



# Backend III

units

## ▼ Fundamentos de Go

### ▼ Packages

Cada archivo de Go pertenece a un package, se declara

```
package packagename
```

- La carpeta que contiene los distintos archivos del package y el package se deben llamar igual a este ultimo
- La declaración del package anterior debe ser la primera línea de código en el archivo fuente de Go
- Todas las funciones, tipos y variables definidas en el archivo fuente de Go pasan a formar parte del paquete declarado.
- El nombre de los packages debería ser una sola palabra y todo en minúscula.

### Package Main

Los programas en Go comienzan a ejecutarse en el package main. Este es un package que se usa con programas que están destinados a ser ejecutables. Se los conoce como comandos. Los otros programas se denominan simplemente packages.

```
package main
```

La función main() es especial, ya que es el punto de entrada de un programa ejecutable. Veamos un ejemplo de un programa ejecutable en Go.

```
func main() {  
    //code  
}
```

### Importaciones

Importación de packages individual:

```
import "fmt"  
import "time"
```

Importación de packages grupal

```
import (  
    "fmt"  
    "time"  
)
```

## ▼ Modules

Antes de iniciar un módulo Go, tenemos que asegurarnos de crear un nuevo repositorio en GitHub —u otros controles de versión (por ejemplo, GitLab)—. Luego, podremos clonar el repositorio. Para iniciar un módulo Go usamos este comando:

**Dominio**      **Nombre del módulo**

```
{ } go mod init github.com/usuarioGithub/go-simple-module
```

## El dominio y el nombre del módulo deben coincidir con el nombre del repositorio que ya se creó.

Después de que se haya iniciado el módulo, se crea el archivo .go.mod. Este contiene las dependencias que se utilizan en la aplicación.

### Agregando Dependencias

Se puede agregar una dependencia a la aplicación usando el comando go get. Go permite agregarle el flag -u para descargarla si no existe o actualizar la dependencia.

```
go get -u github.com/gin-gonic/gin
```

Luego de agregar la dependencia, se puede utilizar dentro de nuestros archivos del módulo agregando el import.

```
import (
    "fmt"
    "github.com/gin-gonic/gin"
)
```

## ▼ Variables

Una variable es una porción de almacenamiento en memoria que contiene datos guardados temporalmente para trabajar con ellos.

### Nomenclatura

- Su nombre debe comenzar con una letra, no puede comenzar con un número.
- Los nombres de las variables no pueden contener espacios.
- Si el nombre de una variable comienza con una letra minúscula, solo se puede acceder dentro del package actual. Esto se considera como variables no exportadas.
- Si el nombre de una variable comienza con mayúscula, se puede acceder a ella desde packages fuera del package actual. Esto se considera como variables exportadas.
- Si un nombre consta de varias palabras, se usa la nomenclatura **camelCase**. Es decir, cada palabra después de la primera debe tener la primera letra en mayúscula. Por ejemplo: empName, empAddress, etc.

### Sintaxis



### Nombre

El nombre de la variable debe ser descriptivo del dato que se va a guardar en ella para ser manipulado.

### Asignación de Valor

Especificamos una variable con el tipo de dato int para declarar y utilizar una variable de tipo entero, es decir, números sin fracción.

```
var horas int
```

Le asignamos el valor "20":

```
horas = 20
```

## Declaración de una Variable

Podemos asignar valores a múltiples variables en una sola línea:

```
producto, precio := "Jean", 10.5
```

La declaración de variables se puede agrupar en bloques para una mayor legibilidad y calidad del código:

```
var (
    producto = "Course"
    cantidad = 25
    precio   = 40.50
    enStock  = true
)
```

Go nos permite realizar una asignación sin hacer una declaración previa. Para eso utilizamos el operador `:=` (dos puntos y el signo igual). Internamente, Go se encarga de declarar la variable y decidir el tipo de dato según el valor que estemos asignándole.

```
var horas := 20
```

No es necesario utilizar la palabra clave `var` o declarar el tipo de variable.

## ▼ Tipos de Datos

### ▼ Colecciones

#### ▼ Arrays

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

```
var a [2]string
```

Para asignar un valor a un array hay que especificar la posición seguida por el valor.

```
a[0] = "Hello"
a[1] = "World"
```

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)
```

La longitud de un array es parte de su tipo, por lo que no se puede cambiar el tamaño de los arrays.

#### ▼ Slices

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.

El siguiente es un slice con elementos de tipo `bool`. Veamos cómo obtener un valor de dicho slice:

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

También los slices se pueden crear con la función `make()`. Esta función 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
```

Otra forma de obtener los valores de un slice es basándonos en un rango que esté formado por dos índices, uno de inicio y otro de fin (separados con dos puntos). Esto también se puede hacer con los arrays.

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

```
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.
}
```

Tiene dos propiedades:

- 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()` y `cap()`.

### Agregar a un slice

Es común tener que agregar elementos a un slice. Para realizar esta tarea, Go nos provee de la función `append()`. Veamos cómo funciona:

```
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 el resto de los parámetros son los valores de tipo “`T`” que queremos agregar. Esta retorna un slice con todos los elementos anteriores más los nuevos.

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

### ▼ Maps

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

Podemos instanciar un map 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.

 La función `make` nos sirve para inicializarlo, pero no podremos introducir datos en la misma sentencia de inicialización.

Podemos determinar cuántos elementos clave-valor tiene un map con la función `len()`:

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

 Esta función devuelve cero para un mapa no inicializado.

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.

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

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

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}
students["Brenda"] = 19
students["Marcos"] = 22
```

Podemos actualizar el valor de un elemento específico consultando su nombre de clave:

```
var students = map[string]int{"Benjamin": 20, "Nahuel": 26}
students["Benjamin"] = 22
```

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

```
delete(students, "Benjamin")
```

## ▼ Operadores

En todo lenguaje de programación necesitamos de operadores para poder resolver problemas de programación. Repasemos qué es un operador, sus tipos y funcionalidades.

[¿Qué es un operador?](#)

## ¿Qué es un operador?

Un operador es un elemento del programa que se aplica a uno o varios operandos en una expresión o instrucción. Los operadores, junto con los operandos, forman una expresión que es una fórmula que define el cálculo de un valor.

## ¿Qué tipos de operadores existen?

Existen diferentes tipos de operadores. Go posee los siguientes operadores:

- Aritméticos
- ▼ Relacionales

Estos operadores son aquellos que nos retornan un valor de verdad. Veamos algunos ejemplos en la siguiente tabla. La columna "Ejemplo" está calculada suponiendo que  $X:=1$  e  $Y:=2$ .

Operador	Descripción	Ejemplo
$==$	Retorna verdadero cuando ambos operandos son iguales. Devuelve falso en cualquier otro caso.	$X == Y$ retorna falso
$!=$	Retorna verdadero solamente cuando los operandos son distintos.	$X != Y$ retorna verdadero
$>$	Retorna verdadero cuando el operando de la izquierda es mayor que el de la derecha.	$X > Y$ retorna falso
$<$	Retorna verdadero cuando el operando de la izquierda es menor que el de la derecha.	$X < Y$ retorna verdadero
$\geq$	Retorna verdadero cuando el operador de la izquierda es mayor o igual al de la derecha.	$X \geq Y$ retorna falso
$\leq$	Retorna verdadero cuando el operador de la izquierda es menor o igual al de la derecha	$X \leq Y$ retorna verdadero

## ▼ Lógicos

Operador	Descripción	Ejemplo
$\&\&$	Operador lógico de conjunción (AND). Compara dos valores bool o expresiones relacionales.	$A \&\& B$ resulta falso $X > 0 \&\& Y < 6$ resulta verdadero
$\ $	Operador lógico de disyunción (OR). Compara dos valores bool o expresiones relacionales.	$A \  B$ resulta verdadero $X < 0 \  Y > 6$ resulta falso
!	Operador de negación. Invierte (niega) el valor bool del operando.	$!A$ resulta falso $B$ resulta verdadero

## ▼ De asignación

Estos operadores nos permiten modificar el valor de nuestras variables durante la ejecución del programa.

Operador	Descripción	Ejemplo
$=$	Operador de asignación simple.	$X = Y + Z$ asigna a X la suma de Y y Z (ni Y ni Z se modifican)
$+=$	Operador de suma y asignación.	$X += Y$ asigna a X el valor de $X + Y$
$-=$	Operador de resta y asignación.	$X -= Y$ asigna a X el valor de $X - Y$
$*=$	Operador de multiplicación y asignación.	$X *= Y$ asigna a X el valor de $X * Y$
$/=$	Operador de división y asignación.	$X /= Y$ asigna a X el valor de $X / Y$
$\%=$	Operador de módulo y asignación.	$X \%= Y$ asigna a X el valor de $X \% Y$

## ▼ De dirección

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

## Precedencia de operadores

La precedencia de un operador indica que tan "estrechamente" se unen dos expresiones juntas. Por ejemplo, en la expresión  $1 + 5 * 3$ , la respuesta es 16 y no 18 porque el operador de multiplicación ("\*") tiene una precedencia mayor que el operador de adición ("+"). Los paréntesis pueden ser usados para forzar la precedencia, si es necesario. Por ejemplo:  $(1 + 5) * 3$ , se evalúa como 18.



## fmt

Al llamar una función del package fmt, Go automáticamente importará el package para ser usado.

### fmt.Println(a ...interface{}) (n int, err error)

Esta función nos permite imprimir por consola lo que le pasemos por parámetro y, a su vez, hará un salto de línea.

### fmt.Printf(format string, a ...interface{}) (n int, err error)

Es similar a la función Println(), pero con una diferencia: nos permite utilizar los caracteres de conversión. Ejemplo: %s (string), %d (enteros).

### fmt.Sprintf(format string, a ...interface{}) string

Genera la misma sintaxis que Printf(), pero con la diferencia de que no muestra en pantalla, sino que genera un string o cadena de caracteres.

### fmt.Scan(a ...interface{}) (n int, err error)

Leerá el texto de la entrada estándar (stdin) hasta que encuentre el primer espacio o salto de línea.

### fmt.Scanf(format string, a ...interface{}) (n int, err error)

Es como Scan(), pero para guardar múltiples argumentos, separados por espacios.

## ▼ Estructuras de Control

### ▼ If/else

La instrucción if nos permite controlar el flujo de instrucciones que seguirá nuestro programa. Esto se logra analizando una cierta condición que, si se cumple, se ejecuta la porción de código determinada. Si no se cumple, este código no se ejecutará.

```
if condición {  
    // instrucciones  
}
```

La instrucción if/else nos permite ejecutar una instrucción si la condición es verdadera (true), y otra si es falsa (false).

```
if condición {  
    // instrucciones si la condición es verdadera  
} else {  
    // instrucciones si la condición es falsa  
}
```

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

Primero analiza la condición-1. Si es true, se ejecuta un conjunto de instrucciones. De lo contrario, se analiza la condición-2. Si se cumple, se ejecuta otro conjunto de instrucciones. De la misma manera, para condiciones-n. Si no se cumple ninguna de las condiciones, se ejecutan las instrucciones dentro del else.

```
if condición_1 {  
    // instrucciones si la condición-1 es verdadera  
} else if condición_2 {  
    // instrucciones si la condición-2 es verdadera  
} else {  
    // instrucciones si todas las condiciones son falsas  
}
```

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

```
if var declaración; condición {  
    // instrucciones si la condición es verdadera  
}
```

## Switch

La estructura de control switch nos permite evaluar múltiples casos condicionales y ejecutar instrucciones sobre la base de estos. Este es una mejor alternativa que utilizar if/else if/else anidados.

```
switch expresion {  
    case condicion_1:  
        // instrucciones si se cumple condicion_1  
    case condicion_n:  
        // instrucciones si se cumple condicion_n  
    default:  
        // instrucciones si no se cumple ninguna condición  
}
```

💡 Switch se compone de tres partes principales

La primera parte se compone de la palabra reservada switch y una expresión la cual vamos a evaluar. Las instrucciones que se ejecutarán, se basarán en el valor de estas expresiones dentro de los casos a evaluar.

Luego tenemos las condiciones. Estas serán comparadas con el valor de las expresiones y, si encuentra una coincidencia, se ejecuta el bloque de instrucciones que contiene.

Finalmente, tenemos la palabra reservada default que nos permite definir un bloque de instrucciones en caso de que no se cumpla ninguno de los casos anteriores.

💡 Este debe ser el último bloque de los posibles casos.

### ▼ Switch sin condición

Podemos utilizar switch sin condición, agregando directamente la condición en el case:

```
var expresion  
switch {  
    case expresion == condicion_1:  
        // instrucciones si se cumple condicion_1  
    case expresion == condicion_2:  
        // instrucciones si se cumple condicion_2  
    default:  
        // instrucciones sin caso  
}
```

### ▼ Switch con múltiples casos

Los casos pueden tener múltiples valores separados por comas que indican los valores que cumplen la condición:

```
switch expresion {  
    case condicion_1, condicion_2, condicion_n:  
        // instrucciones si la condicion_1, condicion_2,  
        // ..., condicion_n  
    default:  
        // instrucciones si no se cumple ninguna condición  
}
```

### ▼ Switch con declaración corta

Tal como es posible hacerlo con if, switch también nos permite declarar una variable para usarla dentro de la sentencia, y luego evaluarla.

```
switch var expresion; expresion{  
    case condicion_1:  
        // instrucciones si cumple condicion_1  
    case condicion_2:  
        // instrucciones si cumple condicion_2  
    default:  
        // instrucciones sin caso  
}
```

### ▼ Switch con fallthrough

Dentro de los casos también podemos utilizar la palabra reservada fallthrough, que indica que se ejecute las instrucciones del caso siguiente:

```
switch {  
    case condicion_1:  
        // instrucciones si cumple condicion_1  
    case condicion_2:  
        // instrucciones si cumple condicion_2  
        fallthrough  
    case condicion_3:  
        // instrucciones si cumple condicion_3 o condicion_2  
    default:  
        // instrucciones sin caso
```

```
}
```

## ▼ For

El bucle for nos permite ejecutar un bloque de código repetidamente. Por lo general, se utiliza para iterar sobre una secuencia de datos (slice, array, map o string).

Para crear bucles en Go solo existe la palabra reservada for, así podemos formar cuatro tipos de iteraciones:

### ▼ Standard for

Esta es la estructura más común de un bucle for. Go tiene una sintaxis estándar compuesta por tres componentes y estos son: declaración, condición y post declaración.

```
for i := 0; i < 100; i++ {
    sum += i
}
```

- Declaración: declara una variable y la expone dentro del scope del bucle.
- Condición: si la condición se cumple, ejecuta el código dentro del bucle; de lo contrario, termina.
- Post declaración: se ejecuta la post declaración, comúnmente se utiliza para modificar el valor de la variable declarada.

### ▼ Bucle while

El bucle while nos permite ejecutar un bloque de código mientras una condición se cumpla. A diferencia del standard for de tres componentes, en este caso, solo tenemos la condición.

```
sum := 1
for sum < 10 {
    sum += sum
}
```

¡Ojo! Dentro del scope del bucle debemos actualizar la condición, si no este se ejecutará indefinidamente.

### ▼ Bucle infinito

Para crear un bucle infinito basta con definir un bucle while con una condición que sea siempre verdadera. También, podemos obtener el mismo resultado si no colocamos una condición.

```
sum := 0
for {
    sum++
}
```

### ▼ Bucle range

La palabra reservada range itera por los elementos de estructuras de datos, retornando siempre dos variables. Tenemos:

- Array or slice: primero el índice, segundo el elemento de la lista.
- String: primero el índice, segundo es la representación del carácter en rune int.
- Map: primero la key, segundo el value del par clave-valor.
- Channel: primero el elemento del canal, segundo está vacío.

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

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

El siguiente ejemplo solo imprime los números impares. Cuando el resto de la división por 2 es 0, se trata de un número par. Por lo que entra en la condición y saltea la iteración.

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

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

```

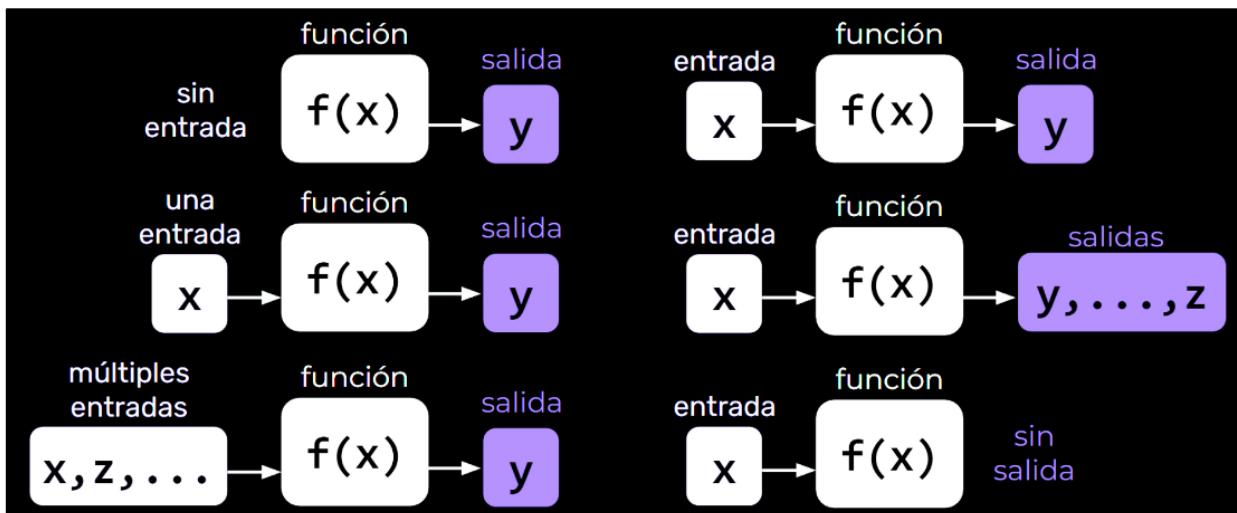
sum := 0
for {
    sum++
    if sum >= 1000 {
        break
    }
}
fmt.Println(sum)
//output: 1000

```

## ▼ Funciones

Una función es un bloque de código, reutilizable, que realiza una operación específica.

Una función, en general, puede recibir ninguna, una o muchas entradas y puede retornar uno, muchos o ningún valor.



## ▼ Composición

```

func sum(a int, b int) int {
    sum := a + b
    return sum
}

```

- **func:** palabra reservada, indica el inicio de una declaración de función.
- **sum:** nombre, si inicia con minúscula, solo estará disponible en el paquete donde se definió.
- **(a int, b int):** parámetros, primero se coloca el nombre del parámetro (a) seguido del tipo de variable (int). Esta lista se encierra en paréntesis (). Si los colocamos vacíos, indicamos que la función no recibe entradas.
- **int:** tipo de retorno, si está presente, es necesario colocar un return. Si no colocamos el tipo de retorno, indicamos que la función no tiene valores de salida.
- **{ sum := a + b return sum }:** cuerpo, es el bloque de código encargado de realizar la operación que cumple la función. Este bloque se encierra entre llaves {}. Esto define el scope de la función que determina el contexto de ejecución de nuestra función. Es decir, todas las variables que se definan dentro de este espacio solo estarán disponibles dentro de él.

Para llamar una función basta con poner su nombre seguido de paréntesis (). En caso de que la función reciba parámetros, deben ir entre ellos. Estos serán asignados en el orden definido en la declaración. Si la función retorna un valor al momento de llamarla, debemos guardar su resultado en una variable.

```

func main() {
    c := sum(5, 5)
    fmt.Println(c)
}

```

## ▼ Elipsis

Go nos proporciona la notación de puntos suspensivos (Ellipsis). Esta permite que nuestras funciones reciban una cantidad dinámica de parámetros.

Para utilizar esta notación, vamos a definir una función de la siguiente manera:

```

func miFuncion(valores ...float64) float64

```

Al momento de llamar a esta función, podremos pasarle la cantidad de valores que queramos, siempre del mismo tipo de dato. Y nuestra función recibirá los parámetros como si fueran un array.

```
miFuncion(2, 3, 2, 1, 2, 3, 4, 5, 6)
```

## ▼ Ejemplos

### ▼ Uno

Vamos a crear una función que reciba, mediante la notación de puntos suspensivos, un número variable de valores numéricos y devolveremos la sumatoria de todos ellos.

```
func suma(values ...float64) float64 {
    var resultado float64
    for _, value := range values {
        resultado += value
    }
    return resultado
}
```

Al llamar a esta función, le podemos pasar todos los valores que queramos sumar.

```
suma(2, 3, 2, 1, 2, 3, 4, 5, 6)
```

También podemos pasar otros parámetros adicionales, pero —en ese caso— el parámetro de notación de puntos suspensivos siempre tiene que estar al final.

```
func miFuncion(valor1 string, valor2 string, valores ...float64)
```

### ▼ Dos

Vamos a realizar un ejemplo más interesante que el anterior. Crearemos una función a la cual le indicaremos la operación a realizar y todos los números a los que se le realizará dicha operación.

Por ejemplo: le indicaremos a la función que queremos realizar una suma y le pasaremos todos los valores que queramos sumar.

```
operacionAritmetica(Suma, 2, 3, 2, 1, 2, 3, 4, 5, 6)
```

Declararemos las constantes con las operaciones a realizar:

```
const (
    Suma   = "+"
    Resta  = "-"
    Multip = "*"
    Divis  = "/"
)
```

Luego, creamos las funciones que realizarán las operaciones.

```
func opSuma(valor1, valor2 float64) float64 {
    return valor1 + valor2
}

func opResta(valor1, valor2 float64) float64 {
    return valor1 - valor2
}

func opMultip(valor1, valor2 float64) float64 {
    return valor1 * valor2
}

func opDivis(valor1, valor2 float64) {
    if valor2 == 0 {
        return 0
    }
    return valor1 / valor2
}
```

También crearemos la función que se encargará de recibir la operación a realizar y los valores a los cuales se le aplicará la operación.

Por cada operación, llamaremos a una función que reciba los valores y la función que vamos a ejecutar por ese operador.

```
func operacionAritmetica(operator string, valores ...float64) float64 {
    switch operator {
        case Suma:
            return orquestadorOperaciones(valores, opSuma)
        case Resta:
            return orquestadorOperaciones(valores, opResta)
        case Multip:
            return orquestadorOperaciones(valores, opMultip)
        case Divis:
            return orquestadorOperaciones(valores, opDivis)
    }
}
```

```

        case Resta:
            return orquestadorOperaciones(valores, opResta)
        case Multip:
            return orquestadorOperaciones(valores, opMultip)
        case Divis:
            return orquestadorOperaciones(valores, opDivis)
    }

    return 0
}

```

Crearemos esa función que se encargará de orquestar las operaciones:

```

func orquestadorOperaciones(valores []float64, operacion func(value1, value2 float64) float64 {
    var resultado float64
    for i, valor := range valores {
        if i == 0 {
            resultado = valor
        } else {
            resultado = operacion(resultado, valor)
        }
    }

    return resultado
}

```

Por último, probamos nuestra aplicación pasándole la operación que queramos realizar.

```

func main() {
    fmt.Println(operacionAritmetica(suma, 2, 3, 2, 1, 2, 3, 4, 5, 6))
}

```

## ▼ Retorno

Para definir a una función que esperamos que nos devuelva un valor, tenemos que indicarle el tipo de dato que esperamos al final de la función. En este caso, creamos una función que reciba dos parámetros y nos devuelva la sumatoria de ellos.

```

func sum(value1, value2 float64) float64 {
    return value1 + value2
}

```

Luego de declarar nuestra función con retorno de un valor, a ese valor se lo asignamos a una variable. En este caso, el resultado de la suma se lo asignamos a la variable result para luego mostrarla por consola.

```

func main() {
    result := sum(4, 5)
    fmt.Println(result)
}

```

## ▼ Ejemplo

Vamos a realizar un ejemplo un poquito más complejo: una función a la cual le pasaremos dos valores y le indicaremos qué operación queremos realizar.

Para ello, definiremos cuatro constantes con las operaciones que queremos realizar, que son las cuatro variables matemáticas principales.

```

const (
    Suma    = "+"
    Resta   = "-"
    Multip = "*"
    Divis  = "/"
)

```

Generamos una función que se encargará de orquestar el tipo de operación que le definiremos por parámetro.

```

func operacionAritmetica(valor1, valor2 float64, operador string) float64 {
    switch operador {
    case Suma:
        return valor1 + valor2
    case Resta:
        return valor1 - valor2
    case Multip:
        return valor1 * valor2
    case Divis:
        return valor1 / valor2
    }
}

```

```

        if valor2 != 0 {
            return valor1 / valor2
        }
    }
    return 0
}

```

Y, por último, llamaremos a esa función y le indicaremos —con las constantes que definimos— qué operación queremos realizar.

```

func main() {
    fmt.Println(operacionAritmetica(6, 2, Suma))
    fmt.Println(operacionAritmetica(6, 2, Resta))
    fmt.Println(operacionAritmetica(6, 2, Multip))
    fmt.Println(operacionAritmetica(6, 2, Divis))
}

```

## ▼ Multiretorno

Para empezar tenemos que indicar los tipos de datos de los valores que retornarán, separados por coma y entre paréntesis.

```
func miFuncion(valor1, valor2 float64) (float64, string, int, bool)
```

Luego, vamos a generar una función que nos devuelva los cuatro resultados de las operaciones aritméticas: suma, resta, multiplicación y división.

```

func operaciones(valor1, valor2 float64) (float64, float64, float64, float64) {
    suma := valor1 + valor2
    resta := valor1 - valor2
    multip := valor1 * valor2
    var divis float64

    if valor2 != 0 {
        divis = valor1 / valor2
    }

    return suma, resta, multip, divis
}

```

Al llamar a nuestra función, debemos recibir todos los valores que retorna.

```

func main() {
    s, r, m, d := operaciones(6, 2)

    fmt.Println("Suma:\t\t", s)
    fmt.Println("Resta:\t\t", r)
    fmt.Println("Multiplicación:\t", m)
    fmt.Println("División:\t", d)
}

```

En Go el retorno de multivalores se utiliza por lo general cuando necesitamos retornar un valor y un error, y necesitamos validar si se produjo un error o no.

Para ello vamos a realizar un ejemplo de una división que nos retorna error en caso de que el divisor sea cero. Utilizaremos el paquete errors que nos permite trabajar con la interfaz de error.

```

import (
    "errors"
)

```

Implementamos nuestra función división y validamos si el divisor es cero. En caso de que lo sea, retornará un error; de lo contrario, realizará la división.

```

func division(dividendo, divisor float64) (float64, error) {

    if divisor == 0 {
        return 0, errors.New("El divisor no puede ser cero")
    }

    return dividendo/divisor, nil
}

```

Ejecutamos nuestra función main() y validamos si la operación fue realizada correctamente.

```
func main() {
```

```

res, err := division(2, 0)

if err != nil {
    // Si hubo error
} else {
    // Si terminó correctamente
}
}

```

## ▼ Valores Nombrados

Para esto, debemos definir en la función no solo el tipo de dato a retornar, sino también el nombre de la variable.

```
func operaciones(valor1, valor2 float64) (suma float64, resta float64, multip float64, divis float64)
```

Dentro de la función, tenemos que almacenar el resultado de las operaciones en dichas variables y luego hacer un return.

```

func operaciones(valor1, valor2 float64) (suma float64, resta float64, multip float64, divis float64) {
    suma = valor1 + valor2
    resta = valor1 - valor2
    multip = valor1 * valor2

    if valor2 != 0 {
        divis = valor1 / valor2
    }

    return
}

```

 De este modo, Go retornará los valores que guardamos en las variables que definimos en la función.

## ▼ Retorno de Funciones

También podemos implementar una función que devuelva otra función.

Para ello debemos indicarle los parámetros y los tipos de datos que retorne dicha función.

En este caso miFuncion nos devolverá otra función que recibe dos parámetros y devuelve un valor en punto flotante.

```
func miFuncion(valor string) func(valor1, valor2 float64) float64
```

Veamos un ejemplo de una función a la cual le indicaremos una operación y nos devolverá una función que realice la operación pasándole dos valores numéricos como parámetros.

### ▼ Ejemplo

Crearemos una función para cada operación. Y cada una de ellas se encargará de una de las operaciones aritméticas: suma, resta, multiplicación y división.

```

func opSuma(valor1, valor2 float64) float64 {
    return valor1 + valor2
}

func opResta(valor1, valor2 float64) float64 {
    return valor1 - valor2
}

func opMultip(valor1, valor2 float64) float64 {
    return valor1 * valor2
}

func opDivis(valor1, valor2 float64) float64 {
    if valor2 == 0 {
        return 0
    }
    return valor1 / valor2
}

```

Generamos una función que se encargue de orquestar las funciones que realizarán las operaciones.

```

func operacionAritmetica(operador string) func(valor1, valor2 float64) float64 {
    switch operador {
    case "Suma":
        return opSuma
    case "Resta":
        return opResta
    case "Multip":
        return opMultip
    case "Divis":
        return opDivis
    }
}

```

```

        return opMultiplicacion
    case "Divis":
        return opDivision
    }

    return nil
}

```

Instanciamos la función indicando la operación a realizar.

Nos devolverá una función a la que le pasaremos los dos valores con los cuales queremos realizar la operación.

```

func main() {
    oper := operacionAritmetica("Suma")
    r := oper(2, 5)
    fmt.Println(r)
}

```

## ▼ Estructuras

Una estructura es un conjunto tipado de campos de datos.

Por ejemplo, podemos definir una estructura "persona" y en ella tener valores como edad, peso, género, profesión, etc. Estas son útiles para agrupar datos, como formar registros personalizados.

Una estructura consta de tipos integrados y definidos por el usuario (la estructura en sí es un tipo definido por el usuario).

El esqueleto básico de una estructura:

```

type nombre struct {
    campo1 tipoDeDatos1
    campo2 tipoDeDatos2
}

```

**type** y **struct** son palabras clave, mientras que la estructura contiene varios campos con su tipo de datos definido.

Definimos una estructura de la siguiente manera: determinamos sus campos, seguido de un espacio y el tipo de dato. Para separar cada campo utilizamos un salto de línea. Si la primera letra es mayúscula, significa que es accesible desde otro paquete, y aplica la misma lógica para los atributos.

```

type Persona struct {
    Nombre     string
    Genero    string
    Edad       int
    Profesion string
}

```

Para instanciar una estructura, podemos utilizar distintas formas:

- Indicar todos los valores que queremos que tengan los campos:

```
p1 := Persona{"Celeste", "Mujer", 34, "Ingeniera"}
```

- Definir los valores para el campo que corresponda. De esta manera podemos no asignar valores a todos los campos y, de ser así, los valores quedarán por defecto según el tipo de dato.

```

p2 := Persona{
    Nombre:      "Nahuel",
    Genero:     "Hombre",
    Edad:        30,
    Profesion:  "Ingeniero",
}

```

Para acceder a un campo de la estructura procedemos de la siguiente manera:

```
p2.Edad
```

Podemos asignar o modificar un valor a un campo de la estructura de la siguiente manera:

```
p2.Edad = 33
```

También podemos definir una estructura vacía e ir asignando los valores.

```
var p3 Persona
p3.Nombre = "Ulises"
p3.Edad = 15
```

## ▼ Estructura en Estructura

Podemos utilizar las estructuras como un tipo de dato. Por ende, podríamos tener estructuras como campos dentro de otra estructura. A eso se lo denomina “composición”. Por ejemplo, podemos tener una estructura gustos dentro de nuestra estructura persona. Para eso debemos declarar nuestra estructura gustos.

```
type Preferencias struct {
    Comidas string
    Peliculas string
    Series string
    Animes string
    Deportes string
}
```

Asignaremos un campo de tipo gustos a nuestra estructura persona:

```
type Persona struct {
    Nombre string
    Género string
    Edad int
    Profesion string
    Peso float64
    Gustos Preferencias
}
```

Hacemos lo siguiente para instanciar nuestra estructura:

```
p1 := Persona{"Celeste", "Mujer", 34, "Ingeniera", 65.5, Preferencias{"pollo", "Titanic", "", "", ""}}
```

También podemos instanciarla haciendo referencia a cada campo:

```
p2 := Persona{
    Nombre: "Nahuel",
    Genero: "Hombre",
    Edad: 30,
    Profesion: "Ingeniero",
    Peso: 77,
    Gustos: Preferencias{
        Comidas: "asado, pollo",
        Peliculas: "Coco",
        Animes: "Shingeki no Kyojin",
    },
}
```

De la misma forma, para acceder a un valor o modificarlo dentro de la estructura “gustos” desde “persona”:

```
fmt.Println(p2.Gustos.Animes)
p2.Gustos.Deportes = "fútbol"
```

O podríamos agregarle directamente la estructura completa:

```
p3 := Persona{}
p3.Nombre = "Ulises"
p3.Edad = 15
p3.Gustos = Preferencias{Comidas: "verduras", Películas: "Entrenando a mi dragón"}
```

## ▼ Etiquetas

Cuando trabajamos con estructuras en Go es posible definir etiquetas. Estas son una forma de adjuntar información adicional a un campo de estructura.

Las etiquetas en una estructura en Go son anotaciones que aparecen después del tipo, en una declaración de una estructura. Cada etiqueta se compone de cadenas cortas asociadas con algún valor correspondiente. Una etiqueta de struct tiene este aspecto y se encuentra entre backticks (`):

```
type MiEstructura struct {
    Campo1 string `miEtiqueta:"valor"`
    Campo2 string `miEtiqueta:"valor"`
    Campo3 string `miEtiqueta:"valor"`
}
```

El codificador JSON de la biblioteca estándar utiliza etiquetas de una estructura como anotaciones. Estas le indican la forma en que desearía nombrar sus campos en el resultado de JSON. Los mecanismos de codificación y decodificación de JSON pueden encontrarse en el paquete encoding/json.

```
type Persona struct {
    PrimerNombre string `json:"primer_nombre"`
    Apellido     string `json:"apellido"`
    Telefono     string `json:"telefono"`
    Direccion    string `json:"direccion"`
}
```

Por lo general, queremos suprimir los campos de salida que no están configurados en JSON. Debido a que todos los tipos de Go tienen un valor "cero", si queremos omitir cualquier resultado cuyo valor sea "cero", debemos incluir en la etiqueta el atributoomitempty.

```
type Persona struct {
    PrimerNombre string `json:"primer_nombre"`
    Apellido     string `json:"apellido"`
    Telefono     string `json:"telefono"`
    Direccion    string `json:"direccion,omitempty"`
}
```

Algunos campos deben exportarse desde las estructuras para que otros paquetes puedan interactuar correctamente con el tipo. Sin embargo, la naturaleza de esos campos puede ser sensible, de modo que en estas circunstancias nos gustaría que el codificador de JSON ignore el campo por completo, incluso cuando esté establecido. Esto se hace usando el valor especial "-" como argumento del valor para una etiqueta de estructura JSON.

```
type Usuario struct {
    Nombre string `json:"nombre"`
    Apellido string `json:"apellido"`
    Password string `json:"-"`
}
```

## ▼ Métodos

Un método es una función con un argumento receptor especial. El receptor (en inglés, receiver) aparece en su propia lista de argumentos entre la palabra clave func y el nombre del método. En Go se pueden definir métodos en los tipos de datos.

Para declarar métodos necesitamos exclusivamente una estructura. Declararemos entonces una estructura Circulo y en ella agregaremos un campo para almacenar el radio.

```
type Circulo struct {
    radio float64
}
```

En Go, definir un método de una estructura es similar a declarar una función, con algunas diferencias importantes. Debemos agregar —entre la palabra reservada func y el nombre del método— a qué estructura corresponde, de la siguiente manera:

```
func (v MiEstructura) metodo(){ }
```

De esta manera, definimos la variable que vamos a utilizar para manipular nuestra estructura desde el método (en el ejemplo, la variable v), y la estructura a la que corresponde.

Definimos nuestro primer método de la estructura Circulo con la variable c, que pertenece a la estructura Circulo, y así podremos acceder a sus variables.

```
func (c Circulo) area() float64 {
    return math.Pi * c.radio * c.radio
}
```

Declaremos el método perim:

```
func (c Circulo) perim() float64 {
    return 2 * math.Pi * c.radio
}
```

💡 `math` es un paquete que nos proporciona Go para realizar cálculos matemáticos más complejos. En este caso, para obtener el valor de Pi.

## ▼ Composición

En otros lenguajes existe el concepto de herencia. Este consiste en tener una clase padre y sus clases hijas. La clase padre es la que transmite su código a las clases hijas.

El concepto de herencia no existe en Go, pero tenemos una composición que utiliza la estructura padre como campo en nuestras estructuras hijas. Esto se conoce como [embedding structs](#).

💡 El propósito de la composición en Go es poder crear programas más grandes a partir de piezas más pequeñas. Esto nos ayuda a diseñar diversos tipos de datos sobre los cuales implementar distintos comportamientos.

Podemos imaginar la composición como una receta de cocina, que está compuesta por otras recetas.

#### ▼ Ejemplo

Declaramos nuestra clase padre Vehículo y en ella agregaremos los campos km y tiempo:

```
type Vehiculo struct {
    km     float64
    tiempo float64
}
```

Declararemos un método para nuestra clase Vehículo que nos imprima en pantalla el valor de sus campos:

```
func (v Vehiculo) detalle() {
    fmt.Printf("km:\t%f\n\ttiempo:\t%f\n", v.km, v.tiempo)
}
```

Declaremos la estructura Auto. En ella, agreguemos un campo de tipo Vehículo. Aquí estamos embebiendo la estructura.

```
type Auto struct {
    v Vehiculo
}
```

Agreguemos un método que reciba tiempo en minutos y se encargue de realizar el cálculo de distancia basándose en 100 km/h:

```
func (a *Auto) Correr(minutos int) {
    a.v.tiempo = float64(minutos) / 60
    a.v.km = a.v.tiempo * 100
}
```

Ahora, el método Detalle que llame al método de la clase padre:

```
func (a *Auto) Detalle() {
    fmt.Println("\nV:\tAuto")
    a.v.detalle()
}
```

## ▼ Punteros

Un puntero es un tipo de dato cuyo valor es una dirección de memoria que se refiere o "apunta a" otra variable

Para crearlos, colocamos el operador "\*" antes de definir el tipo de dato que necesitamos almacenar en esa dirección de memoria.

```
var p *int
```

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

Otras formas de crear punteros:

- Utilizando la función new() la cual recibe como argumento un tipo de dato.
- A través del shorthand de declaración de variables ":=".

```
var p1 *float64
var p2 = new(float64)
var v float64
p3 := &v
```

## ▼ 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 = 50
```

```
fmt.Println("La dirección de memoria de la variable v es: ", &v)
```

Podemos ver que la variable v de tipo entero tiene el valor "50" y se almacena en una dirección de memoria que tendrá un formato del estilo 0x70158811.

#### ▼ Operador de desreferenciación

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 = 50
var p *int
// Hacemos que el puntero p, almacene (apunte a) la dirección de memoria de la variable v.
p = &v
fmt.Printf("El puntero p apunta a la dirección de memoria: %v \n",p)
fmt.Printf("Al desreferenciar el puntero p obtengo el valor de la variable a la cual apunta, es decir, el valor de v: %d \n",*p)
```

El código anterior produce el siguiente resultado:

```
El puntero p referencia a la dirección de memoria: 0x70158811
Al desreferenciar el puntero p, obtengo el valor: 50
```

Podemos ver que el puntero p referencia a la dirección de memoria 0x70158811, y para obtener el valor "50" almacenado en esta dirección, hacemos uso del operador de desreferenciación \*p.

► Ejemplo

#### ▼ Interfaces

Una interfaz es una forma de definir métodos que deben ser utilizados, pero sin definirlos.

Las interfaces son utilizadas para brindar modularidad al lenguaje.

?

¿Qué pasará si queremos generar más figuras utilizando nuestra función?

Aquí es donde entran en juego las interfaces. Estas nos permiten implementar el mismo comportamiento a diferentes objetos.

?

¿Qué pasará si queremos reutilizar nuestra función para poder implementar varias figuras geométricas?

En este caso tendremos que crear una función que retorne una interfaz que pueda implementar todos nuestros objetos geométricos.

#### ▼ Ejemplos

▼ Uno

Generamos una estructura circle y las funciones que mostrarán el área y el perímetro de la figura:

```
type circle struct {
    radius float64
}

func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}

func (c circle) perim() float64 {
    return 2 * math.Pi * c.radius
}
```

Creamos una función details que imprima el área y el perímetro que generamos para dicho objeto:

```
func details(c circle) {
    fmt.Println(c)
    fmt.Println(c.area())
    fmt.Println(c.perim())
}
```

Y ejecutaremos la función:

```
func main() {
    c := circle{radius: 5}
    details(c)
}
```

!! Continua en ejemplo dos

## ▼ Dos

A continuación, definimos nuestra interfaz geometry que contiene dos métodos que adoptarán nuestros objetos:

```
type geometry interface {
    area() float64
    perim() float64
}
```

Generamos otro objeto geométrico. En este caso, un rectángulo que —lógicamente— tenga los mismos métodos:

```
type rect struct {
    width, height float64
}

func (r rect) area() float64 {
    return r.width * r.height
}

func (r rect) perim() float64 {
    return 2*r.width + 2*r.height
}
```

Modificaremos nuestra función details para que, en lugar de recibir un círculo, reciba una figura geométrica:

```
func details(g geometry) {
    fmt.Println(g)
    fmt.Println(g.area())
    fmt.Println(g.perim())
}
```

De esta forma podemos seguir agregando figuras geométricas sin necesidad de modificar nuestra función:

```
func main() {
    c := newCircle(2)
    fmt.Println(c.area())
    fmt.Println(c.perim())
}
```

## ▼ Tres

En el siguiente ejemplo, creamos una función que nos genere el objeto:

```
func newCircle(values float64) circle {
    return circle(radius: values)
}
```

Ejecutamos el main del programa:

```
func main() {
    c := newCircle(2)
    fmt.Println(c.area())
    fmt.Println(c.perim())
}
```

!! Continua en ejemplo cuatro

## ▼ Cuatro

Vamos a reemplazar nuestra función newCircle por newGeometry y le pasaremos dos constantes que definimos para especificar cuál es el objeto que generamos:

```
const (
    rectType = "RECT"
    circleType = "CIRCLE"
)
func newGeometry(geoType string, values ...float64) geometry {
    switch geoType {
        case rectType:
            return rect(width: values[0], height: values[1])
        case circleType:
            return circle(radius: values[0])
    }
}
```

```
    return nil
}
```

Implementamos en nuestro main y corremos el programa:

```
func main() {
    r := newGeometry(rectType, 2, 3)
    fmt.Println(r.area())
    fmt.Println(r.perim())
    c := newGeometry(circleType, 2)
    fmt.Println(c.area())
    fmt.Println(c.perim())
}
```

## ▼ Interfaces Vacías

Son aquellas interfaces que no tienen métodos declarados.

La utilidad de estas interfaces es proveernos un tipo de datos "comodín". Es decir, almacenar valores que sean de un tipo de datos desconocido, o que pueda variar dependiendo el flujo del programa.

¿Cómo se declara una variable con este tipo?

```
var miVariable interface{}
```

¿Cómo funciona?

Como vimos anteriormente, una interfaz define el conjunto mínimo de métodos que un tipo de datos debe implementar para poder ser considerado como implementador de dicha interfaz. Por lo tanto, todos los tipos de datos son considerados implementadores de la interfaz vacía, porque implementan al menos cero métodos.

Ejemplo de un uso de interfaz vacía

```
type ListaHeterogenea struct {
    Data []interface{}
}

func main() {
    l := ListaHeterogenea{}
    l.Data = append(l.Data, 1)
    l.Data = append(l.Data, "hola")
    l.Data = append(l.Data, true)

    fmt.Printf("%v\n", l.Data)
}
```

```
> $ go run interfaces_vacias.go
[1 hola true]
```

## ▼ Type Assertion

La aserción de tipos provee acceso al tipo de datos exacto que está abstraído por una interfaz.

```
var i interface{} = "hello"

s := i.(string)
fmt.Println(s)

s, ok := i.(string)
fmt.Println(s, ok)

f, ok := i.(float64)
fmt.Println(f, ok)

f = i.(float64) // panic
fmt.Println(f)
```

## ▼ Errores

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

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**.

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

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

A diferencia de otros lenguajes —como 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 (por ejemplo: try-catch-finally).
Funciones propias del lenguaje para el manejo de errores.	Excepciones.

#### ▼ ¿Cómo crear y personalizar nuestros errores?

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

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

💡 Estas tres alternativas son muy parecidas entre sí. A pesar de eso, nos enfocaremos en la librería **errors**, ya que es la más utilizada por la comunidad.

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 podemos 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.

errors.New("mensaje")

**Función “New()”**  
Pertenece al package **errors** importado.

#### ▼ Ejemplo

Veamos un ejemplo paso a paso de cómo implementar la función New() del paquete errors.

Primero: definimos nuestro package main e importamos los packages fmt y errors.

```
package main

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

Después, 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.")
}
```

## ▼ Paquete errors

### ▼ Cadenas de errores

En Go se puede envolver un error dentro de otro. Esto forma una jerarquía de errores, donde una instancia de error envuelve a otra y esta —a su vez— puede estar envuelta en otra.

En definitiva, los errores en Go forman cadenas de errores. Para envolver un error en otro utilizamos la función fmt.Errorf() junto con la directiva %w.

```
error_1 := fmt.Errorf("first error")
error_2 := fmt.Errorf("second error: %w", error_1)
```

Como vimos anteriormente, para manejar errores en nuestros programas, Go nos provee un paquete de forma nativa, llamado errors. Este contiene cuatro funciones:

### ▼ New()

Esta función es la más simple de todas. Si bien ya la vimos en detalle en el recurso anterior, recordemos que toma como argumento una cadena de caracteres (text) y retorna dicha cadena como una variable de tipo error:

```
func New(text string) error
err := errors.New("new error")
```

### ▼ Unwrap()

Esta función nos permite desenvolver los errores dentro de la cadena err. Toma el último error de la cadena y comprueba si este contiene otro error. En tal caso, lo retorna; si no devuelve nil.

```
func Unwrap(err error) error
```

1. Creamos error\_1 y luego creamos un error\_2 que envuelve al error\_1.
2. Luego, llamamos a la función Unwrap() y le pasamos como argumento error\_2. Como vimos, nos retorna el error que envuelve.
3. Si comparamos el error desenvuelto (err) con error\_1, veremos que son iguales.

```
error_1 := fmt.Errorf("first error")
error_2 := fmt.Errorf("second error: %w", error_1)

err := errors.Unwrap(error_2)
if err == error_1 {
    fmt.Println("same error")
}
```

### ▼ Is()

La función Is() recibe como argumentos dos variables de tipo error. Estas son: err y target. Si encuentra coincidencia dentro de la cadena de errores de err con el error target, devuelve true.

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

Esta función la utilizamos cuando necesitamos saber si —dentro de la cadena de errores— ocurrió uno exactamente igual a target. Con esta función comparamos error con error directamente.

1. Intentamos abrir un archivo que no existe con la función Open(). Esto genera un error que asignamos a la variable err.
2. Definimos un variable del tipo error (nonExist) que será igual a un fs.ErrNotExist. Esto indica que un archivo no existe.
3. Utilizamos la función Is() que realiza la comparación de tipos de error. Si encuentra coincidencia, devuelve true.

```
_ , err := os.Open("not_exist.txt")  
  
var nonExist error = fs.ErrNotExist  
  
if errors.Is(err, nonExist) {  
    fmt.Println("The file does not exist")  
}
```

#### ▼ As()

La función As() recibe dos argumentos: una variable del tipo err (error) y un target (interfaz vacía) que nos permite pasarle un puntero a un tipo error, y retorna una variable bool.

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

Lo que hace la función es comparar —dentro de la cadena de errores de err— con un puntero a un tipo de errores. Cuando encuentra una primera coincidencia, asigna a target el error y devuelve true.

1. Intentamos abrir un archivo que no existe con la función Open(). Esto genera un error que asignamos a la variable err.
2. Definimos un variable a un puntero de un error tipo PathError, el cual nos indica qué ocurrió relacionado con la lectura de un archivo.
3. Utilizamos la función As() que realiza la comparación de tipos de error. Si encuentra coincidencia, devuelve true.

```
_ , err := os.Open("not_exist.txt")  
  
var pathError *fs.PathError  
  
if errors.As(err, &pathError) {  
    fmt.Println("Path error", pathError.Path)  
}
```

## ▼ Retorno de Errores

En algún momento de nuestras vidas existirá la circunstancia de tomar una decisión buena o mala, y en la programación es igual. Creamos funciones que esperamos devuelvan los datos procesados; sin embargo, existe la posibilidad de un fallo. Es por este motivo que siempre debemos evaluar el camino exitoso y los posibles defectos.

En este punto aprenderemos cómo evaluar fallas para retornar los errores correspondientes y cómo los errores influyen en este proceso.

#### ▼ Funciones Multireturnos

##### ❓ ¿Cuál es el rol de las funciones multireturno en el manejo de errores?

Aquí es donde entran en juego las interfaces. Estas nos permiten implementar el mismo comportamiento a diferentes objetos.

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

Para crear una función que devuelva más de un valor —incluyendo el retorno de un error—, enumeramos los tipos de cada valor devuelto dentro del paréntesis en la firma de la función. Este valor de error indica una **condición de fallo** en nuestra función, es decir, informa que hubo un error en la ejecución normal de la función.

```
func MyFunction() (int, error) {}
```

⚠ Recordemos que el **error** debe ser la **última variable** que retorna nuestra función.

#### ▼ Ejemplo

Tenemos nuestra función sayHi() que retorna un mensaje saludando del tipo string y un error. En nuestra función, la condición de error sucede cuando name viene vacía.

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

```
}
```

Cuando llamemos nuestra función sayHi(), debemos indicar que retorna dos valores en el orden que indica la firma de la función.

```
greeting, err := sayHi(name)
```

#### ▼ Valor Nulo

Al devolver un error de una función con varios valores de retorno, el código idiomático Go establece un **valor nulo** para cada valor "non-error". Esto quiere decir que cuando ocurre una condición de error, todas las variables **que no sean error** deben estar definidas en su valor nulo. Veamos algunos ejemplos:

Tipo	Valor nulo
int, float	0
string	""
bool	false
interface, error	nil

💡 A veces, solo nos interesa el valor de error. Podemos descartar cualquier otro valor no deseado que devuelva una función utilizando el identificador blank "\_".

```
_ , err := sayHi(name)
```

#### ▼ Validaciones

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.

```
greeting, err := sayHi(name)

if err != nil {
    fmt.Println("Error: ", err)
}
```

💡 La sentencia `if err != nil` que se muestra es la forma estándar de manejo de errores en Go, donde el error cambia el curso normal de ejecución. De esta manera, la ejecución correcta (happy path) está en el primer nivel de indentación, y toda la ejecución en condición de error, en el segundo.

## ▼ Panics

El panic es una interrupción abrupta de la ejecución de nuestro programa.

```
func panic(v interface{})
```

Ocurre en escenarios como:

- Exceder la capacidad de indexación de un array
- Invocar métodos en punteros nulos
- Intentar trabajar con canales cerrados

Pueden ser arrojados manualmente

```
panic(err)

> Iniciando...
> panic: open no-file.txt: no such file or directory
> Program exited: status 2.
```

Son útiles para fallar rápidamente en **errores** que nunca deberían durante un funcionamiento normal de un programa

#### ▼ Creando un panic

Además de las operaciones que generan un panic por defecto, también podemos generar y personalizar nuestros propios panic. Para esto, Go nos brinda la función incorporada `panic`.

```
panic("causa del panic")
```

El argumento de un panic es del tipo interface. Podemos utilizarlo para pasar la información que nos ayude a comprender el panic cuando se produzca. 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 (ya sea porque aún no sabemos cómo hacerlo, o porque no tenemos interés en hacerlo, etc.).

#### ▼ Ejemplo

Veamos un ejemplo para el cual debemos, previamente, haber definido nuestro package main e importado los packages fmt y os:

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

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

```
//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.
```

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 eso la ejecución no continuó y no se imprimió “Fin”.

## ▼ Casos de uso

#### ▼ Index out of bounds panic

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 package main e importemos el package fmt y veamos un ejemplo:

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

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.
```

#### ▼ Receptores nulos

Go brinda 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 un 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()
}

```

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.

```

#### ¿Qué podemos hacer para controlar los efectos de 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.

**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.

#### ▼ ¿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 similar a ensure o finally utilizado en otros lenguajes de programación.

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**.

#### ▼ Ejemplo

Definamos nuestro package main, importemos el package **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!")
}

```

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.

```

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**.

**⚠ 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.**

## ▼ ¿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.

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.

### ▼ Ejemplo

Definamos nuestro package `main`, importemos el package `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. En caso de ser impar, producirá un `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.

```
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")
}
```

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!")
}
```

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

```
no es un número par
;Ejecución completada!

Program exited.
```

💡 Observar cómo el valor de `panic` fue recuperado por `recover` y el programa completó su ejecución hasta el final.

## ▼ Errors vs Panics

SIMILIDADES	Diversas maneras para personalizarlos según la necesidad.	
AS	Go nos provee diferentes herramientas para manejar ambas situaciones sin que el programa deje de funcionar.	Go directamente desincentiva el uso de panics.
Pueden ocurrir en la compilación o en tiempo de ejecución.	Ocurren únicamente en tiempo de ejecución.	

DIFERENCIA	la ejecución de nuestro programa.	de ejecución.
	Un mal manejo del error puede derivar en un panic.	Pueden ser producidos por un error que no se manejó correctamente.
	Debemos prever las situaciones de error y manejarlas correctamente siempre.	En la mayoría de los casos, ocurren de forma inesperada.

## ▼ Operaciones de I/O y Archivos

El propósito de un paquete es diseñar y mantener una gran cantidad de programas, agrupando funciones relacionadas en unidades individuales para que puedan ser fáciles de mantener y comprender.

### ▼ Paquete fmt

En el lenguaje Go, el paquete fmt implementa funciones de E/S formateadas con funciones análogas a las funciones printf() y scanf() de C. La función fmt.Sprintf() formatea de acuerdo con un especificador de formato y devuelve la cadena resultante.

```
package main

import "fmt"

func main() {
    fmt.Println("Este es un texto con un salto de línea.")
}
```

### ▼ Paquete io

El paquete io proporciona interfaces básicas para primitivas de E/S. Su trabajo principal es envolver las implementaciones existentes de tales primitivas —como las del sistema operativo del paquete— en interfaces públicas compartidas que abstraen la funcionalidad.

```
r := strings.NewReader("Go es un lenguaje multipropósito.")
b, err := io.ReadAll(r)
if err != nil {
    log.Fatal(err)
}
fmt.Printf("%s", b)

Imprime: Go es un lenguaje multipropósito
```

### ▼ Paquete os

El package os proporciona una interfaz independiente de la plataforma para la funcionalidad del sistema operativo.

Veamos un ejemplo simple, abriendo un archivo y tratando de leer su contenido:

```
func main() {
    file, err := os.Open("file.go")
    if err != nil {
        log.Fatal(err)
    }
}
```

## ▼ Conurrencia y Paralelismo

La capacidad de separar la ejecución de múltiples tareas de forma individual, sin que estas se bloquen entre sí.

Para esto Go presenta las go routines que se pueden considerar como hilos ligeros de ejecución, administrados por Go en tiempo de ejecución.

### 💡 Conurrencia

Sincronización de varias cosas a la vez

### 💡 Paralelismo

Varias cosas que se ejecutan en simultáneo

El paralelismo requiere más de un procesador o núcleo, la concurrencia no.

Múltiples go routines pueden ejecutarse dentro de un mismo hilo.

Para generar una go routine agregamos la palabra **go** antes del llamado a la función.

### ✓ Ventajas

- Más baratas que los hilos.
- La cantidad puede crecer o decrecer según los requisitos del programa.

- Pueden comunicarse entre sí usando un **channel**.

### ⚠ Reglas

1. Saber cuando y cómo va a terminar esa go routine.
2. Mantener el ciclo de vida de la go routine en un único bloque.

## ▼ Canales

Portales o canales de comunicación entre diferentes go routines.

Se declaran de la forma:

```
ch := make(chan int)
```

Y se utilizan dentro de una función pasándolos como parámetros:

```
func multiplicarPorDos(num int, ch chan int) {
    res := num * 2
    ch <- res
}
```

Y esto nos permite que en vez de esperar cierta cantidad de tiempo para terminar la ejecución del programa, podemos utilizar lo que devuelven los channels:

```
func main() {
    n := 2
    ch := make(chan int)

    go multiplicarPorDos(n, ch)

    fmt.Println(<- ch)
}
```

⚠ Cabe mencionar que `<- ch` es bloqueante