicación, etc.

```
x, y := 10, 20
```

Operador	Descripción	Ejemplo
+	Suma	x + y da 30
-	Resta el segundo operando del primero.	x - y da -10
*	Multiplica ambos operandos.	x * y da 200
/	Divide el numerador por el denominador.	x / y da 2
%	Operador de módulo; da el resto después de una división.	x % y da 0
++	Operador de incremento. Aumenta el valor de un número en uno.	x++ da 11
--	Operador de decremento. Disminuye el valor de un número en uno.	y-- da 19

Operadores Aritméticos

{}

```go
package main

import "fmt"

func main() {
   x, y := 10, 20

   fmt.Printf("x + y = %d\n", x+y)
   fmt.Printf("x - y = %d\n", x-y)
   fmt.Printf("x * y = %d\n", x*y)
   fmt.Printf("x / y = %d\n", x/y)
   fmt.Printf("x mod y = %d\n", x%y)

   x++
   fmt.Printf("x++ = %d\n", x)

   y--
   fmt.Printf("y-- = %d\n", y)
}
```

Operadores de Asignación

Los operadores de asignación se utilizan para asignar valores a las variables.

Operador	Descripción	Ejemplo
x = y	Operador de asignación simple	x = y asignará el valor de y en x
x += y	Operador de suma y asignación.	x + = y es equivalente a x = x + y
x -= y	Operador de resta y asignación.	x + = y es equivalente a x = x - y
x *= y	Operador de multiplicación y asignación.	x + = y es equivalente a x = x * y
x /= y	Operador de división y asignación.	x + = y es equivalente a x = x / y
x %= y	Operador de módulo y asignación.	x + = y es equivalente a x = x % y

Operadores de Asignación

{}

```go
package main

import "fmt"

func main() {
   var x, y = 15, 25
   x = y
   fmt.Println("= ", x)
   x = 15
   x += y
   fmt.Println("+=", x)
   x = 50
   x -= y
   fmt.Println("-=", x)
   x = 2
   x *= y
   fmt.Println("*=", x)
   x = 100
   x /= y
   fmt.Println("/=", x)
   x = 40
   x %= y
   fmt.Println("%=", x)
}
```

Operadores de comparación

Los operadores de comparación se utilizan para comparar dos valores. `x, y := 10, 20`

Operador	Descripción	Ejemplo
==	Si son iguales la condición se vuelve verdadera, de lo contrario se vuelve falsa.	(x == y) es falso
!=	Si **no** son iguales la condición se vuelve verdadera, de lo contrario se vuelve falsa.	(x != y) es verdadero
>	Si el valor izquierdo es **mayor** que el valor derecho la condición se vuelve verdadera, de lo contrario se vuelve falsa.	(x > y) es falso
>=	Si el valor izquierdo es **mayor o igual** que el valor derecho la condición se vuelve verdadera, de lo contrario se vuelve falsa.	(x >= y) es falso
<	Si el valor izquierdo es **menor** que el valor derecho la condición se vuelve verdadera, de lo contrario se vuelve falsa.	(x < y) es verdadero
<=	Si el valor izquierdo es menor o igual que el valor derecho la condición se vuelve verdadera, de lo contrario se vuelve falsa.	(x <= y) es verdadero

Operadores de comparación

```
{}
```

```go
package main

import "fmt"

func main() {
    x, y := 10, 20

    fmt.Println(x == y)
    fmt.Println(x != y)
    fmt.Println(x > y)
    fmt.Println(x >= y)
    fmt.Println(x < y)
    fmt.Println(x <= y)
}
```

Operadores lógicos

Los operadores lógicos se utilizan para determinar la lógica entre variables o valores.

Siendo que declaramos las siguientes variables: `x, y, z := 10, 20, 30`

Operador	Descripción	Ejemplo
&&	Operador lógico **y**, devuelve verdadero si ambas ambos operandos son verdaderas.	(x < y && x > z) es falso.
||	Operador lógico **o**, devuelve verdadero si alguno de los dos operandos es verdadero	(x < y || x > z) es verdadero
!	Operador lógico **NOT**. Revierte el estado de su operando. Si una condición es verdadera, el operador **NOT** la convertirá en falsa.	!(x == y && x > z) es verdadero

Operadores lógicos

```
{}
```

```go
package main

import "fmt"

func main() {
    var x, y, z = 10, 20, 30

    fmt.Println(x < y && x > z)
    fmt.Println(x < y || x > z)
    fmt.Println(!(x == y && x > z))
}
```

Operadores lógicos a nivel de bits (bit a bit)

Los operadores bit a bit se utilizan para comparar números (binarios).

Operador	Descripción		Ejemplo
&	Conjunción (AND).	Verdadero (1) mientras ninguno de los operandos sea falso (0).	(0011 0011 & 1100 0011) es igual a: 00000011
|	Disyunción (OR).	Verdadero mientras al menos uno de los operandos sea verdadero.	(0011 0011 | 1100 0011) es igual a: 1111 0011
^	Disyunción exclusiva (XOR).	Verdadero mientras los operandos sean distintos.	(0011 0011 ^ 1100 0011) es igual a: 1111 0000
<<	Corrimiento de bits a la izquierda.	Los bits son movidos a la izquierda la cantidad de posiciones que se especifique.	(0011 0011 << 2) es igual a: 1100 1100
>>	Corrimiento de bits a la derecha.	Los bits son movidos a la derecha la cantidad de posiciones se especifique.	(0011 0011 >> 2) es igual a: 0000 1100

Operadores lógicos a nivel de bits

{}

```go
package main

import "fmt"

func main() {
  var a uint = 60  /* 60 = 0011 1100 */
  var b uint = 13  /* 13 = 0000 1101 */
  var c uint = 0

  c = a & b       /* 12 = 0000 1100 */
  fmt.Printf("Conjunción - El valor de c es %d\n", c )

  c = a | b       /* 61 = 0011 1101 */
  fmt.Printf("Disyunción - El valor de c es %d\n", c )

  c = a ^ b       /* 49 = 0011 0001 */
  fmt.Printf("Disyunción exclusiva - El valor de c es %d\n", c )

  c = a << 2     /* 240 = 1111 0000 */
  fmt.Printf("Corrimiento de bits a la izquierda - El valor de c es %d\n", c )

  c = a >> 2     /* 15 = 0000 1111 */
  fmt.Printf("Corrimiento de bits a la derecha - El valor de c es %d\n", c )
}
```

Operadores de dirección

Por último, en Go también contamos con operadores de direcciones de variables y apuntadores.

Operador	Descripción	Ejemplo
&	Regresa la dirección en memoria del operando.	&x regresa la dirección en memoria de x
*	Apuntador a una variable.	*p apunta a una variable

Operadores de Dirección

```
{}
```

```go
package main

import "fmt"

func main() {
    text := "Hola Mundo”
    var pText *string

    pText = &pText

    fmt.Println(pText) /* 0xc0000a0a0 */
    fmt.Println(*pText) /* Hola Mundo */
}
```

Precedencia de operadores

La precedencia del operador determina la agrupación de términos en una expresión. Esto afecta la forma en que se evalúa una expresión.

Ciertos operadores tienen mayor precedencia que otros; por ejemplo, el operador de multiplicación tiene mayor precedencia que el operador de suma.

Por ejemplo x = 7 + 3 * 2; da como resultado 13 y no 20 porque el operador * tiene mayor precedencia que +, por lo que primero se multiplica por 3 * 2 y luego se suma a 7.

Los operadores con mayor precedencia son los que se evalúan primero, en la siguiente diapositiva veremos los operadores ordenados de mayor a menor procedencia.

Categoría	Operador	Asociatividad
Posfijo	() [] -> . ++ – –	Izquierda a derecha.
Unitario	+ – ! ~ ++ – – (type)* & sizeof	Derecha a izquierda.
Multiplicativo	* / %	Izquierda a derecha.
Adición	+ –	Izquierda a derecha.
Desplazamiento	<< >>	Izquierda a derecha.
Relacional	< <= > >=	Izquierda a derecha.
Igualdad	== !=	Izquierda a derecha.
AND nivel bits	&	Izquierda a derecha.
XOR nivel bits	^	Izquierda a derecha.
OR nivel bits	|	Izquierda a derecha.
AND lógico	&&	Izquierda a derecha.
OR lógico	||	Izquierda a derecha.
Asignación	= += -= *= /= %=>>= <<= &= ^= |=	Derecha a izquierda.
Coma	,	Izquierda a derecha.

ARRAYS & SLICES

GO BASES

// ¿Qué son los Arrays y Slices?

“Son tipos de datos que nos permiten almacenar un conjunto de datos homogéneo, es decir, todos ellos del mismo tipo”.

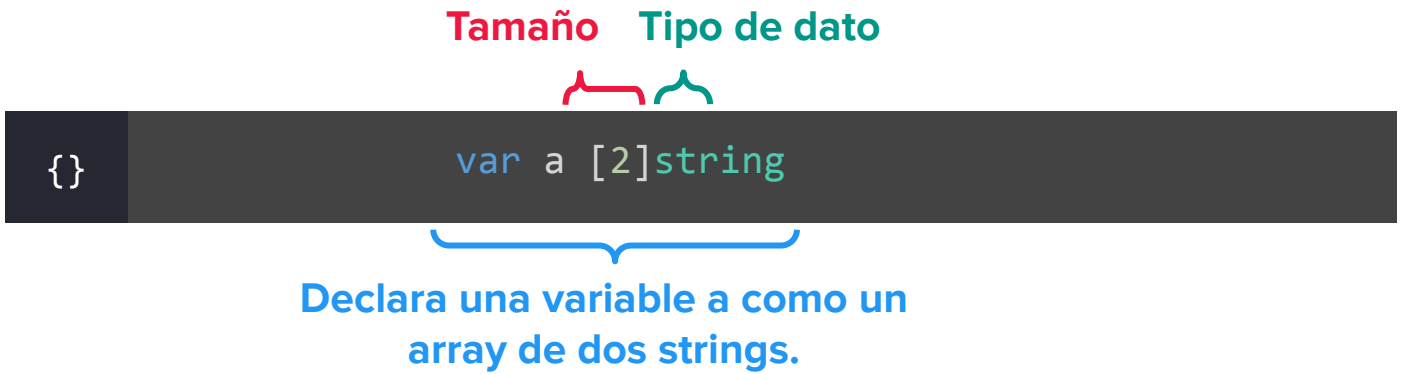
Arrays

IT BOARDING

BOOTCAMP

Array: declaración

Para declarar un array debemos definir un tamaño y un tipo de dato.



Array: asignar valores

Para asignar un valor a un array hay que especificarle la posición seguido por el valor.

Array: obtener valores

Para obtener el valor de un array solo hace falta especificar el nombre de la variable y la posición que deseas obtener:

```
{}
```

```
fmt.Printf(a[0], a[1])
fmt.Println(a)
```

Array: ejemplo completo

```
{}
```

```go
package main

import "fmt"

func main() {
  var a [2]string
  a[0] = "Hello"
  a[1] = "World"
  fmt.Println(a[0], a[1])
  fmt.Println(a)
}
```

Para tener en cuenta...

La longitud de un array es parte de su tipo, por lo que no se puede cambiar el tamaño de los arrays.
Go proporciona una forma cómoda de trabajar con arrays.

Slices

IT BOARDING

BOOTCAMP

Slice

Un slice se declara similar a un array pero a diferencia del array no le tenemos que especificar el tamaño, ya que Go se encarga de manejarlo dinámicamente.

`var s []T` esto es un slice con elementos de tipo `T`

Para obtener un valor de un slice lo hacemos de la siguiente manera:

```
{}
    var s = []bool{true, false}
    fmt.Println(s[0])
```

Crear un Slices con make()

También los slice se pueden crear con la function make()

La función make genera un array con los valores en 0 y devuelve un slice que hace **referencia** a ese array:

```
{}  a := make([]int, 5) // len (a) = 5
```

Slice: obtener rango

También se puede hacer con los arrays

Otra forma de obtener los valores de un slice es en base a un rango que esté formado por dos índices, uno de inicio y otro de fin (separados con dos puntos).

{}

```
package main
import "fmt"

func main() {
   primes := []int{2, 3, 5, 7, 11, 13}
   fmt.Println(primes[1:4]) // Si no ponemos un valor después de los : toma hasta el fin
de elementos del slice y viceversa.
}
```

Esto selecciona un rango semi abierto que incluye el primer elemento, pero excluye el último.

Slice longitud y capacidad

Un Slice tiene tanto una longitud como también una capacidad.

- La longitud de un Slice es el número de elementos que contiene.
- La capacidad de un Slice es el número de elementos del array subyacente, contando desde el primer elemento del segmento.

La longitud y la capacidad de un Slice se pueden obtener utilizando las funciones len(s) y cap(s).

Agregar a un Slice

Es común tener que agregar elementos a un slice, Go nos provee una función para esto. ¿Cómo funciona la función append?

```
{}  func append(s []T, vs ...T) []T
```

Esta función recibe como primer parámetro `(s)` el slice de tipo `(T)` al cual queremos agregarle un valor, y resto de los parámetros son los valores de tipo `(T)` que queramos agregar. La misma retorna un slice con todos los elementos anteriores más los nuevos.

```
{}  var s []int
    s = append(s, 2, 3, 4)
```

Conclusión...

Arrays vs Slices

Los arrays tienen un tamaño definido, el cual debemos definir al momento de instanciarlo, los Slice manejan el tamaño de forma dinámica, pudiendo incrementar en tiempo de ejecución.

MAPS

GO BASES

// ¿Qué son los Maps?

“Los maps nos permiten crear variables de tipo clave-valor, definiendo un tipo de dato para las claves y uno para los valores.”

Declaración de un Map

Para instanciar un map lo podemos hacer de dos maneras.

```
{}
myMap := map[string]int{}
myMap := make(map[string]string)
```

La función `make` toma como argumento el tipo de map y devuelve un map inicializado.

Longitud de un Map

Para determinar cuántos elementos (clave-valor) tiene un map, lo podemos hacer con una función que nos proporciona Go para esto len().

```
{}
var myMap = map[string]int{}
fmt.Println(len(myMap))
```

La función len () devuelve cero para un mapa no inicializado.

Acceder a elementos

Para acceder a un elemento de un map, llamamos al nombre del mismo seguido por el nombre de la clave que queremos acceder, entre corchetes.

```
{}
var students = map[string]int{"Benjamin": 20, "Nahuel": 26}
fmt.Println(students["Benjamin"])
```

La fortaleza de un mapa es su capacidad para recuperar datos rápidamente un valor según la clave. Una clave funciona como un índice, apuntando al valor asociado con esa clave.

Agregar elementos

La adición de un elemento al map se realiza utilizando una nueva clave de índice y asignándole un valor.

```
{}
var students = map[string]int{"Benjamin": 20, "Nahuel": 26}
fmt.Println(students)
students["Brenda"] = 19
students["Marcos"] = 22
fmt.Println(students)
```

Actualizar valores

Puede actualizar el valor de un elemento específico consultando su nombre de clave.

{}

```
var students = map[string]int{"Benjamin": 20, "Nahuel": 26}
fmt.Println(students)
students["Benjamin"] = 22
fmt.Println(students)
```

Eliminar elementos

Go nos proporciona una función para el borrado de elementos de un map.

```
{}
var students = make(map[string]int)
students["Benjamin"] = 20
fmt.Println(students)
delete(students, "Benjamin")
fmt.Println(students)
```

Recorrer elementos de un Map

El for nos permite recorrer los elementos de nuestro map.

```
{}
var students = map[string]int{"Benjamin": 20, "Nahuel": 26}
for key, element := range students {
    fmt.Println("Key:", key, "=>", "Element:", element)
}
```

CONDICIONALES & BUCLES

GO BASES

For

//

IT BOARDING

BOOTCAMP

¿Para qué sirve el for?

El bucle for lo utilizamos para ejecutar un bloque de código repetidamente. Por lo general se utiliza para iterar sobre una secuencia (slice, array, map o string.)
En Go, hay diferentes formas de utilizarlo:

- Standard For
- For Range
- Bucle infinito
- Bucle While

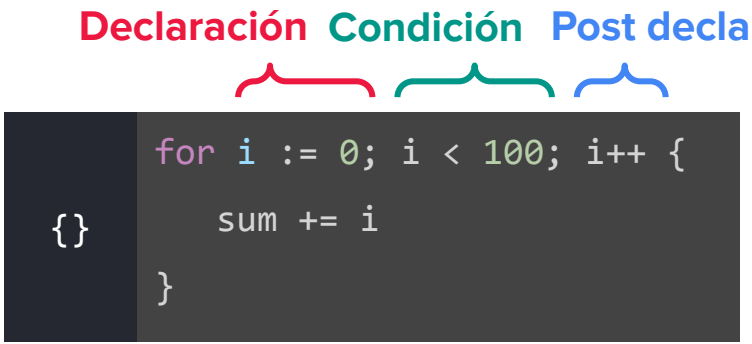
Veremos cada una de estas en las siguientes slides.

¿Qué es Standard For?

Go tiene una sintaxis bastante estándar compuesta por tres componentes similar a la de C, Java o JavaScript. La gran diferencia es que no hacen falta los paréntesis alrededor de los componentes.

El bucle se ejecuta de la siguiente manera:

1. **Declaración:** Declara la variable y la disponibiliza dentro del scope del loop.
2. **Condición:** si la condición se cumple ejecuta el código, de lo contrario el bucle se termina.
3. **Post declaración:** se ejecuta la post declaración
4. **Vuelve al paso 2**



La declaración mayormente se utiliza para saber el índice actual del bucle.

For range

La función `range` itera por los elementos de la mayoría de las estructuras de datos. Cuando se usa range en un arreglo o slice, se obtiene el valor del elemento y el índice donde se encuentra.

En el siguiente ejemplo vemos como recorrer un slice con `range`

{}

```
frutas := []string{"manzana", "banana", "pera"}
for i, fruta := range frutas {
    fmt.Println(i, fruta)
}
```

Bucle infinito

Los bucles infinitos son útiles dentro de las Go routines (tema avanzado), cuando tienen un proceso que debe continuar perpetuamente.

{}

```
sum := 0
for {
    sum++ // repite para siempre
}
// nunca llega ya que el bucle es infinito
```

Bucle “*while*”

El bucle *while* te permite ejecutar un bloque de código hasta que la condición se deje de cumplir. A diferencia del bucle **`for`** de 3 componentes, esto se hace solo con uno, la **condición**. Go no tiene la palabra reservada *while* para este bucle.

```
{}
    sum := 1
    for sum < 10 {
        sum += sum
    }
    fmt.Println(sum)
```

Después del **`for`** solo ponemos la condición

Romper un bucle

Romper un bucle antes de que termine puede ser útil, especialmente en un bucle infinito. La palabra reservada `break` nos permite cortar con la ejecución del bucle.

```
{}
sum := 0
for {
    sum++
    if sum >= 1000 {
        break
    }
}
fmt.Println(sum)
```

Saltar a la siguiente iteración

Puede ser útil pasar a la siguiente iteración de un bucle antes de que termine de correr todo el código del mismo. Esto se hace con la palabra reservada `continue`.

```
for i := 0; i < 10; i++{
   if i % 2 == 0 {
       continue
   }
   fmt.Println(i, "is odd")
}
```

El siguiente ejemplo solo imprime los números impares, cuando detecta un número par salta a la siguiente iteración.

IF... ELSE

//

IT BOARDING

BOOTCAMP

Instrucción if

La instrucción `if` permite ejecutar un porción de tu código si la condición es verdadera.

Sintaxis

```
if condición {
    // código que va a ejecutarse si la condición es verdadera (true)
}
```

A diferencia de otros lenguajes en Go no es necesario que la condición esté entre `()`

Instrucción if

En el siguiente ejemplo evaluaremos el sueldo de una persona, y que si es mayor a 3000 deberá pagar impuestos.

{}

```go
package main

import "fmt"

func main() {
   sueldo := 4500
   if sueldo > 3000 {
       fmt.Println("Esta persona debe pagar impuestos")
   }
}
```

¿Qué sucede si necesito que se ejecute una instrucción en caso de que la condición resulte falsa?

Instrucción if...else

La instrucción `if ... else ...` nos permite ejecutar una institución si la condición es verdadera y otra si es falsa.

Sintaxis

```
if condición {
    // código que va a ejecutarse si la condición es verdadera (true)
} else {
    // código que va a ejecutarse si la condición es falsa (false)
}
```

Instrucción if...else

Siguiendo el ejemplo anterior, ahora si queremos que nuestro programa diga que no debe pagar impuestos, utilizamos la instrucción `else`

{}

```
package main

import "fmt"

func main() {
    sueldo := 4500
    if sueldo > 3000 {
        fmt.Println("Esta persona debe pagar impuestos")
    } else {
        fmt.Println("Esta persona NO debe pagar impuestos")
    }
}
```

Instrucción if...else if... else

La instrucción `if ... else if ... else` nos permite combinar varias declaraciones `if ... else`

Sintaxis

```
if condición-1 {
   // código que va a ejecutarse si la condición-1 es verdadera (true)
} else if condición-2 {
   // código que va a ejecutarse si la condición-2 es verdadera (true)
} else {
   // código que va a ejecutarse ambas condiciones son falsas (false)
}
```

Instrucción if...else if... else

Ahora si queremos que no pague impuestos si su sueldo es menor a $3000, que pague 10% si su sueldo es mayor a $3000 y 15% si es mayor a $4000. Lo hacemos de la siguiente manera:

{}

```go
package main

import "fmt"

func main() {
   var sueldo float64 = 4000
   if sueldo <= 3000 {
       fmt.Println("Esta persona debe pagar impuestos")
   } else if sueldo <= 4000 {
       fmt.Printf("Debe pagar $%4.2f de su sueldo\n", (sueldo/100)*10)
   } else {
       fmt.Printf("Debe pagar $%4.2f de su sueldo\n", (sueldo/100)*15)
   }
}
```

If con declaración corta

La instrucción `if` nos permite una sintaxis compuesta donde podemos instanciar una variable antes de la condición.

Sintaxis

```
if  var declaración;  condición {
   // código que va a ejecutarse si la condición es verdadera (true)
}
```

If con declaración corta

En este ejemplo declaramos la variable edad, y luego la utilizamos para evaluarla en nuestra condición.

```go
package main

import "fmt"

func main() {
   if edad := 20; edad > 150 {
       fmt.Println("¿Eres inmortal?")
   } else if edad >= 18 {
       fmt.Println("Eres mayor de edad")
   } else if edad < 18 && edad > 0 {
       fmt.Println("Eres menor de edad")
   } else {
       fmt.Println("Edad fuera del rango")
   }
}
```

Switch

//

IT BOARDING

BOOTCAMP

Switch básico

El condicional `switch` se utiliza para seleccionar uno de los muchos bloques de código que se ejecutarán, esto una alternativa a múltiples `else-if`

```
switch expresión {
case condicion:
    // Código que se va ejecutar en caso de que se cumpla esta condición
default:
    // Código que se va ejecutar en caso de que no se cumpla ninguna condición
}
```

Algunos aspectos a tener en cuenta:

- A diferencia de otros lenguajes de programación en los cuales el condicional switch requiere de la palabra reservada `break`, en Go **no es necesario** usarla.
- La expresión que va justo después del switch **no requiere paréntesis**.
- La palabra reservada `default` se utiliza si no se encuentra ninguna coincidencia.

Switch básico ejemplo

{}

```go
package main

import "fmt"

func main() {
   day := 1
   switch day {
   case 0:
       fmt.Println("Lunes")
   case 1:
       fmt.Println("Martes")
   case 2:
       fmt.Println("Miércoles")
   case 3:
       fmt.Println("Jueves")
   case 4:
       fmt.Println("Viernes")
   case 5:
       fmt.Println("Sábado")
   case 6:
       fmt.Println("Domingo")
   default:
       fmt.Println("Desconocido")
   }
}
```

Switch sin condición

Podemos utilizar un `switch` sin condición, agregando la condición en el case.
Ej `case condición:` es similar al `if... else...`

{}

```go
package main

import "fmt"

func main() {
    var edad uint8 = 18
    switch {
    case edad >= 150:
        fmt.Println("¿Eres inmortal?")
    case edad >= 18:
        fmt.Println("Eres mayor de edad")
    default:
        fmt.Println("Eres menor de edad")
    }
}
```

Switch con múltiples casos

Los case pueden tener múltiples valores separadas por comas como podemos ver en el siguiente ejemplo.

```
{}
```

```go
package main

import "fmt"

func main() {
    day := "domingo"

    switch day {
    case "lunes", "martes", "miércoles", "jueves", "viernes":
        fmt.Printf("%s es un día de la semana\n", day)
    default:
        fmt.Printf("%s es un día del fin de la semana\n", day)
    }
}
```

Switch con declaración corta

Así como lo podemos hacer en un `if`, los `switch` también permiten declarar una variable para usarla dentro de la sentencia. Lo hacemos de la siguiente manera:

{}

```go
package main

import (
    "fmt"
)

func main() {
    switch day := "domingo"; day {
    case "lunes", "martes", "miércoles", "jueves", "viernes":
        fmt.Printf("%s es un día de la semana\n", day)
    default:
        fmt.Printf("%s es un día del fin de la semana\n", day)
    }
}
```

Switch con fallthrough

Dentro de la sentencia `switch` también podemos utilizar la palabra reservada fallthrough para que ejecute la siguiente porcion de codigo del switch

{}

```go
package main

import (
   "fmt"
   "time"
)

func main() {
   today := time.Now()
   var t int = today.Day()
   switch t {
   case 5, 10, 15:
       fmt.Println("Limpia tu casa.")
   case 25, 26, 27:
       fmt.Println("Comprar comida.")
       fallthrough
   case 31:
       fmt.Println("Hoy hay fiesta.")
   default:
       fmt.Println("No hay información disponible para ese día.")
   }
}
```

Gracias.

IT BOARDING

BOOTCAMP

Autor: Benjamin Berger
Email: benjamin@digitalhouse.com
Última fecha de actualización: 13-06-21

IT BOARDING
BOOTCAMP