

## ¿Qué es Go?

Es un lenguaje open source creado por Google, su aparición fue en el año 2009 y sus diseñadores son Robert Griesemer, Rob Pike y Ken Thompson. Además es un lenguaje de **programación concurrente, compilado, estructurado, tipado estático e imperativo**. También no es un lenguaje orientado a objetos y cuenta con un recolector de basura.

## Características de Go

- Go utiliza una sintaxis parecida a la de C.
- Posee un tipado estático y resulta ser igual de eficiente que C.
- Tiene muchas características y facilidades de lenguajes dinámico como Python.
- No es orientado a objetos debido a que no existe una jerarquía de tipos, pero si implementa la interface.
- Declaraciones de variable simple.
- Duck typing – Tipificación dinámica de datos.
- No tiene excepciones.

## Declaración de variables

Para declarar una variable se usa la palabra reservada VAR, haciendo referencia que es una variable después el nombre de la variable y por último el tipo.

A diferencia de otros lenguajes de programación la declaración en GO es al revés. GO inicializa la variable con un valor por defecto.

```
var numero int
fmt.Println(numero)
```

Figure 1: Declaración de variable en Go

En este caso **numero**, Go lo inicializará con el valor de 0 pero podemos asignarle un valor por defecto.

```
var numero int = 10
fmt.Println(numero)
```

Figure 2: Asignación de variable

Para declarar una variable de manera rápida determinando el tipo de manera dinámica una vez que la variable se le asigna un tipo este ya no se puede cambiar el operador `:=` solo se cuándo se declara una nueva variable.

```
nombre := "hola"
```

*Figure 3: Declaración rápida de una variable*

Otra forma de declarar variables omitiendo el tipo es usando **var + nombre** y dándole una asignación.

```
var numero4 = 70
```

*Figure 4: Otra forma de declaración usando el Duck typing*

En GO se pueden asignar valores de forma consecutiva.

```
nombre, numero = "Eduardo", 20
```

*Figure 5: Asignación de valores de forma consecutiva*

Esta asignación de valores puede resultar útil al momento de intercambiar valores entre variables del mismo tipo.

```
nombre = "Pedro"  
name := "Juan"  
nombre, name = name, nombre
```

*Figure 6: Intercambio de valores entre variables*

```
Nombre: Juan  
Name: Pedro
```

*Figure 7: Variables Intercambiadas*

Esta forma de asignación de valores se puede utilizar cuando se declara una nueva variable.

```
nombre2, name := "Gloria", "Teresa"

fmt.Println("Nombre2: ", nombre2)
```

*Figure 8: Declaración de una variable y asignación*

En este caso **nombre2** es la variable creada y su valor es **Gloria**, mientras que **name** era una variable previamente declarada por lo cual su valor asignado es **Teresa**.

Para declarar múltiples variables se usa **var()**, dentro de los paréntesis se declaran las variables con sus asignaciones. Al usar esta forma de declaración se puede asignar el tipo o bien usando el Duck typing (tipado dinámico).

```
var(
    variable string
    example, example2 = 1,2
)
```

*Figure 9: Declaración de múltiples variables*

## NOTAS:

En **Go** toda variable que se es declarada debe de ser utilizada de la misma forma sucede con los **import** sino se usa una librería importada el compilador obliga a eliminarla o usarla.

La librería **fmt** es una abreviación de **format**, esta se usa para poder imprimir e ingresar datos.

## Alcance de las variables

En Go el alcance de las variables es como cualquier otro lenguaje de programación es decir las variables pueden funcionar de manera global o de manera local.

```
func main() {  
    /*...*/  
  
    var variableLocal = "Esto es una variable local"  
  
    fmt.Println(variableLocal)  
}
```

Figure 10: Ejemplo de variable local

```
var variableLocal = "Esto es una variable local"  
  
func main() {  
    /*...*/  
  
    imprimir()  
}  
  
func imprimir() {  
    fmt.Println(variableLocal)  
}
```

Figure 11: Ejemplo de variable global

### NOTAS:

Cuando se declara una variable global no se permite usar el operador `:=` por lo que es obligado declararla de forma apropiada.

# Tipos de datos en Go

Comencemos por los tipos básicos, estos pueden ser **números (int)**, **booleanos** y **cadenas**.

Tipos numéricos:

## Enteros sin signo

Tipo	Rango	Consumo de memoria
uint8	0 a 255	1 byte
uint16	0 a 65535	2 byte
uint32	0 a 4294967295	4 byte
uint64	0 a 18446744073709551615	8 byte

## Enteros con signo

Tipo	Rango	Consumo de memoria
int8	-128 a 127	1 byte
int16	-32768 a 32767	2 byte
int32	-2147483648 a 2147483647	4 byte
int64	-9223372036854775808 a 9223372036854775807	8 byte

## Alias

Tipo	Rango	Consumo de memoria
byte (lo mismo que uint8)	0 a 255	1 byte
rune (lo mismo que int32)	-2147483648 a 2147483647	4 byte

## Tipos dependientes de la plataforma

Tipo	Rango
uint	Depende de la plataforma (32bits o 64 bits)
int	Depende de la plataforma (32bits o 64 bits)
uintptr	Entero suficientemente largo para almacenar un puntero

Tipos float:

Tipo	Precisión	Consumo de memoria
Float32	6 dígitos	4 byte
Float64	15 dígitos	8 byte
Complex64	Numero complejo para float32	8 byte
Complex128	Numero complejo para float64	16 byte

Tipos boolean:

Los tipos de datos boolean solo pueden almacenar dos valores **false** o **true**.

Tipo string:

En Go a diferencia de los demás lenguajes de programación los string son una secuencia de bytes.

Otras características del tipo string en Go es que son indexables e inmutables. Cuando se intenta acceder a un carácter que conforma un string lo que se obtiene es el código Unicode (utf-8).

```
var stringExample string

stringExample= "Bienvenidos"

fmt.Println(stringExample[3])
```

```
<4 go setup calls>
110

Process finished with exit code 0
```

U+006E	n	110	LATIN SMALL LETTER N
--------	---	-----	----------------------

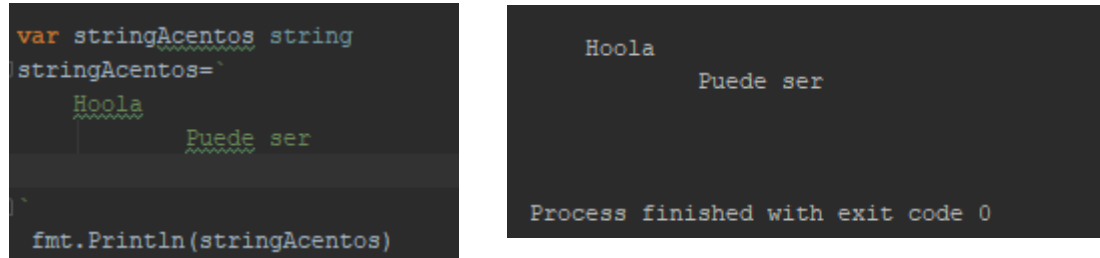
Como se aprecia en el ejemplo, si tratamos de acceder a la posición 3 nos devuelve un número que es el **110** el cuál en utf-8 corresponde a la letra n minúscula.

Para obtener un segmento de caracteres o subcadena como se conoce en java, Go lo hace con algo llamado **Slice**.

```
fmt.Println(stringExample[0:4])
```

Figure 12: Uso de Slice para obtener una subcadena

En Go hay dos formas de crear cadenas, una es usar las comillas dobles ("" ) o con el acento invertido (`), el uso del acento invertido es para que Go use o imprima la cadena tal y como esta, es decir si lleva sangría y espacio además de ignorar los caracteres especiales.

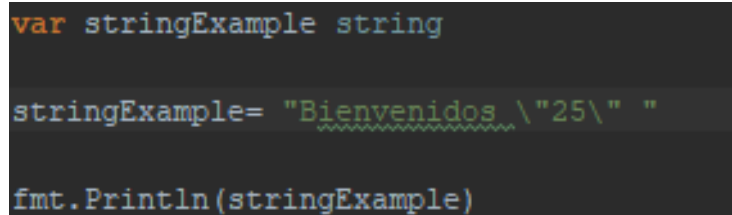


```
var stringAcentos string
stringAcentos=`
    Hoola
    Puede ser
`
fmt.Println(stringAcentos)
```

```
Hoola
    Puede ser

Process finished with exit code 0
```

El uso de caracteres especiales se hace usando la pleca invertida (\).



```
var stringExample string
stringExample= "Bienvenidos \"25\" "
fmt.Println(stringExample)
```

*Figure 13: Uso de caracteres invertidos en una cadena*

Para convertir un tipo de dato a string se usa la función **ltoa** del paquete **strconv**.

En los tipos de conjuntos tenemos los **arreglos** y las **estructuras**.

```
type Student struct{  
    name string  
    lastName string  
    age int  
    average float64  
    genre bool  
}
```

*Figure 14: Ejemplo de una estructura*

```
var simpleArray[2] int  
simpleArray[0] = 1  
simpleArray[1] = 2
```

*Figure 15: Ejemplo de un array con su inicialización*

**Nota:**

- Para conocer la cantidad de caracteres que posee una cadena se usa la función **len**.



## Los **Slice**:

Son iguales a los arrays con la diferencia que estos no se especifica la dimensión que va a tener, además de ser similar a los arrays dinámicos de otros lenguajes.

```
var sliceExample []int
fmt.Println(sliceExample)
```

Figure 16: Ejemplo de creación de un slice

```
sliceInit := []int{1,2,3,4,5}
fmt.Println(sliceInit)
```

Figure 17: Declaración de Slice con su inicialización

También se puede usar la función **make** para crear un slice

```
sliceMake := make([]int, 8)

fmt.Println(sliceMake)
fmt.Println(a: "Longitud del Slice", len(sliceMake))
fmt.Println(a: "Capacidad del Slice", cap(sliceMake))
```

Figure 18: Uso de la función make

A esta función se le pueden pasar 3 parámetros, uno es el tipo del slice, segundo el tamaño y tercero la capacidad.

Los slice al ser similares a los arrays llega un punto en el cual se quedan sin capacidad y al ser estáticos esta no se puede modificar, para ello Go creo la función **append** la cual permite que el slice aumente su longitud y la capacidad. Pero esta función lo que realmente hace es crear otro slice con otro array de fondo y destruir el anterior.

```

var y []int

for i := 1; i < 15; i++){
    y = append(y,i)
    fmt.Println(y)
    fmt.Printf( format: "Longitud y: %d, Capacidad y: %d, Elementos y: %d \n", len(y),cap(y),i)
}

```

Este es un pequeño ejemplo para demostrar que append duplica la capacidad del array que esta de fondo.

```

Array
[1]
Longitud y: 1, Capacidad y: 1, Elementos y: 1
Array
[1 2]
Longitud y: 2, Capacidad y: 2, Elementos y: 2
Array
[1 2 3]
Longitud y: 3, Capacidad y: 4, Elementos y: 3
Array
[1 2 3 4]
Longitud y: 4, Capacidad y: 4, Elementos y: 4
Array
[1 2 3 4 5]
Longitud y: 5, Capacidad y: 8, Elementos y: 5
Array
[1 2 3 4 5 6]
Longitud y: 6, Capacidad y: 8, Elementos y: 6
Array
[1 2 3 4 5 6 7]
Longitud y: 7, Capacidad y: 8, Elementos y: 7
Array
[1 2 3 4 5 6 7 8]
Longitud y: 8, Capacidad y: 8, Elementos y: 8
Array
[1 2 3 4 5 6 7 8 9]
Longitud y: 9, Capacidad y: 16, Elementos y: 9
Array
[1 2 3 4 5 6 7 8 9 10]
Longitud y: 10, Capacidad y: 16, Elementos y: 10
Array
[1 2 3 4 5 6 7 8 9 10 11]
Longitud y: 11, Capacidad y: 16, Elementos y: 11
Array
[1 2 3 4 5 6 7 8 9 10 11 12]
Longitud y: 12, Capacidad y: 16, Elementos y: 12

```

En los slice no solo se puede añadir sino que también se puede copiar, con la función **copy**.

```
sliceOrigin := []int{1, 2, 3}
sliceDestiny := []int{4, 5, 6}
copy(sliceDestiny, sliceOrigin)
fmt.Println(sliceOrigin, sliceDestiny)
```

```
<4 go setup calls>
[1 2 3] [1 2 3]

Process finished with exit code 0
```

Algo muy importante sobre la función copy es que si el slice de destino tiene una longitud menor al slice de origen solo se copian los datos necesarios, lo mismo sucede a la inversa si los datos que contiene el slice de origen es menor a los datos que contiene el slice destino solo se copian los datos necesarios.

```
sliceOrigin := []int{1, 2, 3}
sliceDestiny := []int{4, 5}
copy(sliceDestiny, sliceOrigin)
fmt.Println(sliceOrigin, sliceDestiny)
```

```
<4 go setup calls>
[1 2 3] [1 2]

Process finished with exit code 0
```

## Los Maps

Los maps en Go son iguales a los de otros lenguajes de programación.

```
maps := make(map[string]string)
fmt.Println(maps)

mapExample := make(map[string]string, 2)
fmt.Println(mapExample)

maps["Nombre"] = "Paola"
maps["Edad"] = "20"

fmt.Println(maps["Nombre"])
fmt.Println(maps["Edad"])
```

```
dias := map[int]string{  
    1: "Lunes",  
    2: "Martes",  
    3: "Miercoles",  
}
```

Otra forma de declarar y asignar valor a un map.

Para borrar un dato del map existe la función **delete** .

## Casting de valores

En Go el casting tiene que ser de forma explicito debido que Go no es un lenguaje de gran potencia como lo son Java o C# que se puede hacer un casteo de forma implícita.

```
var entero8 uint8  
var entero32 uint32  
  
entero8 = 15  
entero32 = 230  
  
fmt.Println(entero32 + uint32(entero8))
```

*Figure 19: Ejemplo de castin implícito en Go*

## Estructuras de control

En Go solo existe una estructura iterativa que es el **for**.

```
for i := 0; i < 10 ; i++ {  
    fmt.Println( a: "Número:", i)  
}
```

Figure 20: Estructura for en Go

La estructura **for** en Go esta compuesta de la siguiente manera:

- La declaración de una variable antes de la primera iteración
- La condición a evaluar antes de cada iteración
- La sentencia después de cada iteración, es decir el aumento de la variable definida.

### Nota:

A diferencia de otros lenguajes como C, Java o JavaScript no hay paréntesis alrededor de los componentes del **for** y las llaves son necesarias.

La declaración de la variable y la sentencia de incremento son opcionales.

```
i:=0  
for ; i < 10 ; {  
    i++  
    fmt.Println( a: "Número:", i)  
}
```

Si le quitamos los ; el while que se conoce en C, en Go es el mismo for.

```
i:=0
for i < 10 {
    i++
    fmt.Println( a: "Número:", i)
}
```

## For Range

El for range es como el for each de otros lenguajes de programación recorre un arreglo o un slice del cual podemos tener acceso a sus valores.

```
objects := []string{
    "Mesa",
    "Silla",
    "Ventilador",
    "Licuadora",
}

for index, obj := range objects{
    fmt.Printf( format: "El objeto es %q y esta en el index %d \n", obj, index)
}
```

Otra forma de usar el for range

```
for _, obj := range objects{
    fmt.Printf( format: "El objeto es %q \n", obj )
}
```

El guión bajo se utiliza para omitir el índice ya que range devuelve el valor y el índice.

## Estructura if

Al igual que for el if no es necesario que su condición esté rodeada de paréntesis, pero las llaves son necesarias.

```
if i <= edad {  
    fmt.Println( a: "Si")  
}
```

*Figure 21: Estructura if en Go*

Al igual que en otros lenguajes de programación existe el **if else**

```
if i <= edad {  
    fmt.Println( a: "Usted es menor de edad")  
} else{  
    fmt.Println( a: "Usted tiene 30 años")  
}
```

*Figure 22: If Else en Go*

## Estructura **switch**

En Go la sentencia switch es igual que en otros lenguajes como C, C++, Java, PHP, etc. Con la excepción que en Go solo se ejecuta el caso seleccionado, por lo que la sentencia break necesaria en los demás lenguajes es automática en Go.

```
var choice int8

fmt.Println(a: "Ingresa un número")
fmt.Scanln(&choice)

switch choice {
case 1:
    fmt.Println(a: "Hoy lloverá")
case 2:
    fmt.Println(a: "No va a llover")
}
```

*Figure 23: Switch en Go*

A veces es necesario que el switch evalúe más casos, para eso los desarrolladores de Go crearon la palabra **fallthrough** la cual indica que el programa siga evaluando más casos.



# Funciones

Go al igual que los demás lenguajes de programación incluye lo que son las funciones.

```
func printExample (name string){  
  
    fmt.Println(name)  
  
}
```

También las funciones pueden retornar un valor. Sin embargo hay dos formas de que se retorne un valor.

La primera forma es dejar declarado el tipo del valor que se va a retornar.

```
func printExample() string {  
  
    return "Jasson"  
  
}
```

La segunda es dejar declarado el nombre y el tipo del valor que será retornado.

```
func printExample() (name string) {  
    name = "Jasson"  
    return  
  
}
```

Algo muy interesante en Go es que existen las funciones variables, estas funciones pueden recibir varios parámetros.

```
func multiplicar(numbers ...int8) int {  
    result := 1  
  
    for _, number := range numbers {  
        result *= int(number)  
    }  
  
    return result  
}
```

Clausuras en Go, las clausuras son funciones que están dentro de otra función la función interna tiene acceso a las variables de la función externa. Las clausuras se pueden hacer de varias maneras una es usando una variable para almacenar la función, la otra es crear una función dentro del main guardarla en una variable.

```
multi := multiplicar  
  
fmt.Println(multi( numbers: 1,2,3,4))  
  
func multiplicar(numbers ...int8) int {  
    result := 1  
  
    for _, number := range numbers {  
        result *= int(number)  
    }  
  
    return result  
}
```

El ejemplo anterior es una forma de hacerla, el siguiente ejemplo es la otra forma de hacer funciones dentro de otra función.

```
result :=0
numbers :=[...]int {
    1,
    2,
    4,
    5,
    6,
}

suma := func() {
    for _, numero := range numbers{
        result += numero
    }

    fmt.Printf( format: "La suma es : %d", result)
}

suma()
```

En el ejemplo anterior la forma de implementar es usando una función anónima es decir una función que no posee un nombre.

Para ver si una función contiene implementación se puede usar el valor **nil** que sirve para determinar si tiene o no implementación, además las funciones no se pueden comparar entre ellas.

Además las funciones en Go pueden retornar más de un valor, lo cual es un campo de muchas posibilidades.

```
n,v := retornoMultiple()

fmt.Printf( format: "Número: %d , Cadena: %q ",n,v)
}

func retornoMultiple () (numero int, cadena string){
    numero = 1
    cadena = "Hola"

    return
}
```

En Go la palabra reservada **defer** funciona para ejecutar una función al final es decir cuando termine el todo lo que este en el main exceptuando las funciones que tengan esta palabra reservada.

```
func main() {  
    defer multiplicar( numbers: 1, 2, 5)  
    printName()  
}  
  
func multiplicar(numbers ...int8) {  
    result := 1  
  
    for _, number := range numbers {  
        result *= int(number)  
    }  
  
    fmt.Println(result)  
}  
  
func printName() {  
    fmt.Println( a: "Ejemplo de defer")  
}
```

Para manejar errores en Go se usa **panic y recover**, la función panic imprime un mensaje de error pero este detiene por completo el programa, recover recupera el error que mostró panic y es una forma de manejar errores conocidos pero para hacer este manejo se usa una función anónima con defer.

```
fmt.Println( a: "Hola")  
  
defer func() {  
    cadena := recover()  
    fmt.Println(cadena)  
}()  
panic( v: "Error")
```

## Punteros

Go también puede usar punteros igual que otros lenguajes de programación. A diferencia de C Go no tiene aritmética de punteros.

En Go la declaración de un puntero es **\*T** donde T es el tipo al que se apunta. El operador **&** genera un puntero al valor que contiene otra variable.

```
a := 25
fmt.Println(a: "Valor de a:", a)
fmt.Println(a: "Dirección de memoria de a:", &a)

b := &a

fmt.Println(a: "Valor de b:", b)
fmt.Println(a: "B contiene: ", *b)
```

## Type

En Go podemos hacer nuestro propio tipo de dato pero tomando como base un tipo ya existente, para hacer esto se usa la palabra reservada **Type**.

```
type Money int
```

Además a estos tipos personales se les puede añadir funciones.

```
func (d Money) String() string{
    return fmt.Sprintf(format: "$%d",d)
}
```

## Estructuras

Go al igual que C posee las estructuras las cuales son lo más cercano a una clase en un lenguaje orientado a objetos.

```
type Persona struct {  
    Nombre string  
    edad   int  
}
```

Puede que en Go no exista la jerarquía de clases como lo tienen Java o C# pero se puede hacer una herencia de manera implícita.

```
type Empleado struct {  
    Persona  
    Puesto string  
}
```

```
employee := Empleado{  
    Persona: Persona{  
        Nombre: "Manolo",  
        edad: 40,  
    },  
    Puesto: "Contador",  
}  
  
fmt.Println(employee.Nombre)  
fmt.Println(employee.edad)  
fmt.Println(employee.Puesto)
```

En este ejemplo vemos como la estructura persona está dentro de empleado pero no tiene nombre esto se llama **propiedad anónima**, es decir una propiedad que no posee nombre. Además de notarse que de forma implícita estamos haciendo una herencia de persona a empleado.

**Nota:** Si se quiere acceder a una propiedad de una estructura pero que este en un archivo por separado, el nombre de la propiedad debe de ser mayúscula debido a que si se usa en minúscula su ámbito solo se puede usar en el archivo que lo contiene.