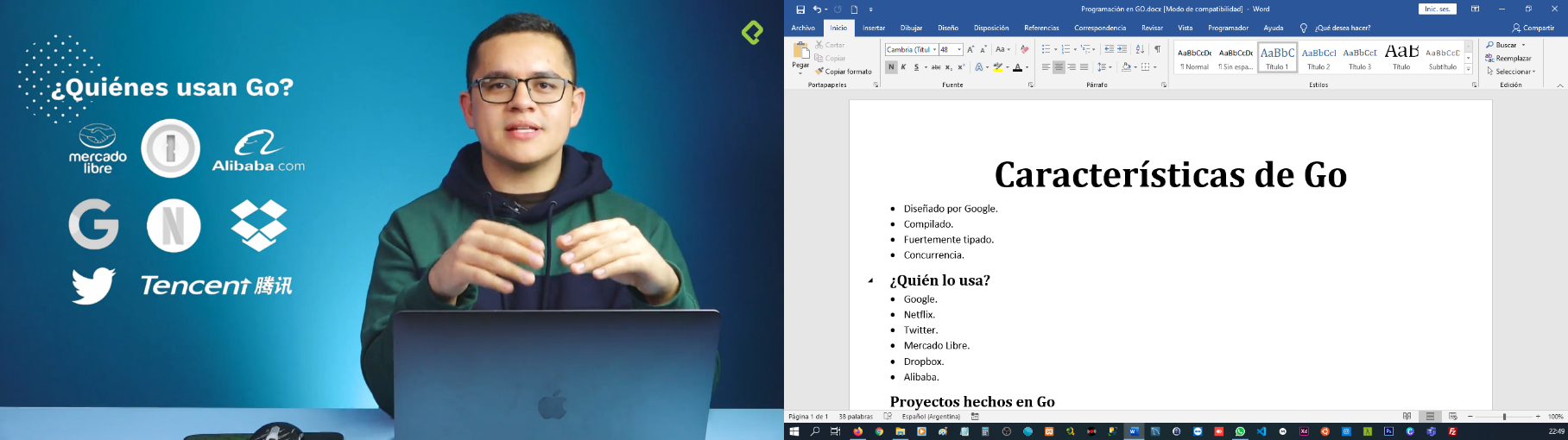
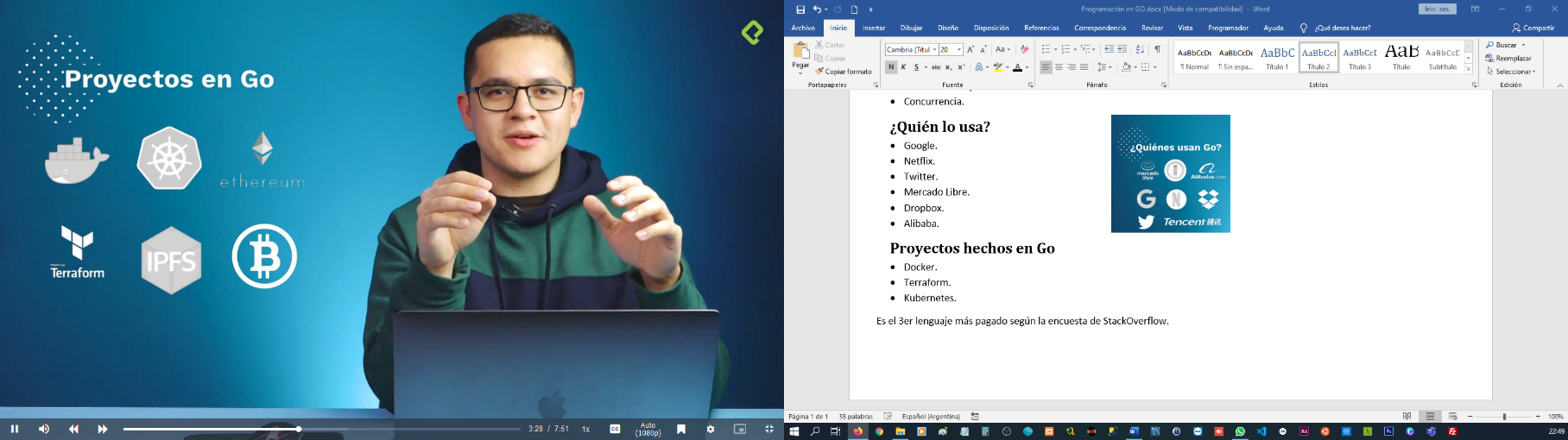
# **Características de Go**

* Diseñado por Google.
* Compilado.
* Fuertemente tipado.
* Concurrencia.

## **¿Quién lo usa?**

* Google.
* Netflix.
* Twitter.
* Mercado Libre.
* Dropbox.
* Alibaba.

## **Proyectos hechos en Go**

* Docker.
* Terraform.
* Kubernetes.

Es el 3er lenguaje más pagado según la encuesta de StackOverflow.

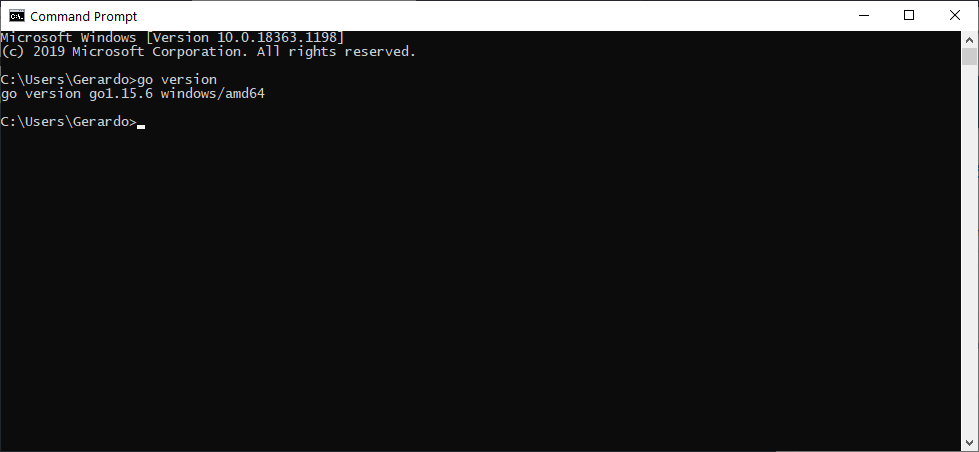
**Dato Curioso**: Uno de los creadores de GOLANG es Ken Thompson, quien también ayudó a diseñar las regExp (regex), creó el lenguaje de programación B (predecesor de C) y también aportó a la creación de UTF-8.

[Pad](https://play.golang.org/) oficial de Go.

# **Instalación en Windows**

Para poder descargar Go debemos dirigirnos al [sitio oficial](https://golang.org/dl/) y descargar la versión para nuestro SO, en este caso, windows.

1. Descargamos el archivo a nuestro equipo.
2. Ejecutamos el instalador que acabamos de descargar.
3. Abrirá una ventana, click en **Next**.
4. Aceptamos los términos.
5. **Next**.
6. Nos preguntará donde queremos hacer la instalación, lo dejamos por defecto.
7. **Install**.
8. Nos pedirá permiso de administrador, le diremos que **yes**.
9. Una vez que el asistente de instalación termine, debemos verificar si la instalación fue exitosa o no abriendo la lnea de comando y colocando **go version**.
10. Nos saldrá la siguiente información si todo ha salido ok.



# **Comandos básicos de Go**

El comando **go** nos servirá para muchísimas cosas entre ellas:

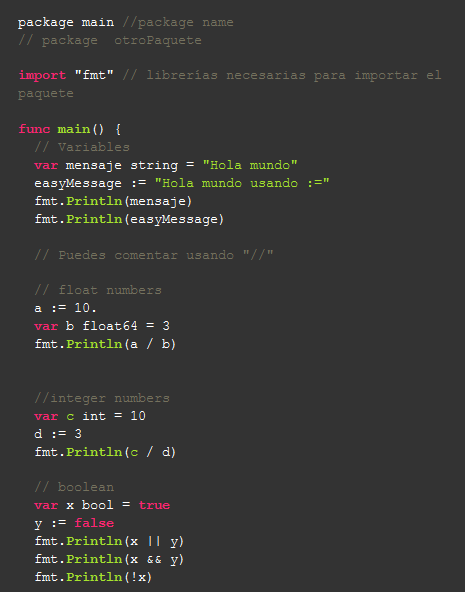
* Ejecutar un programa.
* Compilarlo.
* Instalar módulos.

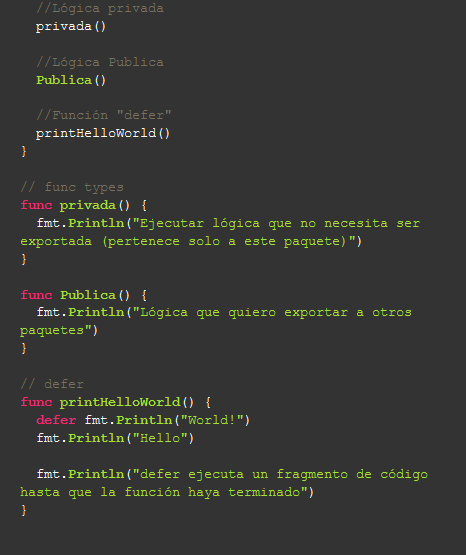
**go build**: Compila el código de tu archivo y genera un archivo binario.

**go run**: Compila tu código, ejecuta el archivo binario y cuando finaliza lo elimina.

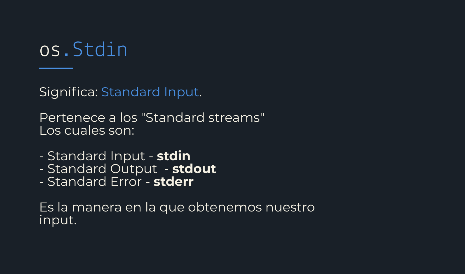
**go mod**: Este comando nos servirá para poder crear nuestros propios módulos.

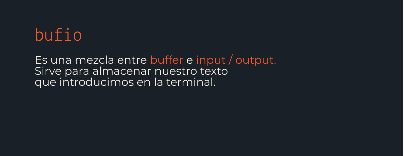
# **Analizando un programa en Go**





# **¿Qué es os.Stdin?**

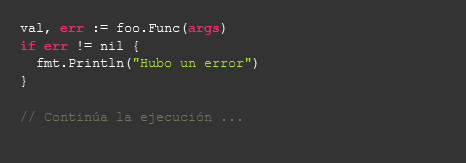




# **Manejo de errores y uso de If**

El manejo de errores dentro de **go** es un tanto más simple que en otros lenguajes de programación dado que las funciones **pueden regresar varios valores**. La norma dentro de **Go** es que las funciones regresen 2 valores usualmente:

1. **El** **valor esperado** **y un error:** Se recibe el **valor** como tal y un **error** con el valor **nil**.
2. **Si hubo un error el valor esperado**: Se recibe como **nil** y se recibe también el error con su valor correspondiente.



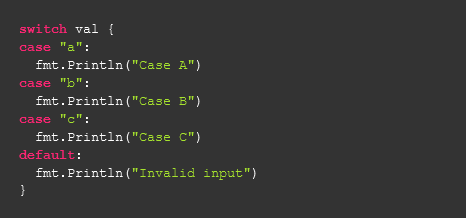
En Go, los errores inesperados también pueden manejarse con la función **panic(<message>)** de tal forma que quedaría así:



# **Switch**

El operador **switch** nos sirve para resolver o facilitar los casos que requieran múltiples comparaciones, en especial **cuando se tienen más de 3 comparaciones**. Lo anterior es debido a que, si no se podrían resolver con un **if**, **else if** y **else**.

La sintaxis en Go para el operador switch es:



Para aquellos casos en los que ninguna de las comparaciones puede manejarlo, utilizamos la palabra reservada default. Esta tiene una funcionalidad muy similar a **else**, que se ejecuta **cuando ninguna de las comparaciones se realizó**.

# **Structs y Receivers**

**Struct**: Tipo de dato que tú mismo creas, el cuál puedes agregar métodos y propiedades.

**type** nombre **struct{}**

Funcionan como formularios en papel que podría usar.

Recogen diferentes datos y los organizan con diferentes nombres de campos. Cuando inicia una variable con una nueva **struct**, es como si fotocopiase un formulario y lo dejase listo para completarse.

Las **Receiver Function** le da la propiedad al **struct** de tener el método dentro de él, es decir, cuando instanciemos a la **struct** vamos a poder llamar a los métodos dentro de la **Receiver Function**.

## **Declaración de struct**

Type noma\_struct struct{}

## **Declaración de función**

func (nombre struct) nombrefunción (variable string) int{

}

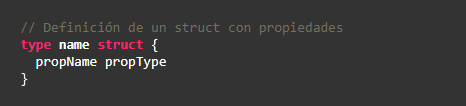
## **Instanciación de variable y llamada de función**

name\_instance := name\_struct{}

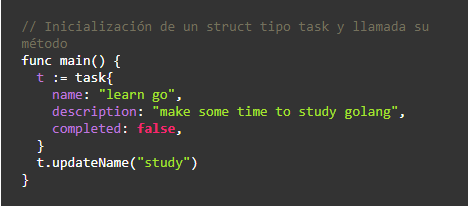
name\_instance.name\_fuction()

# **Structs para Lista de Tareas**

## **Definición de un struct con propiedades**

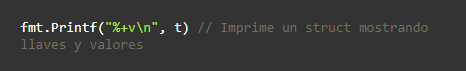


## **Inicialización de un struct tipo task y llamada su método**

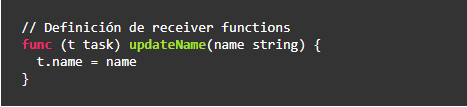


Una particularidad de Go es que, al momento de estar nombrando las propiedades de una **struct** al momento de hacer la inicialización, es que al final de cada uno lleva coma (**,**) **inclusive en la última propiedad**, esto para tener una mayor seguridad y evitar estar olvidando que la última propiedad no debe llevar como podría suceder en otros lenguajes.

En Go existe la convención de nombrar a las variables con una única letra, esto siempre y cuando el scope del código permita ubicar a la variable y su contexto fácilmente.



En Go al momento de hacer una impresión en consola podemos utilizar la función **Printf()** que recibe una plantilla de string en la que podemos pasarle diversos valores de caracteres para formatear de diversas maneras el valor que vamos a imprimir.



# **Punteros**

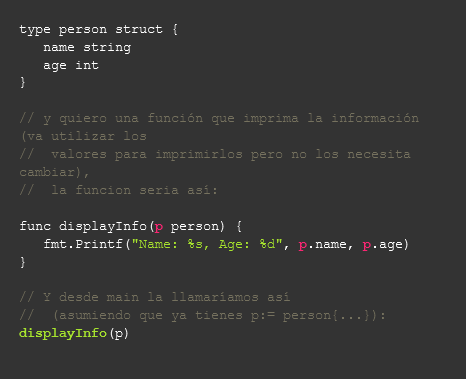
En **Go** hay dos tipos de parámetros que podemos usar en nuestras funciones o métodos:

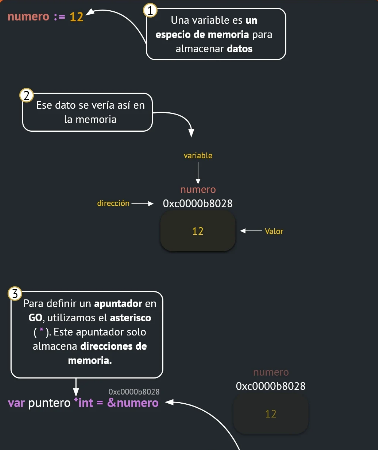
* **Pass by value** (por valor): Aquí la función ara una copia de la variable que se está pasando y modificara la copia.
* **Pass by pointer/reference** (por puntero): Aquí la función obtendrá un puntero que apunta a la misma dirección donde está el valor de la variable o struct.

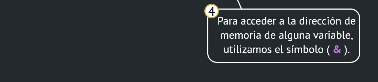
## **Usos comunes**

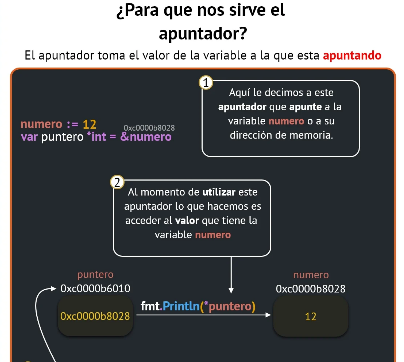
Pass by value es usado cuando necesitamos utilizar los valores de las variables o propiedades de un struct **SIN CAMBIAR EL VALOR DE ELLAS.**

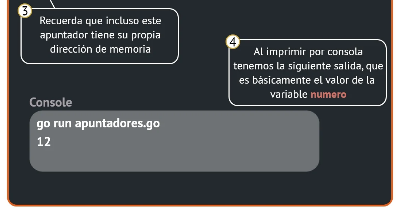
Por ejemplo, si tengo un struct person:





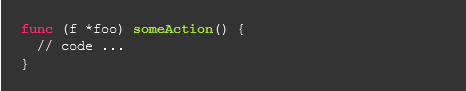






# **Punteros en Structs**

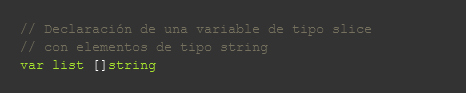
Cuando se definen **receiver functions**, es importante apuntar dichas funciones hacia punteros del **struct**. Esto se hace utilizando asterisco (indicando que es un pointer) previo al tipo del **struct**.



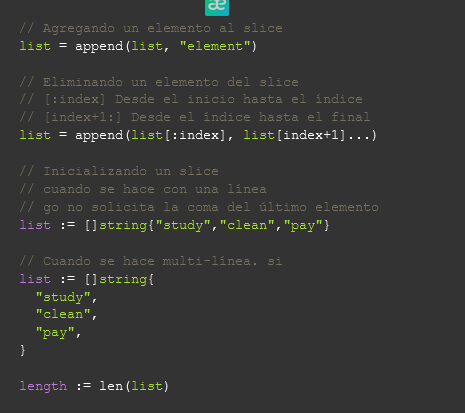
# **Uso de Slices**

Los slices son una estructura de dato que se utiliza de manera similar a los arreglos, pero con un comportamiento más similar a las listas; esto en el aspecto de que tienen una longitud variable.

Así como los arrays, los slices definen su tipo con el uso de corchetes, pero con la diferencia de que no tenemos que indicar su tamaño.

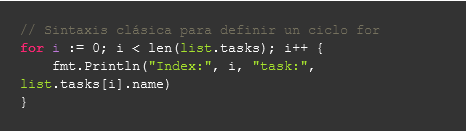


La función **append** sirve para agregar elementos a un slice. Ésta recibe como parámetros el slice y el elemento que se quiere agregar.



El método **len()** nos sirve para saber la longitud de alguna estructura de datos, string, apuntador de un array o un channel.

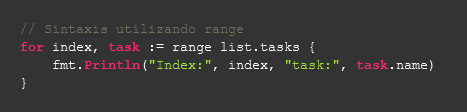
# **Ciclo For**



Este sintaxis se divide en 3 partes divididas por punto y coma:

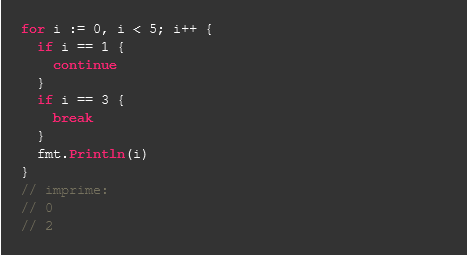
* **i := 0** es la variable que servirá como índice para el ciclo y cuál será su valor inicial.
* **i < len(list.tasks)** condición para terminar el ciclo.
* **i++** indica como crecerá el ciclo.

Se puede acceder directamente al nombre de una task, debido a que **list.task[index]** regresa directamente un **struct** tipo **task** que tiene la propiedad **name**.



Características de range:

* Es equivalente a **foreach** en otros lenguajes de programación
* **Regresa 2 valores**: El **índice** de la estructura de datos y el **valor** contenido. En caso de no usar el índice, se puede reemplazar con un guion bajo (\_).
* Éste puede iterar sobre valores de tipo string, array, slice, channel o map.



Las palabras reservadas **break** y **continue** tienen como función romper un ciclo con comportamientos diferentes.

* **break**: Sirve para romper por completo un ciclo y salga dentro de bugle antes de cumplir la condición.
* **continue**: Termina la iteración en ejecución continuando con las siguientes.

# **¿Qué es una interfaz?**

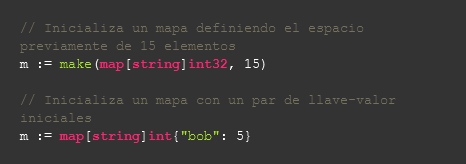
Las interfaces en Go simulan el comportamiento de conceptos como clases e interfaces de otros lenguajes, permitiendo también polimorfismo.

La condición para que un **struct** pueda considerarse de un tipo de interfaz específico se basa en su comportamiento. Esto quiere decir que a diferencia de otros lenguajes no se le definen explícitamente sus métodos, sino que es a partir de los métodos que tenga implementados, se va a inferir su tipo de interfaz.

# **Maps**

Map es una estructura llave-valor. Esto quiere decir que relaciona una llave (cuyo valor no forzoso tiene que ser **int** a diferencia de los índices de arrays ó slices) hacia un valor. La sintaxis de un map es **map[KeyType]ValueType**.

Hay 2 maneras de inicializar maps cada una con sus diferencias:



Utilizando la palabra reservada **make** lo que permite es definir e instanciar un map, pero sin valores iniciales, únicamente con su capacidad.