GO

Indice

Enlaces de interes de GO

Volver al indice ⤴️

- Web oficial
- Zona de descargas
- Documentación oficial
- Wikipedia
- Librerias estandar
- Ejemplos

Introducción

Volver al indice ⤴️

Historia de GO

La compañía **Google**, conocida por dar múltiples servicios en internet, como puede ser su famoso buscador **www.google.es**, es una empresa que maneja una gran variedad de proyectos.

Antes de que existiera Go, Google usaba para sus proyectos principalmente lenguajes de bajo nivel como podían ser C o C++. En el año 2007 tres de sus desarrolladores (**Rob Pike**, **Robert Griesemer** y **Ken Thompson**) se pusieron manos a la obra y cogieron lo mejor del lenguaje C y lo mejor del lenguaje Python para sentar las bases para un nuevo lenguaje de programación. Dos años después, **en el 2009, Google lanzo oficialmente el lenguaje de programación bautizado como Go**, también conocido como Golang o Google Go.

Go nació con el **objetivo de aprender y no cometer los mismos errores** que cometían otros lenguajes de programación mas antiguos, por ellos se cogió lo mejor de cada uno y se eliminaron problemas y limitaciones que estos presentaban.

En la actualidad Go es usado por multitud de compañías como por ejemplo Paypal, Dropbox, Netflix, Uber, Twitch, etc, y lógicamente por su creadora, Google.

Que es GO y sus características

Go es un lenguaje de programación desarrollado por Google que nació con el principal **objetivo de ser un lenguaje bastante simple, pero al mismo tiempo ser muy eficiente**. Este año 2024 cumple 15 años, lo que significa que en comparación con otros lenguajes de programación como C o Java, se trata de un lenguaje joven.

> **Características:**
>
> > **Simplicidad:** Tiene una sintaxis clara y organizada.
> >
> > **OpenSource:** Es un lenguaje de código abierto.
> >
> > **Lenguaje compilado:** El código fuente se compila generando código máquina y así ejecutar el programma.
> >
> > **Usa tipado estático:** Una vez se le asigna un tipo de dato a una variable esta ya no puede cambiar.
> >
> > **No está orientado a objetos (POO)**, pero tiene formas de emularlo
> >
> > **Es multiprocesador y concurrente:** Permite ejecutar múltiples procesos paralelamente en cada uno de los procesadores del equipo.

Instalación

Volver al indice

Archivos de instalación de GO

- Accede a **la zona de descargas** de la web oficial de GO.
- Una vez en la zona de descargas, tienes disponible la sección **"Featured downloads"** en la cual aparecen los paquetes de instalación de los sistemas operativos con sus arquitecturas más comunes.
- También está la sección **"Stable versions"** donde se encuentra un listado completo de paquetes de instalación en sus últimas versiones estables organizados por sistema operativo y por arquitecturas.

> Si en la sección "featured downloads" no aparece el paquete que coincida con tu sistema operativo y arquitectura, en la sección "Stable versions" lo encontraras.

- Por último, está la sección **"Unstable versión"** donde puedes descargar los paquetes de instalación de versiones no estables (No recomendado).

> [!NOTE]
> En el momento de hacer esta documentación (enero del 2024) la versión estable más actualizada es la 1.21.6.

- Instalación de GO en Windows -

- En Windows la forma de instalar GO es muy simple. En la zona de descargas de la página web oficial de GO debes **descargar el paquete de Windows** y ejecutarlo para iniciar la instalación.
- **El proceso de instalación se resume en pulsar "siguiente"** en las pantallas por lo que no tiene ninguna dificultad.
- Una vez finalizada la instalación puedes **confirmar que esta se ha realizado correctamente** abriendo una consola CMD y ejecutando el comando:

```
go versión
```

- Debería aparecer un **mensaje indicándote la versión instalada**, que en este caso debería ser la 1.21.6.

- Instalación de GO en Linux -

- Para instalar Go en linux abre una **consola de comandos y ejecuta los siguientes comandos:**

```
# Te sitúas en la carpeta de descargas
cd Descargas

# Descargas el paquete de instalación
wget https://go.dev/dl/go1.21.6.linux-amd64.tar.gz

# Copias el paquete en la ruta /usr/local
sudo cp go1.21.6.linux.amd64.tar.gz /usr/local

# Te sitúas en la carpeta /usr/local
cd ~
cd /usr/local

# Creas la carpeta GO y descomprimes en ella el paquete
sudo rm -rf /usr/local/go && sudo tar -C /usr/local -xzf go1.21.6.linux-
amd64.tar.gz
```

- El siguiente paso es **añadir acceso de variable global:**

```
export PATH=$PATH:/usr/local/go/bin
```

- **Para confirmar** que se ha añadido correctamente ejecutamos el siguiente comando:

```
echo $PATH

# Si aparece “/usr/local/go/bin” significa que funcionó correctamente.
```

- Finalmente, para **confirmar que GO esta instalado**, ejecuta el comando:

```
go versión
```

- Debería aparecer un **mensaje indicando la versión instalada**, que en este caso debería ser la 1.21.6.

- Instalación de GO en MacOS -

- Para instalar GO en un equipo con MacOS puedes usar el **mismo método que con el sistema operativo Windows**.
- En la zona de descargas de la página web oficial de GO descargas el **paquete de instalación de MacOS para ARM64** o **el paquete de instalación de MacOS para x86-64** y lo ejecutas para iniciar la

instalación.

- El proceso de **instalación se resume en pulsar "siguiente"** en las pantallas por lo que no tiene ninguna dificultad.
- Una vez finalizada la instalación puedes **confirmar que esta se ha realizado correctamente** abriendo una consola CMD y ejecutando el comando:

```
go versión
```

- Debería aparecer un **mensaje indicándote la versión instalada**, que en este caso debería ser la 1.21.6.

Archivos de instalación del IDE Visual Studio Code

- Para empezar a programar con GO primero debemos decidir que Entorno de desarrollo (IDE) vamos a usar.
- Existe un IDE llamado **GoLand** el cual fue creado en exclusiva para ser usado junto con Go, pero por desgracia se trata de un IDE de pago.
- Existen muchas opciones, pero el **IDE más usado en la actualidad y que además es compatible con GO es Visual Studio Code**, por lo que es el que recomiendo.

- Instalación de VSC en Windows -

- Para instalar Visual Studio Code debes acceder a la **zona de descargas** de su página web oficial, pulsar en el botón donde pone "Windows" y **descargar el paquete de instalación.**

> [!NOTE]
> En el momento de realizar esta documentación la versión más actualizada es la 1.85.

- Una vez descargado debes ejecutar el paquete de instalación. El **proceso de instalación se resume en pulsar "siguiente"** en las pantallas por lo que no tiene ninguna dificultad.

- Instalación de VSC en Linux -

- Para realizar la instalación de Visual Studio Code en linux **abre una consola y ejecuta los siguientes comandos:**

```
sudo apt update

sudo apt install software-properties-common apt-transport-https wget

wget -q https://packages.microsoft.com/keys/microsoft.asc -O- | sudo apt-key add -
sudo add-apt-repository "deb [arch=amd64]
https://packages.microsoft.com/repos/vscode stable main"

sudo apt install code
```

- Instalación de VSC en MacOS -

- Para instalar Visual Studio Code debes acceder a la **zona de descargas** de su página web oficial, pulsar en el botón donde pone “Mac” y **descargar el paquete de instalación.**

> [!NOTE]
> En el momento de realizar esta documentación la versión más actualizada es la 1.85.

- Una vez descargado debes ejecutar el paquete de instalación. El **proceso de instalación se resume en pulsar “siguiente”** en las pantallas por lo que no tiene ninguna dificultad.

- Extensiones recomendadas -

- Una vez tengas instalado el IDE Visual Studio Code, lo primero que harás es ir a la **sección de extensiones** (Control + Mayus + X), escribir **“go”** e instalar la primera extensión que aparece.
- La extensión “GO” añadirá funciones extras a Visual Studio Code cuando estés programando con el lenguaje GO, como por ejemplo **IntelliSense.**
- Si tras la instalación de la extensión "go” o cuando empieces a generar tus primeros códigos de GO aparece en la esquina de abajo a la derecha algún **mensaje indicándote que hace falta instalar alguna extensión extra, no lo dudes y pulsa el botón instalar** ya que te hará falta.
- También sería recomendable que instalaras la extensión **“Spanish Languaje Pack for Visual Studio Code”** para que tengas el IDE en español.

Hola Mundo

Volver al indice ⤴️

Como suele ser común con todos los lenguajes de programación, el primer código que probaras será el famoso **“Hola Mundo”.**

- Para ello lo primero será **crear una carpeta** en el directorio que tú quieras de tu ordenador, teniendo en cuenta que esa carpeta será donde vas a guardar tus proyectos en GO.
- Una vez creada la carpeta, en Visual Studio Code debes ir a **“Archivo > Agregar carpeta al área de trabajo”** y seleccionar la carpeta que acabas de crear.
- Lo siguiente será crear un archivo dentro de la carpeta, al que llamaremos **“HolaMundo.go”.**
- Para generar el primer Hola Mundo debes escribir el siguiente código:

```
package main

import "fmt"

func main() {
    fmt.Println("Hola mundo")
}
```

- Para ejecutarlo debes pulsar el botón **“Run code”** que se encuentra en la parte superior derecha de Visual Studio Code, o usar el atajo de teclado **Control + Alt + N.**
- El resultado será que se muestre por consola el mensaje “Hola mundo”.

Conceptos basicos

Volver al indice ⤴️

Ya sabes que es GO y tienes listo el entorno para empezar a usarlo, pero antes **es importante entender unos conceptos básicos.**

Concurrencia

La concurrencia es un concepto importante a tener en cuenta ya que **es uno de los aspectos fundamentales de GO**, el cual permite la creación de programas concurrentes de manera fácil y eficiente.

Lo más común es que un programa siga un camino a la hora de ejecutarse y hasta que no finaliza una tarea no continúa con la siguiente. La concurrencia **permite que varias tareas se ejecuten simultáneamente.**

Go introduce el concepto de **goroutines**, que son **unidades de concurrencia** que permiten que las funciones se ejecuten de manera concurrente. A la hora de gestionar, por ejemplo, un servidor, la concurrencia es algo clave para mejorar la eficiencia.

Paquetes e importaciones

Los programas escritos en lenguaje GO **se organizan en paquetes** que ayudan a gestionar y modular el código. Siempre que empieces un proyecto en GO el **primer archivo .go que crees será el paquete principal main.** Cada archivo .go que crees **debe tener en su primera línea de código el nombre del paquete al que pertenece.**

```
//archivo main.go
package main

func main() {
    //Cuerpo del codigo
}
```

Uno de los puntos interesantes de la organización por paquetes es que **puedes importar paquetes te has creado a otros archivos GO y usarlos.**

```
// Archivo geometry.go
package geometry

import "math"

func AreaOfCircle(radius float64) float64 {
   return math.Pi * radius * radius
}
```

```
// En el archivo main.go importas el paquete geometry
package main

import (
    "fmt"
    "Paquete/geometry"
)

func main() {
    radius := 5.0
    area := geometry.AreaOfCircle(radius)
    fmt.Printf("El área del círculo es: %.2f\n", area)
}
```

GO cuenta con bastantes **paquetes estándar que ya vienen incluidos en el propio instalador** que también puedes ser importados. Por ejemplo, podemos importar el paquete fmt que nos permitiría escribir textos en consola y el paquete math que sirve para realizar calculos matematicos.

```
//archivo main.go

package main

import (
    "fmt"
    "math"
)

func main() {
    //Cuerpo del codigo
}
```

Convención de nomenclaturas

Cuando hablamos de convención de nomenclaturas nos estamos refiriendo a que **los programadores siguen unas reglas de escritura de código** que ayuda a que, por ejemplo, en caso de que otra persona tenga que utilizar tu código, le sea más fácil de entender.

- Los **nombres de los paquetes deben ser cortos**, no usar guiones y evitar mezclar mayúsculas y minúsculas.
- Los **nombres de las variables y funciones deben ser camelCase** (palabras juntas, sin espacios ni guiones, la primera letra de cada palabra en mayúscula excepto la primera. Ejemplo: miNuevaVariable).

```
var miNuevaVariable int
func calcularSuma(a, b int) int {
    return a + b
}
```

- El **nombre de las Interfaces debe comenzar por la letra "I" seguida de camelCase**

```
type IConectar interface {
    Conectar() error
}
```

- El **nombre de las constantes se escriben en mayúsculas y en caso de ser varias palabras, separadas por guiones.**

```
const MAX_INT = 100
```

Comentarios

Los comentarios son líneas de código que los programadores que revisen el código podrán ver, pero que una vez el programa se compile y ejecute, serán ignoradas.

El objetivo de escribir comentarios es **dejar explicado el funcionamiento del código.**

> [!TIP] Usar comentarios está considerado como una muy buena práctica.

No solo es útil por si en un futuro otro programador tiene que revisar tu código, si no que si pasado un tiempo tienes que revisar tu propio código te ayudara a recordar de una manera rápida y simple que hacía cada cosa.

```
//Esto es un comentario de línea
```

```
/*Esto es un comentario
en bloque de varias lineas */
```

Generación de documentación con Godoc

En GO existe una herramienta llamada **“godoc”** que sirve para generar **documentación automática** sobre el proyecto a partir de los comentarios que se van escribiendo.

Al ejecutar la herramienta **se extrae automáticamente todos los comentarios y genera una documentación** con todos ellos.

Para ejecutar “godoc” debes situarte en el directorio raíz de tu proyecto GO y escribir por consola este comando:

```
godoc -http :PUERTO
```

> En “PUERTO” debes poner un puerto que no esté siendo utilizado para otros servicios de tu ordenador.

Una vez ejecutado el comando podrás acceder a la documentación desde tu navegador ingresando **http://localhost:PUERTO/pkg/PAQUETE**

> En “PAQUETE” debes escribir de que archivo .go (que debe coincidir con el nombre del paquete) quieres revisar la documentación de los comentarios.

Supongamos que uso el puerto 8080 y quiero revisar la documentación generada en mi archivo principal main.go

```
http://localhost:8080/pkg/main
```

Tipos de datos

Durante la creación del código estarás obligado a usar distintos tipos de datos. Es importante entender todos los tipos de datos que existen y siempre que sea posible usar el tipo que mejor se adapte a la situación

- Enteros -

El tipo de dato entero sería lo que comúnmente conocemos como **un número normal. (1, 2 3, etc).**

> Los tipos de datos enteros que existen son: **int8, int16, int32, int64, uint8, uint16, uint32 y uint64.**

Cada uno de ellos ocupa una cantidad de espacio en memoria y cuanto más ocupe, más alto será el rango de valores que permite manejar.

Tipo	Tamaño	Rango
int8	1 byte	-128 to 127
int16	2 bytes	-32,768 to 32,767
int32	4 bytes	-2,147,483,648 to 2,147,483,647
int64	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
uint8	1 byte	0 to 255
uint16	2 bytes	0 to 65,535
uint32	4 bytes	0 to 4,294,967,295
uint64	8 bytes	0 to 18,446,744,073,709,551,615

> [!NOTE]
> Por convenio se usa siempre (salvo casos muy puntuales) el tipo int.

```
var x int = 10
```

Tipo	Tamaño	Rango
int	Variable	Depende de la arquitectura del sistema (32 o 64 bits)

- Flotantes -

El tipo de dato flotante sería lo que comúnmente conocemos como **un número decimal. (1.5, 2.8, etc).**

> Los tipos de datos flotantes que existen son: float32 y float64.

Cada uno de ellos ocupa una cierta capacidad de espacio en memoria y cuanto más ocupe, más alto será el rango de valores que permite manejar.

Tipo	Tamaño	Rango
float32	4 bytes	-3.4028234663852886e+38 to 3.4028234663852886e+38
float64	8 bytes	-1.7976931348623157e+308 to 1.7976931348623157e+308

> [!NOTE]
> El más usado por regla general es el tipo float64.

- Booleanos -

En programación, los booleanos son un tipo de dato fundamental utilizado para almacenar **valores lógicos.** El tipo de dato booleano puede tomar uno de dos valores: **true (verdadero) o false (falso).** Estos valores son esenciales para controlar la lógica en los programas y para tomar decisiones basadas en condiciones.

```
var esVerdadero bool = true
var esFalso bool = false
```

- Strings -

En el mundo de la programación, un "string" no es más que **una secuencia de caracteres.**

> Para entenderlo mejor, imagina que una cadena de texto es como una línea de letras, números y símbolos.

Puedes crear una cadena de texto usando la **palabra clave var y luego asignarle tu mensaje entre comillas.**

```
var miCadena string = "Hola, mundo!"
```

> En este código, estamos diciendo: "Vamos a tener una cadena de texto llamada miCadena, y le daremos el valor de 'Hola, mundo!'.

El tipo de dato string es uno de los más importantes, además, **permite ser manipulado de múltiples formas:**

- **Concatenación de Cadenas:** Imagina que quieres unir dos mensajes. Puedes hacerlo usando el símbolo +:

```
var saludo = "Hola, "
var nombre = "usuario"
var mensaje = saludo + nombre
```

- **Longitud de una Cadena:** Un string es al fin y al cabo un conjunto de caracteres por lo que podemos saber su longitud contando, para ello se usa la función len():

```
// En este caso, longitud sería 8.
var palabra = "elefante"
var longitud = len(palabra)
```

- **Obtener un Carácter de una Cadena:** De igual manera que puedes contar la longitud de caracteres de un string, puedes hacer referencia al carácter que quieras de la cadena utilizando corchetes y el número que represente su posición:

```
// En este caso, primerCaracter sería "H", ya que en la posición 0 de la cadena está la letra "H".
var miCadena = "Hola, mundo!"
var primerCaracter = miCadena[0]
```

- **Conversión a Mayúsculas o Minúsculas:** Es posible manipular el string y convertir todos los caracteres a mayúsculas o a minúsculas

```
// En este caso, mensajeMayusculas sería "HOLA, MUNDO!" y mensajeMinusculas sería "hola, mundo!"
var mensaje = "Hola, mundo!"
var mensajeMayusculas = strings.ToUpper(mensaje)
var mensajeMinusculas = strings.ToLower(mensaje))
```

- **Buscar una cadena de caracteres dentro de un string:** Se puede dar el caso de que el string sea una frase y quieras saber si la frase contiene alguna palabra o letra en particular, pues tambien es posible conseguirlo.

```
// En este caso, contieneHola sería true, ya que la cadena contiene la palabra
"Hola".
var mensaje = "Hola, mundo!"
var contieneHola = strings.Contains(mensaje, "Hola")
```

- Conversión entre tipos de datos -

En la mayoría de lenguajes de programación existe la conversación de tipo de datos, es decir, **poder cambiar el tipo de dato a una variable**. Por ejemplo, convertir una variable de tipo entero en tipo flotante.

En GO es posible realizar la conversión de tipos, pero **no se realiza de la misma manera que en otros lenguajes de programación**. Lo común es cambiar el tipo de dato a la variable, pero en Go una vez se le asigna a una variable su tipo de dato no es posible modificarlo.

Para poder realizar la conversión de todas en GO lo que se hace es **guardar el valor de la variable a la que queremos cambiarle el tipo, en una nueva variable y a esa nueva variable asignarle el tipo que queramos.**

```
//Tienes una variable entera de valor 10
var x int = 10
// float64(x) es la expresión de conversión de tipo que toma el valor de x y lo
convierte a float64, creando así una nueva variable “y” de tipo float64.
var y float64 = float64(x)
```

Operadores

- Operadores aritméticos -

Los operadores aritméticos son aquellos que **permiten realizar operaciones matemáticas entre números.**

- **Suma (+):** Permite sumar dos valores

```
resultado = 5 + 3 // resultado será 8
otroResultado = resultado + 10  // también podemos sumar otro valor
```

- **Resta (-):** Permite restar el segundo valor al primero

```
resultado = 5 - 3 // resultado será 2
otroResultado = resultado - 8   // también podemos restar otro valor
```

- **Multiplicación (*):** Permite multiplicar dos valores

```
resultado = 5 * 3  // resultado será 15
otroResultado = resultado * 2   // también podemos multiplicar por otro valor
```

- **Division (/):** Permite dividir el primer valor por el segundo

```
resultado = 10 / 2 // resultado será 5
otroResultado = resultado / 3   // también podemos dividir por otro valor
```

- **Modulo (%):** El resultado es el resto de la division del primer valor por el segundo

```
resultado = 10 % 3 // resultado será 1
otroResultado = resultado % 2   // también podemos calcular el módulo respecto a
otro valor
```

- **Incremento (+=) y decremento (-=):** Realiza el incremento o decremento de un valor en la cantidad que le indiquemos.

```
contador = 5
contador += 1   // incrementa el contador en 1
contador -= 2   // decrementa el contador en 2
```

- **Operaciones combinadas:** Puedes combinar varios operadores aritméticos en una expresión para realizar cálculos más complejos.

```
resultado = (5 + 3) * 2 / (4 - 1)   // resultado será 6
```

A la hora de realizar operaciones combinadas hay que tener en cuenta el **orden de prioridad** a la hora de hacer los calculos:

1.) Paréntesis (): Las operaciones dentro de paréntesis tienen la mayor prioridad.

2.) Multiplicación *, *División /, Módulo %:* Estos operadores tienen la misma prioridad y se evalúan de izquierda a derecha.

3.) Suma +, Resta -: Estos operadores también tienen la misma prioridad y se evalúan de izquierda a derecha.

- Operadores lógicos -

Los operadores lógicos sirven para realizar **operaciones de lógica booleana.** Se debe coloca el operador entre dos expresiones o condiciones y se evalúa dependiendo del operador.

- **AND (&&):** Evalúa que ambas expresiones son verdaderas, en caso de serlo devuelve verdadero y en caso contrario falso. Traducido al lenguaje humano significa "y".

```
resultado = (5 > 3) && (4 < 7)  // Se cumplen ambas expresiones por lo que el resultado será verdadero
resultado2 = (2 == 2) && (10 <= 15)  // Verdadero, ambas expresiones son verdaderas
```

- **OR (| |):** Evalúa que alguna de las expresiones sean verdaderas, en caso de serlo devuelve verdadero y en caso contrario falso. Traducido al lenguaje humano significaría "o".

```
resultado = (5 > 3) || (4 > 7)  // Se cumple al menos una expresión por lo que el resultado será verdadero
resultado2 = (1 == 0) || (3 != 3)  // Falso, ya que ambas expresiones son falsas
```

- **NOT (!):** Niega el valor real de la expresión, por lo que si él fuera una expresión verdadera devolvería falso y si fuera una expresión falsa devolvería verdadero.

```
resultado = !(5 > 3)  // La expresión sería verdadera pero al negarla el resultado será falso
resultado2 = !(10 == 10)  // Falso, ya que la expresión original es verdadera
```

- **Combinación de operadores lógicos:** Es posible combinar varios operadores lógicos para construir expresiones más complejas.

```
resultado = (3 < 5) && ((10 == 10) || (8 > 12))  // Verdadero, se cumplen todas las condiciones
```

A la hora de realizar operaciones lógicas hay que tener en cuenta el **orden de prioridad** de los operadores:

1.)NOT !

2.)OR ||

3.)AND &&

- Operadores de comparación -

Los operadores de comparación sirven para **comparar dos valores y devolver un resultado booleano dependiendo del operador.**

- **Igual a (==):** Compara dos valores y si son iguales devuelve verdadero

```
resultado = (5 == 5)  // Ambos valores son iguales por lo que el resultado será verdadero
```

- **Distinto a (!=):** Compara dos valores y si son distintos el resultado será verdadero

```
resultado = (5 != 3)  // Los valores son distintos por lo que el resultado será verdadero
```

- **Mayor que (>):** Compara ambos valores y si el primero es mayor que el resultado será verdadero, de lo contrario sera falso

```
resultado = (5 > 3)  // El primer valor es mayor que el segundo, por lo que el resultado será verdadero
```

- **Menor que (<):** Compara ambos valores y si el primero es menor que el segundo el resultado será verdadero, de lo contrario será falso.

```
resultado = (5 < 3)  // El primer valor es mayor que el segundo por lo que el resultado será falso
```

- **Mayor o igual que (>=):** Compara ambos valores y en caso de que el primero sea mayor o igual que el segundo el resultado será verdadero, de lo contrario será falso

```
resultado = (5 >= 5)  // El primer valor es mayor o igual que el segundo valor por lo que el resultado será verdadero
```

- **Menor o igual que (<=):** Compara ambos valores y en caso de que el primero sea menor o igual que el segundo el resultado será verdadero, de lo contrario será falso

```
resultado = (5 <= 3)  // El primer valor es mayor que el segundo valor por lo que el resultado será falso
```

Variables

Volver al indice ⤴️

Es muy importante tener claro el concepto de variable, de lo contrario el resto de conceptos más avanzados serán imposibles de manejar porque prácticamente todos ellos hacen uso de estas variables.

La definición de variable por convenio sería: **Un espacio en memoria donde se almacenan datos.**

Para que puedas entenderlo debes pensar que tu ordenador tiene una cantidad limitada de memoria RAM. Cuando creas una variable en tu código, **lo que estás haciendo es coger un trozo de ese espacio y almacenar en él un valor, el cual luego podrás usar a lo largo de tu programa.** Eso significa que el espacio para crear variables es limitado, esto era un problema hace varias décadas cuando los equipos informáticos tenían cantidades de RAM muy bajas, pero ya hoy en día no suele serlo. Eso no significa que se deba abusar y generar código inútil, todo lo contrario, **hoy en día se prima que el código sea lo más simple posible en busca de que un programa sea optimo en lo que respecta a gasto de recursos.**

Declaración y reasignación

En GO tienes varias **formas de declarar una variable:**

- Se pueden crear variables **sin ningún valor establecido**, es decir, vacías.

```
// La variable x será int y no tendrá ningún valor asignado
var x int
```

- A continuación **le asignamos un valor** a la variable x.

> [!IMPORTANT]
> Como anteriormente se le asigno el tipo int su valor debe ser numérico, de lo contrario dará error.

```
// Asignamos el valor 10 a la variable x que hasta el momento estaba vacía
x = 10
```

- Si en el anterior ejemplo declaramos una variable vacía y luego le asignamos un valor, en este ejemplo realizamos ambas acciones juntas con una **acción en una sola línea**

```
// Declaración de una variable especificando que su contenido sea 20 y su tipo
int:
var z int = 20
```

- Tienes la opción de **declarar una variable sin especificar su tipo de dato** y que sea GO quien automáticamente le asigna el tipo.

> [!IMPORTANT]
> Debes darle un valor a la variable y en base a ese valor GO le asignará el tipo que le corresponda.

```
// Declaración de una variable con valor 20 sin especificar su tipo
var z = 20
```

- Un concepto importante que no se mencionó antes es que **las variables, como su nombre indica, pueden variar su valor**, es decir, puedes declarar una variable con el valor 10 y más adelante cambiarle el valor por 20.

> [!IMPORTANT]
> Se puede cambiar el valor, pero hay que respetar el tipo de dato.

```
// Declaras una variable vacía de tipo int
var y int

// Se le asigna el valor 5 a la variable
y = 5

// Se le reasigna un nuevo valor sustituyendo el antiguo
y = 10
```

```
// Declaras una variable asignando un valor de 3, por lo que GO le asigna el tipo
int
var m = 3

// Como GO le asigno el tipo int, no hay problema con cambiar su valor por otro
int
m = 6
```

```
// Declaras una variable asignando un valor 10, por lo que GO le asigna el tipo
int
var h = 10

// Aunque no esté indicado, GO le asignó el tipo int a la variable, por lo que si
se intenta asignarle un valor distinto a su tipo, esto provocará un error
h = “hola”
error!!!
```

Variables según su ambito

Cuando se habla del ámbito de una variable lo que quiere decir es si **esa variable es accesible en todo el código o solo en una parte específica.**

- Variables locales -

Un concepto que aún no se ha explicado son las funciones. Se explicarán con detalle más adelante, pero para que lo entiendas, son trozos de código que pueden ser o no invocados y cumplen una función en particular.

Una variable local es una variable que ha sido **declarada dentro de una función y solo existe dentro de ella.** Eso quiere decir que una variable que se encuentra dentro de una función no afecta al resto del código que se encuentre fuera de ella, sino solo a lo que ocurra dentro de esa función.

- En este ejemplo, la variable local sería z que es la que se encuentra dentro de la función. Si al finalizar el código quisieras saber el valor de z lo te encontrarás es que la variable no está declarada.

```
var g = 10

func main() {
   var z = 5
}
```

- Variables globales -

Una variable global es una variable que ha sido **declarada en el cuerpo principal del programa**, es decir, no se encuentra dentro de ninguna función. Este tipo de variables pueden ser accedidas desde cualquier parte del programa.

- En este ejemplo, la variable "g" sería una variable global.

```
var g = 10

func main() {
   var z = 5
}
```

> [!WARNING]
> El uso de variables globales se considera una mala práctica de programación.

Constantes

Volver al indice ⤴️

Si en el apartado anterior se dijo que una de las características de las variables es que pueden variar su valor establecido, ahora hablaremos de las constantes, que como su nombre indica, **su valor una vez establecido es constante y no puede cambiar.**

La principal duda que uno puede hacerse es, ¿por qué usar una constante y no una variable que permite más maniobrabilidad?

La realidad es que las constantes **se suelen usar solo en casos muy específicos en los cuales sabemos que un valor no cambiara nunca**, como por ejemplo: el número PI que sabemos que su valor siempre será 3.14 o los días de la semana.

```
const PI = 3.14159
```

Arrays

Volver al indice ⤴️

Los arrays (o también llamados arreglos) son **estructuras de datos donde se almacenan valores de un mismo tipo.**

Puedes imaginar un array como una caja donde puedes ir metiendo valores de manera ordenada y que cuando necesites alguno de esos valores puedes acceder a él sabiendo el puesto que ocupa dentro de la caja.

- Es esencial entender que todos los **valores dentro de un array deben ser del mismo tipo**. Esto significa que si creas un array de enteros, todos los elementos deben ser enteros.

```
// Creas un array de enteros (int) con la capacidad de 3 valores
var arr [3]int
```

- Un punto importante es que a la hora de guardar y acceder al contenido de **un array no se empieza por el nuevo 1, sino por el 0**. Eso quiere decir que si creas un array con capacidad de almacenar 3 valores, a la hora de guardar y acceder a esos valores no accederás a los huecos 1, 2 y 3, sino a los huecos 0, 1 y 2.

```
// Creas un array de 5 huecos
var arr [5]int

// Guardas en el primer hueco un valor
arr[0] = 5
```

- También tienes la posibilidad de **crear un array y añadirle contenido en la misma línea** de código

```
var arr = [3]int{5, 10, 15}
```

- Así como un array es un contenedor de variables, existe la posibilidad de que un array contenga otro array. A esta estructura se le llama **array multidimensional o matriz**. No hay límite en la cantidad de arrays que puedes anidar, pero ten en cuenta que la complejidad para manejarlos aumenta considerablemente.

```
// Ejemplo: Crear una matriz bidimensional
var matriz = [2][3]int{{1, 2, 3}, {4, 5, 6}}
```

- **Longitud del Array:** Es importante mencionar que un array tiene una longitud fija que se establece en el momento de su creación. La longitud no puede modificarse una vez que se ha definido.

```
// Ejemplo: Crear un array de longitud 4
var arr [4]int
```

- **Slices como Alternativa Dinámica a los Arrays:** Los slices en Go son una estructura de datos más versátil y dinámica en comparación con los arrays. A diferencia de los arrays, los slices no tienen una longitud fija al ser creados, lo que los hace más flexibles para gestionar conjuntos de datos de tamaño variable.
- **Creación de Slices:** Puedes crear un slice utilizando la función make o directamente con la sintaxis de corchetes sin especificar una longitud.

```
// Crear un slice que contenera int, de longitud 3, utilizando la función make
slice := make([]int, 3)

// Crear un slice que contendra los strings "a", "b" y "c", sin especificar
longitud (longitud dinámica)
otroSlice := []string{"a", "b", "c"}
```

- **Agregar y Eliminar Elementos:** A diferencia de los arrays, los slices permiten agregar y eliminar elementos de manera dinámica. Puedes utilizar las funciones append y el slicing para lograr esto.

```
// Agregar un elemento a un slice
slice = append(slice, 4)

// Eliminar un elemento de un slice
indiceAEliminar := 1
slice = append(slice[:indiceAEliminar])
```

Estructuras de control

Volver al indice ⤴️

Un concepto muy básico de la programación es que por defecto es lineal, es decir, **los códigos se van ejecutando en orden desde la línea 1 hasta el final.**

Las estructuras de control son instrucciones que provocan que **el código no siga ese camino lineal** y dependiendo de ciertas situaciones pueda desde saltarse un trozo de código hasta saltar a la otra punta del código para ejecutar una función específica.

Condicionales

Los condicionales son estructuras de control que **piden que algo se cumpla para ejecutarse.**

- If, else y else if -

- La primera estructura de control condicional es **if**, la cual es una de las que más usadas, sobre todo cuando se está comenzando a programar.
- Al usarlo lo que estás haciendo es evaluar una **condición, que en el caso de ser verdadera ejecuta un código** y en caso de ser falsa lo ignora.

```
// En este código se evalúa la condición de que x sea mayor que 10. Al ser
verdadera se ejecuta lo que está dentro del if.
var x int = 11
if x > 10 {
  x = 100
}
```

- La segunda estructura de control condicional es **else**, el cual complementa al if haciendo que **si no se cumple uno se cumpla el otro.** Si la condición del if se cumple, se ejecuta el código que está dentro de este, pero si la condición resulta ser falsa, lo que se ejecuta es el código del else.

```
// En este código se evalúa la condición y es falsa, por lo que se ignora el
código del if y se ejecuta el del else.
var x int = 11
if x > 15 {
    x = 100
} else {
    x = 200
}
```

- La tercera estructura de control condicional es el **else if**, el cual sería como **si añadieras más if a la estructura de control**. Primero se evaluaría el if, en caso de ser falso, se evaluaría el else if y si también es falso, finalmente se llegaría al else. Puedes poner tantos else if como quieras.

```
// En este código se evalúa la condición del if y es falsa, luego se evalúa el
else if, que al ser verdadera se ejecuta su trozo de código, ignorando el resto.
var x int = 10
if x > 10 {
    x= 100
} else if x == 10 {
    x = 150
} else {
    x = 200
}
```

> [!WARNING]
> Es muy común, sobre todo al comenzar a programar, abusar del uso de if llegando al punto de hasta crear **if anidados** (if unos dentro de otros). **Esto último se considera una mala práctica de programación.**

- Switch -

La estructura de control condicional switch permite coger una variable y **hacer una comparación con diferentes condiciones.**

- Inicialmente habrá una variable que tendrá un valor y a continuación se darán varios casos en los cuales se busca que alguno de ellos coincida con el valor de la variable que se dio al inicio. **Si alguno de los casos coincide con la variable, el código de ese caso es el que se ejecutara.** Si ocurriera que ningún caso se cumple, el código que se ejecutaría sería el **default**, que funcionaria como si de un else se tratara.

```
// Tenemos una variable llamada x con el valor “juan”
var x string = "juan"
    switch x {
      // El primer caso no coincide con el valor de x, por lo que se ignora
      case "alberto": xxx
    // El segundo caso coincide con el valor de x, por lo que se ejecuta el código
yyy
      case "juan": yyy
      // El default es ignorado porque ya se ha cumplido uno de los posibles casos
      default: zzz
    }
```

Bucles

Los bucles son estructuras de control que provocan que **un mismo fragmento de código se repita de manera indefinida** mientras se cumplan un cierto requisito.

- For -

Si antes se mencionó que la estructura de control if era una de las más usadas, sobre todo al comenzar a programar, el for se usa igual o incluso más cuando el nivel ya empieza a subir.

La estructura de control for tiene múltiples funciones, pero para resumirlo se podría decir que **mientras la condición del for se cumpla, el código que está dentro de este se repetirá de manera indefinida.**

> [!CAUTION] A la hora de manejar bucles debes tener mucho cuidado con las condiciones ya que si una condición se cumple siempre daría lugar a un **bucle infinito**, lo que provocaría que el programa fallara. Siempre hay que asegurarse de dejar una forma de que la condición deje de cumplirse y finalice el bucle.

- La **estructura estandar del for** es la siguiente:

```
for [inicialización]; [condición]; [modificación] { // código a ejecutar}
```

- **Inicialización:** Se ejecuta solo en la primera iteración del for.
- **Condición:** En cada iteración del bucle se evalúa esta condición, que en caso de ser verdadera, se ejecutaría el código del bucle.
- **Modificación:** Al finalizar la iteración se ejecutaría la modificación que esté indicada.

> La inicialización, condición y modificación no son obligatorias, lo que permite generar infinitas posibilidades al usar un bucle for.

```
// En este ejemplo, se inicializa el for generando una variable a la que se le da el valor de 0.

// Se evalúa si la variable es menos de 10, al ser verdadero se ejecutaría el código dentro del for.

// Al final de cada iteración se suma 1 a la variable

// Tras 10 iteraciones la variable será 10, no se cumplirá la condición y el for finalizará

for i := 0; i < 10; i++ {
xxx
}
```

- While y do while -

Los bucles **while y do-while** son 2 tipos de bucles que generalmente **existen en la mayoría de lenguajes de programación, pero GO no es uno de ellos**. Aunque no existan oficialmente en GO, sí que **es posible emularlos** haciendo un uso específico del bucle for.

Lo primero sería entender que son los bucles while y do-while. Estos bucles tienen un funcionamiento simple de entender: **mientras se cumpla la condición, el código se seguirá repitiendo hasta que deje de cumplirse.**

La principal diferencia entre estos 2 bucles es que el bucle **while solo se ejecuta si se cumple la condición**, mientras que el bucle **do-while siempre se ejecuta mínimo una vez** y tras eso se comporta como si un while normal se tratase.

- La forma de **emular un bucle while con un for** sería la siguiente:

```
// Se inicializa la variable con valor 0
// Si se cumple la condición se ejecuta el código, el cual tiene al final un
incremento.
// Una vez la variable sea 5 y ya no se cumpla la condición dejara de ejecutarse
el bucle.

var i int = 0
for i < 5 {
  XXX
  i++
}
```

- Para **emular un do-while** hay que insertar un condicional dentro del bucle y darle uso al comodín break:

```
// Se inicializa la variable con valor 0
// Una vez dentro del bucle se hace la primera iteración obligatoria. Se hace el
incremento.
// El bucle se repetirá hasta que la condición se cumpla, momento en el que el
break finalizara el bucle.

var j int = 0

for {
 j++
  if j > 4 {
  break
  }
}
```

Funciones

Volver al indice ⤴️

Una función es un **bloque de código que realiza una tarea específica.** Puedes pensar en una función como una especie de "receta" que puedes llamar cuando necesitas realizar una acción en particular.

- Para **definir una función en Go**, utiliza la siguiente sintaxis básica:

```
func nombreDeLaFuncion(parametro1 tipo, parametro2 tipo) tipoDeRetorno {
    // Cuerpo de la función
    return valorDeRetorno
}
```

```
**func:** Palabra clave para declarar una función.

**nombreDeLaFuncion:** El nombre que describe para la función.

**parametro1, parametro2:** Parámetros que la función puede recibir.

**tipoDeRetorno:** Tipo de dato que la función devuelve.

**valorDeRetorno:** Valor que devolvera la función a donde fue invocada.
```

```
func sumar(a int, b int) int {
    resultado := a + b
    return resultado
}
```

- **Llamando a una Función:** Después de definir una función, puedes llamarla desde otras partes de tu programa. Por ejemplo:

```
// Aquí estas llamando a la función "suma", pasandole como parametros el 5 y el 3.
El resultado de la función se almacena en la variable "resultadoSuma".
resultadoSuma := sumar(5, 3)
```

- **Parámetros y Argumentos:** Al definir una función puedes indicar los valores que la función espera recibir cuando es llamada. Los valores que le pases a la función podran ser usados dentro de esta. Los valores reales que pasas a una función se llaman argumentos.

```
func saludar(nombre string) {
// Cuerpo de la función
}

// Llamando a la función con un argumento
saludar("Juan")
```

- **Ámbito de las Variables:** Este concepto ya fue explicado en un punto anterior, pero no viene mal recordarlo. En el caso de que una variable se genere dentro de una función, solo existira dentro de dicha función y una vez finalice la función, desaparecera.

```
package main

func suma {
    // Variables locales dentro de la función suma
    // Sus ámbitos están limitados a esta función
    numeroUno = 10
    numeroDos = 5
    resultado = numeroUno + numeroDos
    return resultado
}

func main() {
    // La variables numeroUno y numeroDos no son accesibles en la función main
    // Generaría un error si se intenta acceder a ellas aquí
}
```

- **Funciones ya preestablecidas:**

fmt.Println() - Imprimir en Consola:

> La función Println se utiliza para imprimir en la consola y agrega una nueva línea al final.

```
package main

import "fmt"

func main() {
    fmt.Println("¡Hola, mundo!")
}
```

fmt.Sprintf() - Formatear Cadenas de Texto:

> La función Sprintf se utiliza para formatear cadenas de texto sin imprimir en la consola.

```
package main

import "fmt"

func main() {
    nombre := "Juan"
    edad := 25
    mensaje := fmt.Sprintf("Hola, mi nombre es %s y tengo %d años.", nombre, edad)
    fmt.Println(mensaje)
}
```

make() - Crear Slices:

> La función make se utiliza para inicializar y asignar memoria para Slices.

```
package main

import "fmt"

func main() {
    // Crear un slice de enteros con longitud 3
    miSlice := make([]int, 3)
}
```

append() - Agregar Elementos a Slices:

> La función append se utiliza para agregar elementos a un slice.

```
package main

import "fmt"

func main() {
    miSlice := []int{1, 2, 3}
    miSlice = append(miSlice, 4, 5)
    fmt.Println(miSlice) // Imprimirá: [1 2 3 4 5]
}
```

len() - Obtener Longitud de Slices y Array:

> La función len devuelve la longitud de un slice o array.

```
package main

import "fmt"

func main() {
    miSlice := []int{1, 2, 3, 4, 5}
    longitud := len(miSlice)
    fmt.Println("La longitud del slice es:", longitud) // Imprimirá: La longitud
del slice es: 5
```

- **Recursividad:** La recursividad es un concepto en programación donde una función se llama a sí misma, ya sea directa o indirectamente.

> Hay que tener extremo cuidado a la hora de manejar código recursivo ya que si no se define correctamente la condición de salida (también llamada condición base) se generaría un bucle infinito y el programa daría error.

```
package main

import "fmt"

func imprimirNumeros(n int) {
    if n <= 0 {
        return
    }
    imprimirNumeros(n - 1)
    fmt.Println(n)
}

func main() {
    numeroMaximo := 5
    fmt.Printf("Imprimiendo números del 1 al %d de forma recursiva:\n",
numeroMaximo)
    imprimirNumeros(numeroMaximo)
}
```

> La función "imprimirNumeros" se llama a sí misma de manera recursiva para imprimir los números del 1 al n. La condición base "if n <= 0" detiene la recursividad.

Structs

Volver al indice ⤴️

Un struct es una **forma de organizar datos relacionados.** Es como una caja con compartimentos, cada uno con un nombre y espacio para un tipo específico de información. Cada "celda" dentro del contenedor tiene un nombre y puede contener un valor. Esto hace que sea fácil agrupar datos que pertenecen juntos.

> [!NOTE]
> En el proximo punto de la documentación se tratara la Programación Orientada a Objetos y los Structs tiene un papel muy importante.

```
// Ejemplo de un struct en Go
type Persona struct {
    Nombre string
    Edad   int
    Altura float64
}
```

Has creado un **struct, o tipo,** llamado Persona que tiene tres compartimentos: Nombre (cadena de texto), Edad (número entero) y Altura (número decimal).

- **¿Cómo usar un Struct?**

Para utilizar un struct, primero necesitas **crear una instancia de él**. Es como crear una **copia de la caja con los compartimentos**, y luego llenar esos compartimentos con datos.

```
// Crear una nueva persona
miPersona := Persona{
    Nombre: "Juan",
    Edad:   35,
    Altura: 1.75,
}
```

- **Acceder a los Datos del Struct**

Puedes acceder a los datos de un struct **utilizando el nombre del compartimento**.

```
fmt.Println("Nombre:", miPersona.Nombre)
fmt.Println("Edad:", miPersona.Edad)
fmt.Println("Altura:", miPersona.Altura)
```

> Esto imprimirá en la consola la información asociada a cada compartimento.

Programación orientada a objetos (POO)

Volver al indice ⤴️

La programación orientada a objetos, abreviando, **POO**, es uno de los conceptos más importantes dentro de cualquier lenguaje de programación.

Existe unos conceptos genericos a la hora de hablar de la POO, pero **GO no sigue al pie de la letra dichos conceptos genericos**, por esa razón antes de empezar de la POO de GO lo más adecuado es hablar de como suele ser la POO de normalmente.

Programación orientada a objetos generica

La programación orientada a objetos es un **paradigma de programación que busca convertir objetos del mundo real en códigos de programación de manera abstracta.**

Vamos a usar de ejemplo un coche. Imagínate que quieres escribir un código de programación sobre un objeto coche, pues podrías escribir un código que hiciera referencia a las **características físicas del coche (propiedades)** y a las **acciones que este puede realizar (métodos).**

- **Abstracción y Clases**

La abstracción es el concepto de **crear clases que representen entidades abstractas en código de programación.** Una **clase es una plantilla que define las características y el comportamiento que tendrá un objeto.** Una forma simple de explicarlo sería que una clase es un molde con el cual puedes crear objetos que representan cosas de la vida real.

```
public class Coche {
    // Atributos (características)
    String marca;
    String modelo;
    int año;

    // Métodos (comportamientos)
    public void arrancar() {
        System.out.println("El coche ha arrancado.");
    }

    public void detener() {
        System.out.println("El coche se ha detenido.");
    }
}
```

- **Constructor**

Un constructor es un **método especial dentro de una clase** que tiene el mismo nombre que la clase y se utiliza para **inicializar los atributos de un objeto** cuando se crea una instancia de la clase.

```
public class Coche {
    // Atributos (características)
    String marca;
    String modelo;
    int año;

    // Constructor
    public Coche(String marca, String modelo, int año) {
        this.marca = marca;
        this.modelo = modelo;
        this.año = año;
    }
}
```

- **Objeto**

Una vez ya tienes una clase creada, puedes usarla para generar objetos, que **son instancias de la clase**. En el ejemplo anterior de coche, puedes crear una instancia de la clase y llamarla, por ejemplo, “miCoche”. “miCoche” sería un nuevo objeto y puedes definir sus propiedades y métodos.

```
// Crear un objeto coche
Coche miCoche = new Coche();

// Estableces sus propiedades
miCoche.marca = "Toyota";
miCoche.modelo = "Corolla";
miCoche.año = 2022;

// Puedes llamar a sus métodos
miCoche.arrancar();
miCoche.detener();
```

- **Herencia**

La herencia de clases permite que **una clase herede las características y las acciones de otra clase.** Tiene cierta similitud a como sería en la vida real, una clase padre puede heredar sus características y habilidades a una clase hija. La herencia suele usarse cuando una clase tiene características muy genéricas y quieres crear una clase más especifica que las compartan. Por ejemplo, si tienes la clase padre coche, la cual puede hacer referencia a muchos tipos de coches, y creas la clase hija coche deportivo. El coche tiene ruedas, puertas y otras cosas muy genéricas, pues si creas la clase coche deportivo que herede de coche, el coche deportivo tendrá las mismas características genéricas, pero además se le podrán agregar otras nuevas exclusivas de él.

La palabra clave para que una clase herede de otra es extends

```
public class CocheDeportivo extends Coche {
    // Nuevas características específicas de coches deportivos
    boolean esDeportivo = true;
}
```

> En este ejemplo, CocheDeportivo tendrá, aunque no se muestre a simple vista, todas las propiedades y métodos de coche, además de los suyos propios.

- **Polimorfismo**

El polimorfismo significa **“muchas formas”** y se refiere a que **un objeto puede comportarse de distintas formas dependiendo del contexto.**

Se trata de un concepto difícil de explicar, pero con un ejemplo práctico se entiende mucho mejor. Supón que tienes una clase coche y dicha clase tiene el método arrancar() que al llamarlo muestra por pantalla el mensaje “Arrancando el coche”:

```
public class Coche {
    public void arrancar() {
        System.out.println("Arrancando el coche");
    }
}
```

El polimorfismo permite que, por ejemplo, si dos clases heredan de coche, lo que significaría que heredan el método arrancar() ambas, pero el método es sobreescrito por cada clase y hacen un uso único de él.

```
public class CocheElectrico extends Coche {
    @Override
    public void arrancar() {
        System.out.println("Arrancando el coche eléctrico");
    }
}

public class CocheGasolina extends Coche {
    @Override
    public void arrancar() {
        System.out.println("Arrancando el coche de gasolina");
    }
}
```

> La clase Coche Eléctrico sobreescribe el método y cuando haga uso de él lo que se mostrara por pantalla es “Arrancando el coche eléctrico”, mientras que la clase coche Gasolina también sobreescribe el método y en su caso lo que se mostraría por pantalla es “Arrancando el coche de gasolina”

- **Encapsulamiento**

El encapsulamiento se usa cuando queremos **darle privacidad a una clase**, es decir, se refiere a la **ocultación de los detalles internos de una clase, protegiendo sus atributos.**

A la hora de crear una clase se debe definir quienes tendrán acceso a sus atributos y características internas. **No es seguro que una clase sea pública ya que cualquiera podría modificarla** y eso provocaría que también cambien los objetos que se instancien de ella. Para evitarlo se suelen poner privadas y crear unos métodos especiales que son los encargados de obtener y modificar los atributos y características internas:

- **Getter y setter**

Los métodos públicos getter y setter son un estándar que se usa cuando una clase está encapsulada y **permite un acceso controlado y aplicar lógica adicional** si es necesario.

```
public class Coche {
    private String marca;

    // Métodos getters y setters
    public String getMarca() {
        return marca;
    }

    public void setMarca(String marca) {
        this.marca = marca;
    }
}
```

Programación orientada a objetos en GO

La razón de haber hablado de la POO desde un punto de vista estándar y no hablar directamente de la POO en GO es porque aunque es un lenguaje de programación que incluye características de la POO, **tiene un enfoque diferente en comparación con otros lenguajes que la usan de manera más tradicional.**

- **Abstracción. Structs en lugar de Clases**

Podemos generar códigos que representen de manera abstracta cosas de la vida real, pero **en GO no existen las clases** como en el resto de lenguajes de programación con POO, **en su lugar se hace uso de los Structs**, de los cuales ya se habló anteriormente.

Puedes crear un Struct para definir lo que en la POO estándar sería las propiedades de la clase. Para generar los métodos se hace uno de funciones.

```
type Coche struct {
    Modelo string
    Anio   int
    Velocidad int
}

func (c *Coche) Arrancar() {
    fmt.Println("El coche", c.Modelo, "ha arrancado.")
}
```

> De esta manera estamos definiendo lo que en la POO estándar sería una clase coche con sus características y tu método “arrancar()

- **Encapsulamiento usando paquetes**

En el apartado “Paquetes e importaciones” se explicó que GO se organiza en paquetes que ayudaban a su gestión y modularización. Gracias a este concepto es posible crear **un Struct con sus características y métodos, almacenarlo en un paquete y decidir que partes serán o no visibles fuera del paquete.** Si el nombre del identificador comienza con una letra mayúscula, es exportado y visible fuera del paquete, pero si el identificador comienza con una letra minúscula, es privado y solo es visible dentro del paquete en el que está definido.

```
package automovil

type Coche struct {
    Modelo    string // mayuscula (puede ser accedido desde fuera del paquete)
    anio      int    // minuscula (privado, solo accesible dentro del paquete)
    velocidad int    // minuscula (privado, solo accesible dentro del paquete)
}
```

- **Composición de tipos en lugar de herencia**

En GO no existe la herencia, pero se permite crear **nuevos Structs o tipos componiéndolos de tipos ya existentes** incluyéndolos como campos en la nueva estructura.

```
// Vehiculo es un tipo base
type Vehiculo struct {
    Marca  string
    Modelo string
}

// Coche es un tipo que compone Vehiculo
type Coche struct {
    Vehiculo
    Puertas int
}

func main() {
    // Crear una instancia de Coche
    miCoche := Coche{
        Vehiculo: Vehiculo{
            Marca:  "Toyota",
            Modelo: "Camry",
        },
        Puertas: 4,
    }
```

> En este ejemplo, Coche tiene todos los campos de Vehiculo, lo cual simularía la herencia, y además tiene su propio campo Puertas. Esto se conoce como composición de tipos.

- **Polimorfismo a través de interfaces**

En GO **se puede lograr el polimorfismo a través de interfaces**. Cuando se crea una interface lo que se define es un conjunto de métodos y cualquier Struct o tipo que implemente esos métodos significa que está implementando esa interfaz.

Supongamos que tenemos una interfaz llamada Conductor con un método Conducir:

```
type Conductor interface {
    Conducir()
}
```

Ahora, creamos dos tipos diferentes: CocheNormal y CocheDeportivo. Ambos implementan el método Conducir:

```
type CocheNormal struct {
    Modelo string
}

func (c CocheNormal) Conducir() {
    fmt.Println("Conduciendo un coche normal:", c.Modelo)
}
```

```
type CocheDeportivo struct {
    Modelo string
}

func (c CocheDeportivo) Conducir() {
    fmt.Println("Conduciendo un coche deportivo:", c.Modelo)
}
```

La función RealizarViaje toma un parámetro del tipo Conductor, que significa que puede recibir cualquier tipo que implemente la interfaz Conductor.

```
func RealizarViaje(c Conductor) {
    c.Conducir()
}
```

Aunque RealizarViaje espera un Conductor, puede trabajar con diferentes tipos de coches sin necesidad de cambiar su implementación. Esto es polimorfismo en acción: tratamos diferentes tipos de coches de manera uniforme a través de la interfaz Conductor.

```
cocheNormal := CocheNormal{Modelo: "Sedan"}
cocheDeportivo := CocheDeportivo{Modelo: "Deportivo"}

RealizarViaje(cocheNormal)    // Conduce un coche normal
RealizarViaje(cocheDeportivo) // Conduce un coche deportivo
```

CRUD

Volver al indice ⤴️

CRUD es el **acrónimo de "Create" (crear), Read (Leer), Update (Actualizar) y Delete (Borrar).** Este concepto sirve para definir las cuatro operaciones básicas que pueden realizarse en la mayor parte de las bases de datos.

Estas operaciones permiten crear nuevos datos (Create), leer los ya existentes (Read), actualizarlos (Update) y eliminarlos (Delete).

Es una función básica pero al mismo tiempo imprescindible para muchos sistemas de información, ya que **permite realizar tareas básicas de mantenimiento y gestionar los datos de las bases de datos.**

CRUD en GO

Para finalizar la documentación se hará una pequeña práctica en la cual se realizara un CRUD usando el lenguaje de programación GO. Será un CRUD básico de una sola entidad y para la conexión a base de datos se usará Mysql.

Los **pasos previos** antes de iniciar el proyecto son **Instalar GO**, lo cual ya está explicado en uno de los primeros puntos de la documentación e **Instalar Mysql**

- **Instalar Mysql en Windows**

Para instalar Mysql en Windows puedes **descargar el instalador desde la web oficial** Una vez descargado simplemente se debe ejecutar el instalador e ir siguiendo los pasos que se indican. La parte más importante de la instalación es cuando te pide que indiques una contraseña para Mysql, esta será la contraseña que se usara para realizar más tarde la conexión con la base de datos.

- **Instalar Mysql en Linux**

Para instalar mysql en Linux se usarán los siguientes comandos:

```
sudo apt update
sudo apt install mysql-server mysql-client
```

Una vez tienes instalado Mysql escribe en el terminal los comandos:

```
sudo mysql -u root -p
use mysql;
ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY 'root';
FLUSH PRIVILEGES;
exit
```

Tras realizar estos cambios solo quedaría reiniciar el servicio de mysql y acceder. El usuario y la contraseña serán “root”.

```
sudo service mysql restart
mysql -u root -p
```

- **Creación de la base de datos**

Para acceder a Mysql se debe abrir un terminal de consola y ejecutar el comando:

```
 mysql -u root -p
```

Una vez conectado a Mysql, se procede a crear la base de datos y la entidad a la que realizaremos el CRUD:

```
create database if not exists agenda;
use agenda;
create table if not exists agenda(
  id bigint unsigned not null auto_increment,
  nombre varchar(255) not null,
  direccion varchar(255) not null,
  correo_electronico varchar(255) not null,
  primary key(id)
);
```

- **Creación del proyecto en GO e importancia de libreria**

Para empezar se debe **crear una carpeta donde se iniciara el proyecto**. Una vez creada, a través del terminal de comandos, te debes **situar dentro de la carpeta** y ejecutar el siguiente comando **“go mod init ”**

> Por convenio, en la X se suele poner la cuenta de github donde guardaras el proyecto. Suponiendo que tu cuenta de github se llamase “Juan” y quisieras llamar al proyecto “crud” el comando sería:

```
go mod init github/juan/crud
```

Sabrás que el comando ha funcionado y tienes tu directorio de proyecto listo si dentro de la carpeta ha aparecido un archivo llamado **go.mod**, el cual no debemos tocar.

A continuación debes **descargar una librería que necesitaras** para realizar la conexión con mysql:

```
go get -u github.com/go-sql-driver/mysql
```

- **Proyecto crud**

Debes **crear manualmente un archivo, con el nombre que quieras, pero que tenga la extensión .go**, ahí es donde se escribirá el código del proyecto crud.

Se te va a presentar el código del proyecto paso a paso y al final del todo se facilitará el código completo.

Importación

Al inicio **debes importar en el proyecto aquellas librerías que serán necesarias mas adelante**, como son la que te permite interactuar con la base de datos mysql, imprimir mensajes por pantalla y trabajar con la entrada/salida de datos.

```
import (
    "database/sql"
    "fmt"
    _ "github.com/go-sql-driver/mysql"
    "os"
)
```

Creación del Struct Contacto

El siguiente paso es **crear una estructura de datos** que será la que represente un contacto

```
type Contacto struct {
    Id              int
    Nombre          string
    Direccion       string
    CorreoElectronico string
}
```

Función "obtenerBaseDatos"

Se crea una función que será la encargada de realizar la **conexión con la base de datos**

```
func obtenerBaseDeDatos() (*sql.DB, error) {
    db, err := sql.Open("mysql", "root:@tcp(127.0.0.1:3306)/agenda")
    if err != nil {
        return nil, err
    }
    return db, nil
}
```

Función “main”

Esta es la **función principal del proyecto.** Es donde estará el menú que te permitirá interactuar con el crud.

```
func main() {
    menu := `¿Qué deseas hacer?
[1] -- Insertar
[2] -- Mostrar
[3] -- Actualizar
[4] -- Eliminar
[5] -- Salir
-----> `

    var eleccion int
    var c Contacto

    for eleccion != 5 {
        fmt.Println(menu)
        fmt.Scanln(&eleccion)

        switch eleccion {
        case 1:
            c = leerContacto()
            if err := insertar(c); err != nil {
                fmt.Printf("Error insertando: %v\n", err)
            } else {
                fmt.Println("Insertado correctamente")
            }
        case 2:
            mostrarContactos()
        case 3:
            c = leerContacto()
            if err := actualizar(c); err != nil {
                fmt.Printf("Error actualizando: %v\n", err)
            } else {
                fmt.Println("Actualizado correctamente")
            }
        case 4:
            c = leerID()
            if err := eliminar(c); err != nil {
                fmt.Printf("Error eliminando: %v\n", err)
            } else {
                fmt.Println("Eliminado correctamente")
            }
        }
    }
}
```

Funciones del CRUD

Estas son las **funciones que dan funcionalidad al CRUD y que son llamadas a través del menu de la funcion principal main:** "leerContacto", "leerID", "insertar", "mostrarContacto", "obtenerContactos", "actualizar" y "eliminar":

```
func leerContacto() Contacto {
    var c Contacto
    fmt.Println("Ingresa el nombre:")
    fmt.Scanln(&c.Nombre)
    fmt.Println("Ingresa la dirección:")
    fmt.Scanln(&c.Direccion)
    fmt.Println("Ingresa el correo electrónico:")
    fmt.Scanln(&c.CorreoElectronico)
    return c
}

func leerID() Contacto {
    var c Contacto
    fmt.Println("Ingresa el ID:")
    fmt.Scanln(&c.Id)
    return c
}

func insertar(c Contacto) error {
    db, err := obtenerBaseDeDatos()
    if err != nil {
        return err
    }
    defer db.Close()

    _, err = db.Exec("INSERT INTO agenda (nombre, direccion, correo_electronico)
VALUES(?, ?, ?)", c.Nombre, c.Direccion, c.CorreoElectronico)
    return err
}

func mostrarContactos() {
    contactos, err := obtenerContactos()
    if err != nil {
        fmt.Printf("Error obteniendo contactos: %v\n", err)
        return
    }

    for _, contacto := range contactos {
        fmt.Println("====================")
        fmt.Printf("Id: %d\n", contacto.Id)
        fmt.Printf("Nombre: %s\n", contacto.Nombre)
        fmt.Printf("Dirección: %s\n", contacto.Direccion)
        fmt.Printf("E-mail: %s\n", contacto.CorreoElectronico)
    }
}
```

```
func obtenerContactos() ([]Contacto, error) {
    var contactos []Contacto
    db, err := obtenerBaseDeDatos()
    if err != nil {
        return nil, err
    }
    defer db.Close()

    rows, err := db.Query("SELECT id, nombre, direccion, correo_electronico FROM
agenda")
    if err != nil {
        return nil, err
    }
    defer rows.Close()

    for rows.Next() {
        var c Contacto
        err := rows.Scan(&c.Id, &c.Nombre, &c.Direccion, &c.CorreoElectronico)
        if err != nil {
            return nil, err
        }
        contactos = append(contactos, c)
    }

    return contactos, nil
}

func actualizar(c Contacto) error {
    db, err := obtenerBaseDeDatos()
    if err != nil {
        return err
    }
    defer db.Close()

    _, err = db.Exec("UPDATE agenda SET nombre = ?, direccion = ?,
correo_electronico = ? WHERE id = ?", c.Nombre, c.Direccion, c.CorreoElectronico,
c.Id)
    return err
}

func eliminar(c Contacto) error {
    db, err := obtenerBaseDeDatos()
    if err != nil {
        return err
    }
    defer db.Close()

    _, err = db.Exec("DELETE FROM agenda WHERE id = ?", c.Id)
    return err
}
```

CODIGO CRUD COMPLETO

```go
package main

import (
    "database/sql"
    "fmt"
    _ "github.com/go-sql-driver/mysql"
    "os"
)

type Contacto struct {
    Id               int
    Nombre           string
    Direccion        string
    CorreoElectronico string
}

func obtenerBaseDeDatos() (*sql.DB, error) {
    db, err := sql.Open("mysql", "root:@tcp(127.0.0.1:3306)/agenda")
    if err != nil {
        return nil, err
    }
    return db, nil
}

func main() {
    menu := `¿Qué deseas hacer?
[1] -- Insertar
[2] -- Mostrar
[3] -- Actualizar
[4] -- Eliminar
[5] -- Salir
-----> `

    var eleccion int
    var c Contacto

    for eleccion != 5 {
        fmt.Println(menu)
        fmt.Scanln(&eleccion)

        switch eleccion {
        case 1:
            c = leerContacto()
            if err := insertar(c); err != nil {
                fmt.Printf("Error insertando: %v\n", err)
            } else {
                fmt.Println("Insertado correctamente")
            }
        case 2:
            mostrarContactos()
```

```
        case 3:
            c = leerContacto()
            if err := actualizar(c); err != nil {
                fmt.Printf("Error actualizando: %v\n", err)
            } else {
                fmt.Println("Actualizado correctamente")
            }
        case 4:
            c = leerID()
            if err := eliminar(c); err != nil {
                fmt.Printf("Error eliminando: %v\n", err)
            } else {
                fmt.Println("Eliminado correctamente")
            }
        }
    }
}

func leerContacto() Contacto {
    var c Contacto
    fmt.Println("Ingresa el nombre:")
    fmt.Scanln(&c.Nombre)
    fmt.Println("Ingresa la dirección:")
    fmt.Scanln(&c.Direccion)
    fmt.Println("Ingresa el correo electrónico:")
    fmt.Scanln(&c.CorreoElectronico)
    return c
}

func leerID() Contacto {
    var c Contacto
    fmt.Println("Ingresa el ID:")
    fmt.Scanln(&c.Id)
    return c
}

func insertar(c Contacto) error {
    db, err := obtenerBaseDeDatos()
    if err != nil {
        return err
    }
    defer db.Close()

    _, err = db.Exec("INSERT INTO agenda (nombre, direccion, correo_electronico)
VALUES(?, ?, ?)", c.Nombre, c.Direccion, c.CorreoElectronico)
    return err
}

func mostrarContactos() {
    contactos, err := obtenerContactos()
    if err != nil {
        fmt.Printf("Error obteniendo contactos: %v\n", err)
        return
    }
```

```
    for _, contacto := range contactos {
        fmt.Println("====================")
        fmt.Printf("Id: %d\n", contacto.Id)
        fmt.Printf("Nombre: %s\n", contacto.Nombre)
        fmt.Printf("Dirección: %s\n", contacto.Direccion)
        fmt.Printf("E-mail: %s\n", contacto.CorreoElectronico)
    }
}

func obtenerContactos() ([]Contacto, error) {
    var contactos []Contacto
    db, err := obtenerBaseDeDatos()
    if err != nil {
        return nil, err
    }
    defer db.Close()

    rows, err := db.Query("SELECT id, nombre, direccion, correo_electronico FROM
agenda")
    if err != nil {
        return nil, err
    }
    defer rows.Close()

    for rows.Next() {
        var c Contacto
        err := rows.Scan(&c.Id, &c.Nombre, &c.Direccion, &c.CorreoElectronico)
        if err != nil {
            return nil, err
        }
        contactos = append(contactos, c)
    }

    return contactos, nil
}

func actualizar(c Contacto) error {
    db, err := obtenerBaseDeDatos()
    if err != nil {
        return err
    }
    defer db.Close()

    _, err = db.Exec("UPDATE agenda SET nombre = ?, direccion = ?,
correo_electronico = ? WHERE id = ?", c.Nombre, c.Direccion, c.CorreoElectronico,
c.Id)
    return err
}

func eliminar(c Contacto) error {
    db, err := obtenerBaseDeDatos()
    if err != nil {
        return err
```

```
    }
    defer db.Close()

    _, err = db.Exec("DELETE FROM agenda WHERE id = ?", c.Id)
    return err
}
```
