Golang Teamcubation Tutorial

Recursos

1. Instalación
    a. https://go.dev/doc/install
2. Visual Code
    a. Extensiones recomendadas:
        i. Docker
        ii. Docker Linter
        iii. GitHub Pull Requests and Issues
        iv. Go
        v. Go Autotest
        vi. Go Doc
        vii. Go Outliner
        viii. Go Test Explorer
        ix. Linter
        x. Prettier
        xi. Thunder Client
        xii. TODO Highlight
3. Documentation, ejemplos y comunidad
    a. https://go.dev/doc/#
    b. https://golang.org/doc/effective_go.html
    c. http://goconejemplos.com/
    d. https://forum.golangbridge.org/
    e. https://go.dev/ref/spec
    f. https://roadmap.sh/golang
    g. https://gin-gonic.com/docs/
4. Repositorio
    a. https://github.com/devpablocristo/golang
5. Playground
    a. https://goplay.tools
6. Librería estándar
    a. https://pkg.go.dev/std

Instalacion

Descarga Golang e Instalar

- https://go.dev/doc/install

IDEs

- https://code.visualstudio.com/
- https://www.jetbrains.com/go/download/

Metodología de Trabajo

- Scrum / Kanban
- Dailies
- Retro
- Sprints
- Plannings
- Estimaciones
- 1on1

Onboarding a GO

Características

Go, también conocido como Golang, es un lenguaje de programación moderno y eficiente desarrollado en Google. Su diseño se centra en la simplicidad, la eficiencia y la facilidad de uso, con un énfasis particular en la concurrencia.

Historia y Origen de Go

Go fue creado en Google por Robert Griesemer, Rob Pike y Ken Thompson. El lenguaje se lanzó en noviembre de 2009, con el objetivo de abordar desafíos específicos de programación a gran escala que enfrentaba Google. Go combina la eficiencia de los lenguajes compilados como C con la facilidad de uso de lenguajes como Python.

Los Creadores de Go

- Robert Griesemer: Aportó su experiencia en sistemas de ejecución y lenguajes de programación, habiendo trabajado en proyectos como V8 y HotSpot JVM.
- Rob Pike: Reconocido por su contribución al desarrollo de Unix y co-creador del UTF-8.
- Ken Thompson: Co-creador del sistema operativo Unix y del lenguaje de programación B.

Google y Go

Aunque Go comenzó como un proyecto interno en Google, rápidamente ganó popularidad en la comunidad de código abierto. Google continúa siendo un actor principal en el desarrollo de Go, utilizando el lenguaje en varios de sus sistemas y servicios internos. Proyectos notables como Kubernetes y Docker están construidos utilizando Go.

Conceptos Clave en Go

Sintaxis Básica

Go ofrece una sintaxis clara y concisa. Aquí hay algunos ejemplos básicos:

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

Metas Básicas de Go (Golang)

Go (Golang) es un lenguaje de programación moderno que se centra en hacer que el desarrollo de software sea eficiente y accesible. Fue diseñado con tres metas fundamentales en mente:

1. Compilación Eficiente

La eficiencia en la compilación es una característica clave de Go. Esto se refleja en:

1. **Tiempo de Compilación Rápido:** Go está diseñado para compilar código rápidamente. Esto se logra a través de una serie de decisiones de diseño, como un modelo de dependencias simplificado y una sintaxis que facilita el análisis del compilador.
2. **Simplicidad en la Gestión de Dependencias:** Con la introducción de módulos en Go, la gestión de dependencias se ha simplificado enormemente, lo que contribuye a una compilación más rápida y menos problemas relacionados con la gestión de librerías externas.

2. Ejecución Eficiente

La ejecución eficiente es otro pilar fundamental en el diseño de Go. Esto incluye:

1. **Gestión de Recursos:** Go utiliza un recolector de basura para la gestión automática de la memoria, lo que ayuda a prevenir fugas de memoria y otros problemas relacionados con la gestión manual de la memoria.
2. **Concurrencia Integrada:** A través de goroutines y canales, Go ofrece un modelo de concurrencia robusto y de alto rendimiento. Esto permite a los desarrolladores escribir programas concurrentes de manera más sencilla, aprovechando de manera eficiente los recursos del sistema.
3. **Compilación a Código Nativo:** Go compila directamente a código máquina, lo que significa que los programas escritos en Go pueden ejecutarse con un alto rendimiento, aprovechando al máximo el hardware subyacente.

3. Programación Sencilla

Go promueve un enfoque de programación sencillo y productivo:

1. **Sintaxis Clara y Concisa:** Go elimina la verbosidad innecesaria en la sintaxis y se enfoca en la claridad. Esto hace que el código sea más fácil de escribir, leer y mantener.
2. **Facilidad de Aprendizaje:** A diferencia de otros lenguajes que pueden tener características complejas y abrumadoras, Go se mantiene simple y coherente, lo que facilita el aprendizaje para los nuevos programadores.
3. **Herramientas Efectivas:** Go viene con un conjunto de herramientas poderosas, pero simples, como el formateador de código `go fmt`, la herramienta de pruebas `go test`, y el sistema de gestión de paquetes `go mod`, que hacen que el desarrollo sea más eficiente y agradable.

¿Go es un lenguaje orientado a objetos?

TODO EN GO ES ACERCA DE TIPOS

Go posee características de programación orientada a objetos, como tipos y métodos, pero no sigue el modelo tradicional con jerarquías de clases. En lugar de ello, Go utiliza interfaces para un enfoque más flexible y general. Go también permite incrustar tipos dentro de otros, lo que proporciona una alternativa sencilla a la herencia de clases. A diferencia de lenguajes como C++ o Java, en Go se pueden definir métodos para cualquier tipo de dato, no solo para tipos primitivos, sin limitarse a estructuras específicas.

Además, la falta de una jerarquía de tipos hace que los "objetos" en Go, se sientan mucho más ligeros que en lenguajes como C ++ o Java.

Características de la OOP

1. **Encapsulamiento:** Consiste en agrupar los datos (atributos) y los métodos (funciones) que operan sobre los datos en una unidad llamada "objeto".

2. **Abstracción:** Se refiere a la capacidad de concentrarse en las características esenciales de un objeto

3. **Herencia:** Permite que una clase (llamada subclase o clase derivada) herede atributos y métodos de otra clase

4. **Polimorfismo:** Significa "muchas formas". En OOP, permite que objetos de diferentes clases sean tratados como objetos de una clase común.

Go tiene aspectos de **OOP**, pero en GO, todo es acerca del **TIPO**.

- No se crean clases, se crean **TIPOS**.
- No se inicializa, se crea un **VALOR** de un **TIPO**.
- Los **TIPOS** son definidos por el usuario.
- Se pueden obtener **VALORES** de **esos TIPOS.**
- Se pueden asignar **VALORES** a **TIPOS** definidos por los usuarios.

Características de OOP en Go

1. **Encapsulamiento:**
    a. Uso de structs para encapsular datos y métodos.
    b. Control de visibilidad mediante identificadores exportados y no exportados (mayúsculas/minúsculas).
2. **Composición (en reemplazo de la herencia):**
    a. Uso de structs embebidos en lugar de la herencia tradicional.
    b. Permite la reutilización y extensión de código.
3. **Polimorfismo:**
    a. A través de interfaces en lugar de herencia.
    b. Permite el tratamiento de diferentes tipos como un mismo tipo de interfaz.
4. **Métodos:**
    a. Definición de comportamientos asociados con tipos (incluyendo tipos primitivos y structs).
5. **Interfaces:**
    a. Definición de contratos que otros tipos pueden implementar.
    b. Permite un enfoque flexible y desacoplado en el diseño de software.

Partes fundamentales

Todo proyecto de Go consta de al menos 3 partes fundamentales:

- Funciones.
- Paquetes.
- Módulos.

Funciones

- Una función, que será el ámbito, se desarrolla el algoritmo deseado.

Paquetes

- Lo primero que aparece en cualquier archivo de Go es la siguiente:
  - <package nompkg>
    - Define el paquete al cual pertenece el código (en ese caso el paquete se llama 'nompkg') o sea, el scope o ámbito de las variables, funciones, etc que se creen.
    - El caso más simple es el del <package main> ya que todo proyecto go debe utilizar si o si el paquete main.
    - aaaTodo paquete debe cumplir solo una funcion, por ejemplo se creamos el paquete "calculadora", en el se desarrollaran todas las tareas relacionadas con una calculadora y solo sobre una calculadora.
- La segunda parte define cuales son los paquetes que importamos para utilizar en nuestro proyecto.

```
import   "fmt"
```

```
import (
    "fmt"
    "context"
)
```

Módulos

Los módulos son grupos de paquetes que definen en el archivo go.mod.

El archivo go.mod se crea de forma automatizada con el comando:

```
$ go mod init nombre-del-proyecto
```

En Go el nombre del proyecto suele configurarse con la ruta a un repositorio, ejemplo:

```
$ go mod init github.com/github-username/my-app
```

En resumen, los módulos son grupos de paquetes, los paquetes son grupos de funciones,

Todas estas partes podemos verlas fácilmente con el ejemplo clásico 'Hello world'

- https://goplay.tools/snippet/ZgFR4mUil_g

Tipos, variables y valores

Tipos, variables y valores

Tipos

En Go existen los siguientes grupos de tipos primitivos:

1. Boolean types
2. Numeric types
3. String types
4. Array types
5. Slice types
6. Struct types
7. Pointer types
8. Function types
9. Interface types
10. Map types
11. Channel types

Cada grupo tiene diferentes tipos y cada tipo tiene características determinadas.

Se desarrollarán cada uno más adelante.

Los tipos son la piedra angular del lenguaje.

Variables

En Go NO se puede tener variables sin usar.

https://goplay.tools/snippet/fU4MTRxHqEx

Si por algún motivo es necesario declarar una variable, pero todavía no se puede utilizar el carácter "_".

https://goplay.tools/snippet/Q3uO-ZmUTgo

Short declaration operator

En Go hay varias formas de declarar una variable, una muy utilizada es el "short declaration operator". Solo se usa la primera vez que declara e inicializa la variable con algún valor. Solo puede ser utilizado dentro del ámbito o scope de una función.

- " := "
  - Declara variables y asigna un valor.
- x : = 42
  - Declara la variable 42 y le asigna un valor
- x = 12
  - La segunda vez que se asigna un valor se usa solo "=" porque ya fue declarada.

https://goplay.tools/snippet/9Zwpc8RmNyn

The var keyword

La palabra reservada 'var' se utiliza para declarar variables.

https://goplay.tools/snippet/DAdrybYvQ7q

El ámbito (“scope”) de “**a”** es toda la extensión del paquete, o sea, es una variable global, mientras que el del “**b”** solo es la función main. Es importante notar que en fuera del ámbito de una función no se puede utilizar el short declaration operator.

nota: evitar usar global scope, suele dar problemas.

Explorando los tipos

https://goplay.tools/snippet/gloyfLFuubl

Decimos que un lenguaje es de tipado estático, porque los tipos tienen que definirse en tiempo de compilación para que el programa funcione.

    - O sea, si por ejemplo definimos una variable de tipo string esta solo podrá contener instancias de este tipo, y si no es así, el compilador nos mostrará un error.

- Lo mismo puede pasar cuando llamamos a un método o función. Si no existe o hemos escrito mal el nombre, el compilador nos mostrará un error y no nos dejará ejecutar el programa hasta que lo arreglemos.
- **zero value:** En Golang es el valor predeterminado que se le asigna a una variable si no se inicializa manualmente o si el valor de esta no es proporcionado por el usuario.

Alias de un tipo, Creando tu propio tipo

Es posible crear un alias de un tipo, esto puede ser útil por diversos motivos, por ejemplo para que sea más

clara el uso de un tipo, ej.:

- **type UserID int**: en este caso UserID es simplemente otra forma a la que se podrá referirse al tipo int, pero cuando se utilice, al tener un nombre explícito, será más fácil identificar su utilización.
- **var x UserID**: x es de tipo UserID.

En Go se pueden crear nuevos tipos con diversas características, esta habilidad es muy útil.

- https://goplay.tools/snippet/AL_AmrqK5Ai

Conversión

Se puede convertir de un tipo en otro.

- https://goplay.tools/snippet/REmS6mrur_d

Constantes

En Golang, las constantes son valores fijos e inmutables que se declaran en el programa y cuyo valor no puede ser cambiado durante la ejecución del mismo.

- Sintaxis: const pi = 3.1416
- https://goplay.tools/snippet/uHwaWbOYgDE

Types 1 (bool, int y string)

Bool type

Un bool es un tipo de dato que solo puede tener dos valores posibles: true (verdadero) o false (falso).

- Zero value: false.
- https://goplay.tools/snippet/xk7uN_0wD8u

Numeric types

Hay una variedad de tipos numéricos, el más estándar de todos ellos el tipo int, con el que se declaran enteros positivos como negativos.

- https://go.dev/ref/spec#Numeric_types
- https://goplay.tools/snippet/MoqfdZk0qKG

String type

Un string es una secuencia de bytes que representa un texto y se utiliza para representar datos de caracteres, como nombres, direcciones, mensajes, etc.

- https://goplay.tools/snippet/n7WfDMHK1xK

Runes

Una rune es un tipo de datos que representa un punto de código Unicode. Go admite cadenas UTF-8 y cada carácter individual en una cadena se representa como su valor Unicode. Un valor de rune es el valor numérico del carácter Unicode correspondiente.

- var letter rune = 'a'
- En este caso, letter tendrá el valor numérico de la letra 'a', que es 97.

IOTA

En Golang, iota es una constante predeclarada que se utiliza en la definición de enumeraciones o constantes relacionadas. Iota se inicializa en 0 y se incrementa automáticamente por uno cada vez que se utiliza en una definición de constante.

Aquí hay un ejemplo de cómo se puede utilizar iota para definir una enumeración de días de la semana:

```
type DayOfWeek int

const (
    Sunday DayOfWeek = iota
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
)
```

En este ejemplo, se define un tipo de enumeración llamado DayOfWeek, que es de tipo int. Luego, se utilizan constantes definidas utilizando iota para enumerar los días de la semana. Debido a que Sunday es la primera constante que utiliza iota, se inicializa en 0. Cada constante subsiguiente en la lista utiliza automáticamente el valor de iota incrementado en uno.

Por lo tanto, en este ejemplo, Monday se inicializa con un valor de 1, Tuesday con un valor de 2, y así sucesivamente.

En este otro ejemplo se le da dalores a las difentes magnitudes especificadas.

```
type MagnitudByte float64

const (
    // descarta el primer valor asignándolo al identificador blanco
    _  = iota
    KB MagnitudByte = 1 << (10 * iota)
    MB
    GB
    TB
    PB
    EB
    ZB
    YB
)
```

https://goplay.tools/snippet/B_ms6lt4iHo

Ejercicios 1

01

- Usar el short declarator operator para declarar un entero, un string y un booleano.
- Imprimirlos por pantalla en la misma sentencia.
- Imprimir cada variable por separado.
- Solución: https://goplay.tools/snippet/XweMgC5eFKv

02

- Declarar tres variables globales un entero un string y un booleano.
- Imprimir sus tipos y sus valores por defecto (zeros).
- Solución: https://goplay.tools/snippet/pCE5gbIGp6h

03

- Declarar tres variables en el scope del package y darles un valor inicial.
- Asignar sus valores a un string.
- Imprimir el string.
- Solución: https://goplay.tools/snippet/W95gEAEq33k

04

- Crear un tipo.
- Imprimir valor el ‘zero’ o valor por defecto del tipo creado.
- Imprimir por pantalla el tipo y el valor.
- Darle un valor.
- Imprimir por pantalla.

- Solución: https://goplay.tools/snippet/eAeQIbM3fQt

05

- Crear un nuevo tipo a partir de un entero en el scope del paquete.
- Crear una variable del nuevo tipo.
- Crear un entero en el scope del paquete.
- Asignarles a ambas un valor.
- Convertir una variable al tipo de la otra y sumarlas.
- Imprimir por pantalla el valor y tiempo resultante.
- Solución: https://play.golang.org/p/GVywLj5HfZU

Types 2 (Grouping data: arrays, slices y maps)

NOTA: Al agrupar información

- *Si son del mismo tipo son slices.*
- *Si son de diferente tipo son structs.*

Arrays

La diferencia entre un slice y un array es que estos últimos declaran una cantidad definida de elementos, mientras que los slices no, o sea, la cantidad es dinámica.

Los arrays no son muy utilizados Go, se usan para construir bloques de slices que son los que realmente son finalmente usados en Go. La longitud es parte del tipo.

- Declaración: var x [tamaño]tipo
- https://goplay.tools/snippet/i9geuPX5zIt

Slices

La diferencia fundamental entre los slices y los arrays es que estos primeros son de tamaño dinámico.

- *NOTA: Un composite literal en Go es una sintaxis especial para definir valores de tipo estructurado, como arreglos, mapas y estructuras, directamente en línea en el código fuente.*
  - composite literal slice: `[]tipo{valores}`
  - sliceDeInts := `[]int{4,5,7,9,26}`
- https://goplay.tools/snippet/3tCJVPebgHY

Slices - for range

- https://goplay.tools/snippet/9ZJ8sa8bAg1

Slices - slicing a slice

- https://goplay.tools/snippet/zeIm0BI9fAV

Slices - append to a slice

```
// append = adjuntar (Agregar valores a un slice)
unSlice := []int{1,2,3}
unSlice = append(unSlice, 44, 11, 55, 88)
// output: [1,2,3,44, 11, 55, 88]
```

- https://goplay.tools/snippet/UnruCgdc0T-

Slices - deleting from a slice

- Para eliminar elementos: append(desde, hasta)
- x = append(x[:2], x[4:]...)
- https://goplay.tools/snippet/0aWYTeb-oRL

Strings

Son un tipo de datos básico que se utiliza para representar texto como una secuencia de bytes inmutable.

1. **Inmutabilidad:** Una vez que se crea un string en Go, su contenido no puede ser cambiado. Esto significa que cualquier operación que parezca modificar un string en realidad crea un nuevo string con los cambios.
2. **Representación Interna:** Internamente, un string en Go se representa como una secuencia de bytes (no caracteres).

3. **Longitud y Acceso:** La longitud de un string se puede obtener con la función `len()`, que devuelve el número de bytes, no el número de caracteres.

4. **Strings y Slice de Bytes:** Existe una relación estrecha entre strings y slice de bytes en Go. Se puede convertir un string en un slice de bytes y viceversa, lo que es útil para operaciones como la manipulación de archivos o la comunicación de red.

- https://goplay.tools/snippet/uS9ef5PzK45

La función Make

Se utiliza para crear y asignar memoria a un nuevo objeto de tipo slice, map o channel con cierta capacidad o tamaño especificado. La sintaxis general de la función es la siguiente:

- ***make(tipo, longitud/capacidad)***
- Donde tipo hace referencia al tipo de dato que se quiere crear (slice, mapa o canal), y longitud/capacidad es un número entero que indica el tamaño inicial del objeto creado.

Para slices, se debe indicar su capacidad, pero también puede aceptar un segundo argumento opcional que indique su longitud inicial.

```
mySlice := make([]int, 5, 10) // Crea un slice con longitud 5 y capacidad 10.
```

Slices - make

- https://goplay.tools/snippet/WjfemrsIEBF
- https://tour.golang.org/moretypes/13

```
unSlice := make(slice, long, cap)
a := make([]int, 5) //a len=5 cap=5 [0 0 0 0 0]
b := make([]int, 0, 5) //b len=0 cap=5 []
c := b[:2] //c len=2 cap=5 [0 0]
d := c[2:5] //d len=3 cap=3 [0 0 0]
```

Slices - multidimensional slice

Un slice multidimensional es una estructura de datos que tiene uno o más slices dentro de otro slice. Cada slice interno puede tener una longitud diferente y pertenece a un slice externo. Por ejemplo:

```
multiSlice := [][]int{{1, 2, 3}, {4, 5, 6}}
```

- *NOTA: Los slices multidimensionales suelen ser difíciles de entender y trabajar.*
- https://goplay.tools/snippet/XoQ1j4xfmVp

Maps

Son almacenes con claves, o sea, se pueden guardar valores “debajo” de claves.

Los maps son muy eficientes para la obtención de valores.

En Go, existen diferentes maneras de crear un mapa, y cada una de ellas tiene ciertas implicaciones en términos de comportamiento y rendimiento.

Crear un Mapa Sin `make`

Cuando creas un mapa directamente sin usar `make`, por ejemplo:

```
var m map[string]int
```

En este caso, `m` se inicializa a su valor cero, que es `nil`. Esto significa que el mapa no está listo para usarse; cualquier intento de añadir elementos a este mapa resultará en un error de tiempo de ejecución (panic), ya que técnicamente no ha sido inicializado.

```
// map[tipo clave] tipo valor
m := map[string]int{
"juan": 10,
"carlos": 14,
}
//m[“juan”] ---> output 10
```

Cuando se crea un mapa en Go de la manera descrita, se inicializa el mapa con valores específicos y el mapa está listo para ser utilizado inmediatamente. Este método es una forma de inicialización literal de un mapa y es equivalente a crear un mapa con `make` y luego añadirle elementos, pero todo en un solo paso.

- https://goplay.tools/snippet/g0YSxcYDeCt

Crear un Mapa con `make` Sin Especificar Capacidad

Si usas `make` para crear un mapa sin especificar una capacidad inicial, como en:

```
m := make(map[string]int)
```

Aquí, `m` es un mapa válido e inicializado, pero sin una capacidad predefinida. El mapa está listo para usarse y se redimensionará automáticamente a medida que agregues elementos. No tiene una capacidad inicial, por lo que Go gestionará el crecimiento del mapa conforme se añadan elementos.

Crear un Mapa con `make` Especificando una Capacidad

Cuando creas un mapa con `make` y especificas una capacidad inicial, como:

```
m := make(map[string]int, 100)
```

En este ejemplo, `m` es un mapa que se inicializa con una capacidad para aproximadamente 100 elementos. Especificar una capacidad inicial puede ser más eficiente en términos de rendimiento, ya que reduce la cantidad de redimensionamientos necesarios a medida que el mapa crece. Es importante notar que esta capacidad no es un límite rígido; el mapa puede crecer más allá de esta capacidad si es necesario, pero Go intentará minimizar las operaciones de redimensionamiento al asignar espacio adicional según esta capacidad inicial.

Map - add element & range

- https://goplay.tools/snippet/uozPgIoUPwD

Map - delete

- https://goplay.tools/snippet/wr6j83_zkiB

El paquete fmt

Este paquete se usa normalmente para imprimir sentencias en pantalla.

fmt viene de format package y es el paquete que se encargará de darle formato a cualquier tipo de entrada o salida de datos. Con entrada y salida de datos, me refiero a stdin y stdout.

Por cierto, fmt cuenta con soporte para caracteres especiales.

Println

La clásica función print, echo, console.log o equivalente, con todo y su newline añadido al final.

Println es capaz de imprimir varios argumentos seguidos, incluso si son de diferente tipo.

```
fmt.Println(1, "Hola Mundo!", nil)
```

Printf

Printf es como Println, pero con operadores de posición que le dicen a Go el tipo de dato que le estamos pasando.

```
texto := "World!"
numero := 42
// %s es de string y %d de digit
fmt.Printf("Hello %s %d", texto, numero)
```

Tipos de Operadores

En Go hay múltiples tipos de operadores para muchísimos tipos de datos. Hay algunos operadores de posición destacables.

- %T, tipo de dato (string, int, booleano, etc.).
- %v, valor en el formato predeterminado de Go.
- %d números.
- %s strings.
- %t, Para booleanos, retorna la palabra, false o true.
- %x, número de base 16.
- %o, número de base 8.
- %e, número, notación científica.
- %9.2f, flotante con ancho de 9 y precisión de 2.
- %.2f, flotante con ancho predeterminado y precisión de 2.
- %q un string o cadena de texto, entre comillas, previamente escapado.

Se pueden ver el resto de operadores en la documentación oficial de Go.

Sprintf

Guarda la misma sintaxis que printf, pero con la diferencia de que no imprime en pantalla, sino que genera un string o cadena de texto.

```
var message string = fmt.Sprintf("Hello %s %d", texto, numero)
```

https://goplay.tools/snippet/bHF_an0d2lD

Scan

Scan leerá el texto de la entrada estándar (stdin) hasta que encuentre el primer espacio o salto de línea y nos retornará el número de argumentos recibidos.

```
var mensaje string
fmt.Scan(&mensaje)
fmt.Println(mensaje)
```

El código anterior hace solo dos cosas: guardar el texto que escribimos en la consola en la variable mensaje y, posteriormente, imprimirlo en pantalla.

Scanf

Scanf es como Scan, pero para guardar múltiples argumentos, separados por espacios.

Mira como primero creamos las tres variables, para después, como primer argumento para Scanf pasarle el orden en el que recibirá los argumentos como un tipo de dato, separados por espacio y al final la dirección de las variables a las cuales los tiene que asignar.

```
var (
nombre   string
apellido string
telefono int
)
// nombre apellido telefono
fmt.Scanf("%s %s %d", &nombre, &apellido, &telefono)
fmt.Printf("Nombre: %s, Apellido: %s, telefono: %d", nombre, apellido, telefono)
```

Control de flujo

Loop - init, condition, post

En Go no existe “while”, se usa “for” los diferentes loops.

```
// Sustituto de while
n := 1
for n < 5 {
    n *= 2
}
fmt.Println(n) // 8 (1*2*2*2)
```

```
// Loop de 3 componentes
sum := 0
for i := 1; i < 5; i++ {
    sum += i
}
fmt.Println(sum) // 10 (1+2+3+4)
```

```
// Loop infinito
sum := 0
for {
    sum++ // repeated forever
}
fmt.Println(sum) // never reach
```

```
// Loop for-each range
// Los bucles que recorren slices, arrays, maps, channels o strings muchas veces son mas // eficientes
con un range loop.
// recorre todo el array del principio al final, asigna en s el contenido de array
palabras := []string{"hello", "world"}
// s := range palabras <--- ojo asi se asigna
for i, s := range palabras {
    // i es la posición del array
   // s el contenido del array
    fmt.Println(i, s)
}
// Output:
// 0 hello
// 1 world
```

Loop - nesting loops

https://goplay.tools/snippet/Zt5IbujYT3H

Loop - for statement

https://golang.org/ref/spec#For_statements

Loop - break & continue

https://goplay.tools/snippet/Zt5IbujYT3H

Condiciones - if statement

https://goplay.tools/snippet/s8oH-bkpJpu

Condiciones - if, else if, else

https://goplay.tools/snippet/jYSExDZShpv

Condiciones - switch statement

- switch on boolean value: https://goplay.tools/snippet/Ht1G6Ajz0wl
- no default fallthrough: https://goplay.tools/snippet/JJHP0xBTQ9E
- funky fallthrough: https://goplay.tools/snippet/FceZ7y4x3Tr
    - “fallthrough” (caer a través), hará que se imprima la siguiente sentencia.
- default https://goplay.tools/snippet/T158zKEEARl
- switch on a value: https://goplay.tools/snippet/ECPfjVaV0LE
- switch on multiple matches for a case: https://goplay.tools/snippet/liFGMVqwtf5

Condiciones - switch statement documentación

https://golang.org/ref/spec#Switch_statements

Operadores lógicos condicionales

https://goplay.tools/snippet/4qVqVHPBANL

https://goplay.tools/snippet/WyvRCuRKzFY

Ejercicios

01

- Usar el bucle for para imprimir el índice desde el 1 hasta el número 100.
- Solución: https://goplay.tools/snippet/wmBBe0I4TYf

02

- Usar un bucle for para mostrar los años desde 1970 hasta hoy
- Solución: https://goplay.tools/snippet/8K-ji-UN51x

03

- Con un COMPOSITE LITERAL:
    - Crear un array que contenga 5 valores de tipo int.
    - Asignar los valores a cada posición.
- Recorrer el array e imprimir los valores.
- Usar format printing:
    - Imprimir el tipo del array.
- Solución: https://goplay.tools/snippet/DQWe-HgT5eP

04

- Con un composite literal:
  - Crear un slice de tipo int
  - Asignar 10 valores
- Recorrer el slice e imprimir los valores
- Con format printing
  - Imprimir el tipo del slice
- Solución: https://goplay.tools/snippet/j-36Yj6VdhB

05

- A partir de este slice:
  - x := []int{42, 43, 44, 45, 46, 47, 48, 49, 50, 51}
- Obtener:
  - [42 43 44 45 46]
  - [47 48 49 50 51]
  - [44 45 46 47 48]
  - [43 44 45 46 47]
- Solución: https://goplay.tools/snippet/_wVTD3ZehlY

06

- En x := []int{42, 43, 44, 45, 46, 47, 48, 49, 50, 51}
- Añadir 52
- Imprimir slice x
- añadir En una línea 53, 54, 55
- Imprimir slice x
- y := []int{56, 57, 58, 59, 60}
- Añadir y a x
- Imprimir slice x
- Solución: https://goplay.tools/snippet/b4sCLHev3tU

07

- Eliminar parte del slice.
- Comenzar con este slice: x := []int{42, 43, 44, 45, 46, 47, 48, 49, 50, 51}
- Usar append y slicing para obtener estos valores: [42 43 44 48 49 50 51]
- Solución: https://play.golang.org/p/M0WDF7jjjSR

08

Crear un slice que contenga todos los días de la semana:

- Slice inicial: diasSemana = []string{`Lunes`, ` Martes`, ` Miercoles`, ` Jueves`, ` Viernes`, ` Sabado`, `Domingo`}
- Como se sabe cuál es la longitud del slice, es más eficiente usar make.
- Imprimir su longitud, capacidad y cada valor y posición.
- Solución: https://goplay.tools/snippet/17RD3b7b04H

09

- Crear un slice multidimensional.
- Almacenar la siguiente info:
  - “Homero”, "Simpson", "Vivo en Springfield”.
  - "Montgomery", "Burns", "Excelente...".
- Recorrer todos los valores y extraer todos sus valores.
- Solución: https://goplay.tools/snippet/hdZw2awmgfl

10

- Crear un map de slices de strings.
- Imprimir los valores con su índice.
- Solución: https://goplay.tools/snippet/jeVBLpZlCwO

11

- Con el código del ejercicio anterior, añadir un nuevo récord al map.
- Imprimir con range loop.
- Solución: https://goplay.tools/snippet/5DYElhCuywQ

12

- Con el código del ejercicio anterior, eliminar un récord del map.
- Imprimir con range loop
- Solución: https://goplay.tools/snippet/Xoups58mY5A

Types 3 (Structs)

Structs

Los structs en Golang son una forma de agrupar valores de diferentes tipos de datos bajo un solo tipo definido por el usuario. Los structs son útiles para representar entidades del mundo real, como un objeto o registro en un programa.

https://goplay.tools/snippet/X2yGrKhBe-1

https://goplay.tools/snippet/BwoeAiqQi0W

En este ejemplo, se define un struct llamado "Persona" con tres campos: Nombre, Apellido y Edad. Luego, en la función main, creamos una instancia de Persona y asignamos valores a los campos. Después, imprimimos los valores de los campos en la consola

https://goplay.tools/snippet/OBNFtPPSKRO

En este ejemplo, hemos definido una función llamada "NombreCompleto()" que toma como receptor una instancia de Persona y devuelve el nombre completo de la persona. Luego, en la función main, creamos una instancia de Persona y llamamos a la función NombreCompleto().

https://goplay.tools/snippet/O7Xnef0K3uz

Embebed structs

Los structs incrustados en Golang son una forma de combinar dos o más structs en uno solo, lo que permite la herencia de propiedades de un struct dentro de otro. Esto se utiliza para lograr algo similar a la herencia en la programación orientada a objetos, aunque en Go no existe la herencia en el sentido tradicional.

https://goplay.tools/snippet/h-5LIZV1Xy1

En este ejemplo, se define un struct llamado "Dirección" con tres campos: Calle, Ciudad y País. Luego, en el struct "Persona", incrustamos el struct "Dirección". Al hacerlo, los campos de "Dirección" se vuelven accesibles directamente a través de una instancia de "Persona".

En la función main, creamos instancias de "Dirección" y "Persona", y asignamos valores a sus campos. Después, imprimimos los valores de los campos en la consola, notarás que los campos del struct incrustado "Dirección" pueden ser accedidos directamente desde la instancia "persona1" sin tener que acceder a "Dirección" primero.

https://goplay.tools/snippet/jDCP-J0Z5Vz

Anonymous structs

Los structs anónimos en Golang son una forma de crear structs sin asignarles un nombre explícitamente. Son útiles cuando necesitas una estructura de datos simple y temporal que no se utilizará en otras partes del código.

https://goplay.tools/snippet/_ACcNnIDF4z

En este ejemplo, en lugar de definir un struct "Persona" con nombre, creamos un struct anónimo con los campos Nombre, Apellido y Edad. Luego, en la función main, creamos una instancia de este struct anónimo y asignamos valores a sus campos. Después, imprimimos los valores de los campos en la consola.

Ten en cuenta que los structs anónimos son útiles cuando necesitas una estructura de datos de una sola vez o en un ámbito limitado. No se pueden reutilizar fácilmente en otras partes del código y no admiten métodos ni pueden incrustarse en otros structs como los structs con nombre.

Si no hay necesidad de ponerle un nombre, se puede invocar directamente y llenarla.

https://goplay.tools/snippet/TqiJZxmADEb

Ejercicios

01

- Crear un struct de tipo persona, debe almacenar:
  - nombre
  - apellido
  - comidas favoritas
- Crear dos personas, imprimir el nombre y las comidas.

Solución: https://goplay.tools/snippet/PZyDmQm5jZd

02

Con el código del ejercicio anterior:

- Almacenar los valores de tipo persona en un map,
- con el key del apellido.
- Acceder a cada valor en el map.
- Imprimir los valores del slice.

Solución: https://goplay.tools/snippet/XgE8aFhHyl0

03

- Crear struct tipo vehículo, con los campos:
  - puertas
  - color
- Crear 2 nuevos tipos:
  - sedan
  - camion
- Embeber vehículo en ambos.
- Para camión agregar el campo "cuatro Ruedas" bool.
- Para sedan agregar el campo "deLujo" bool.
- Con un composite literal, crear el valor de tipo camión y asignar los valores a los campos.
- Con un composite literal, crear el valor de tipo sedan y asignar los valores a los campos.

Solución: https://goplay.tools/snippet/WqGh6bd2rKr

Ejemplo composite literal: https://goplay.tools/snippet/VmVkSWSt5E6

Ejercicios 1

Variables

1. Crea una variable "nombre" de tipo string y asígnale tu nombre. Luego imprime el valor de la variable por la consola.
2. Crea una variable "edad" de tipo int y asígnale tu edad. Luego imprime el valor de la variable por la consola.
3. Crea una variable "pi" de tipo float64 y asígnale el valor de pi (3.14159). Luego imprime el valor de la variable por la consola.

Condicionales

1. Crea una variable "número" de tipo int y asígnale un número cualquiera. Luego crea un condicional que imprima "Es un número positivo" si el número es mayor que cero, "Es un número negativo" si el número es menor que cero, y "El número es cero" si el número es igual a cero.
2. Crea una variable "edad" de tipo int y asígnale tu edad. Luego crea un condicional que imprima "Eres mayor de edad" si la edad es mayor o igual a 18 años, y "Eres menor de edad" si la edad es menor que 18 años.

Estructuras de Repetición

1. Crea un loop que imprima los números del 1 al 10 por la consola.
2. Crea un slice de tipo int con los números del 1 al 5. Luego crea un loop que imprima los elementos del slice por la consola.

Structs

1. Crea una estructura llamada "persona" que contenga los campos "nombre" y "edad". Luego crea una variable de tipo "persona" y asígnale tus datos. Luego imprime los datos de la persona por la consola.
2. Crea una estructura llamada "producto" que contenga los campos "nombre", "precio" y "cantidad". Luego crea una variable de tipo "producto" y asígnale los datos de un producto cualquiera. Luego imprime los datos del producto por la consola

Integradores

1. Crea un programa que pregunte al usuario su edad y luego determine si es mayor de edad o no.
2. Crea un programa que pida al usuario un número y luego determine si es par o impar.
3. Crea un programa que pida al usuario dos números y luego determine cuál es el mayor de los dos.
4. Crea un programa que simule un juego de adivinanza. El programa debe generar un número aleatorio entre 1 y 100, y luego preguntar al usuario que adivine el número. Si el número que ingresó el usuario es mayor que el número aleatorio, el programa debe imprimir "El número es menor". Si el número que ingresó el usuario es menor que el número aleatorio, el programa debe imprimir "El número es mayor". Si el número que ingresó el usuario es igual al número aleatorio, el programa debe imprimir "¡Adivinaste!" y terminar el juego.
5. Crea un programa que le pida al usuario su nombre y luego imprima un saludo personalizado. Si el nombre ingresado es "Juan", el programa debe imprimir "¡Hola, Juan!". Si el nombre ingresado es diferente de "Juan", el programa debe imprimir "¡Hola, [nombre]!".

Types 4 (Funciones)

En programación, una función o método es una sección de código que se puede llamar desde otras partes del programa para realizar una tarea específica. En Go, una función es una secuencia de instrucciones que realizan una tarea específica, y que puede tener parámetros y valores de retorno opcionales.

Una función en Go se define utilizando la palabra clave `func`, seguida del nombre de la función, los parámetros (si los hay) y el cuerpo de la función entre llaves `{}`. Por ejemplo:

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

En este ejemplo, se define la función `sumar()`, que toma dos parámetros de tipo `int` y devuelve un valor de tipo `int`. La función calcula la suma de los dos números y devuelve el resultado.

Por otro lado, un método en Go es una función que está asociada con un tipo específico y que puede ser llamada por una instancia de ese tipo. Un método se define utilizando la palabra clave `func`, seguida del nombre del tipo y el nombre del método, los parámetros (si los hay) y el cuerpo de la función entre llaves `{}`. Por ejemplo:

```
type Persona struct {
nombre string
edad int
}
func(p persona) presentarse() {
fmt.Printf("Hola, mi nombre es %s y tengo %d años.\n", p.nombre p.edad)
}
```

En este ejemplo, se define el tipo `Persona` con dos campos `nombre` y `edad`, y se define el método `presentarse()`. El método toma un parámetro implícito de tipo `Persona` y utiliza los campos `nombre` y `edad` de la instancia para imprimir un mensaje de presentación en la consola.

Sintaxis

```
func nombreFunc (parametros) (return/s) { ... }
func foo(nombre string, apellido string) (string, bool) { ... }
```

https://goplay.tools/snippet/Yo8YAw3M7rm

Funciones con valores de retorno

Las funciones que tienen valores de retorno devuelven un resultado al programa que las llama.

```
func suma(a int, b int) int {
  return a + b
}
```

Funciones sin valores de retorno

Las funciones que no tienen valores de retorno simplemente realizan una tarea específica sin devolver nada al programa que las llama. Por ejemplo:

```
func imprimirMensaje(mensaje string) {
  fmt.Println(mensaje)
}
```

Funciones variádicas

Las funciones variádicas pueden aceptar un número variable de argumentos. Para declarar una función variádica, se utiliza `...` Después del tipo del último parámetro. Por ejemplo:

```
// Función variádica para sumar números.
func suma(numeros ...int) int {
    total := 0
    for _, numero := range numeros {
        total += numero
    }
    return total
}
func main() {
    resultado := suma(1, 2, 3, 4, 5)
    fmt.Println("La suma es:", resultado)
}
```

https://goplay.tools/snippet/51VlmT3wVJQ

https://gobyexample.com/variadic-functions

```
func foo (x ...int) <--- se almacenan como []int (slice de ints)
```

Usando un slice

https://goplay.tools/snippet/a9OD881vtFi

```
unSliceDeInts := []int {1,2,3,4,5}
sum(unSliceDeInts)
// sum espera ints, NO un slice de ints
// entonces esto no es válido
func sum (x ...int) int {...}
// es necesario transformar el slice en lo que la función necesita recibir
x := sum(xi...)
// los valores son enviados como ints individuales, que es lo q necesita sum
sum(x)
```

Los variadic parameters NO marcan ERROR si no se les pasa nada.

O sea, variadic, significa 0 o más:

```
func sum (x ...int)
```

```
CORRECTO: sum ()
func(s string, x ...int)
CORRECTO: sum("hola")
```

https://golang.org/ref/spec#Passing_arguments_to_..._parameters

Funciones anónimas

Las funciones anónimas son funciones sin nombre que se definen dentro de otras funciones. Pueden ser útiles para realizar tareas específicas dentro de una función. Por ejemplo:

```
func saludarEnDiferentesIdiomas(nombre string) {
    // Función anónima para saludar en español
    español := func() {
        fmt.Printf("Hola, %s!\n", nombre)
    }
    // Función anónima para saludar en inglés
    inglés := func() {
        fmt.Printf("Hello, %s!\n", nombre)
    }
    // Función anónima para saludar en francés
    francés := func() {
        fmt.Printf("Bonjour, %s!\n", nombre)
    }
    español()
    ingles()
    frances()
}
```

Funciones de orden superior

Las funciones de orden superior son funciones que aceptan una o más funciones como argumentos y/o devuelven una función como resultado. Estas funciones pueden ser muy útiles para abstraer y reutilizar lógica.

```
package main
import (
    "fmt"
)
// Tipo para una función que no toma argumentos y no devuelve nada.
type simpleFunc func()
// funcionEjecutora es una función de orden superior que toma una función simpleFunc y la ejecuta.
func funcionEjecutora(f simpleFunc) {
    fmt.Println("Antes de ejecutar la función.")
    f() // Ejecuta la función pasada como argumento.
    fmt.Println("Después de ejecutar la función.")
}
// Ejemplo de función que podemos pasar a funcionEjecutora.
func saludar() {
    fmt.Println("¡Hola, mundo!")
}
func main() {
    funcionEjecutora(saludar)
```

```
}
```

Las function types y los function values pueden ser usados y pasado como cualquier otro valor.

Todo en Go se pasa como VALOR.

https://goplay.tools/snippet/ISLa6ngTi46

Funciones sin cuerpo (no Body)

Go permite declarar funciones sin un cuerpo para algunos casos específicos, como cuando se utiliza la declaración `func init()` para inicializar el programa o para declarar una función que será implementada más adelante en el código

Funciones `init()`: En Go, la función `init()` es una función especial que se utiliza para realizar la inicialización de un paquete antes de que se ejecute el programa. La función `init()` no tiene parámetros ni valor de retorno, y no se llama directamente en el programa. En su lugar, se ejecuta automáticamente cuando se importa el paquete.

```
package main
import (
“fmt”
)
func init() {
  fmt.Println(”Inicializando programa...”)
}
func main() {
  fmt.Println(”Ejecutando programa...”)
}
```

En este ejemplo, la función `init()` imprime un mensaje en la consola cuando se importa el paquete, mientras que la función `main()` imprime un mensaje diferente cuando se ejecuta el programa.

Defer

https://golang.org/ref/spec#Defer_statements

https://goplay.tools/snippet/gvtVSxglaeH

Difiere la función al final.

Anonymous func

https://goplay.tools/snippet/XlJgjwM1aL9

- Una función anónima NO TIENE nombre.

func expression

https://goplay.tools/snippet/DgfJZaVqfQ4

https://golang.org/ref/spec#Function_types

Returning a func

https://goplay.tools/snippet/GiktfDrn1nF

Callback

https://goplay.tools/snippet/oTKea6DUsT7

- En Go las funciones son “ciudadanos de primera clase”, se las puede, pasar, retornar, asignar a variables.
- Un callback es pasar una función (func) como un argumento.
- Tratar de evitar su uso, es confuso.

Closure

https://goplay.tools/snippet/KXYELup38SB

- Es la propiedad de acotar el ámbito de una variable.
- Una función closure, también conocida simplemente como "closure", es una función que puede capturar variables del entorno en el cual se declara. Los closures son útiles porque pueden acceder y modificar variables que están fuera de su propio ámbito, permitiendo mantener un estado entre llamadas.
- Las funciones closures son comunes en Go, especialmente cuando se trabaja con funciones de orden superior, donde una función devuelve otra función o acepta una función como argumento.

Recursion

https://goplay.tools/snippet/11t6cy6wg0q

- Todo lo que se hace recursión puede hacerse con loops.
- Son confusas y consumen más memoria.
- Es cuando una función se llama a sí misma n veces.
- Los bucles son mejores (casi siempre).
- Tratar de evitar.

¿Es GO OO?

https://blog.friendsofgo.tech/posts/es_go_un_lenguaje_orientado_a_objetos/

Sí y no. A pesar de que Go tiene tipos y métodos y permite la programación orientada a objetos, no hay jerarquía de tipos. El concepto de “interfaz” en Go tiene una aproximación que creemos que es mucho más sencilla de usar y en algunos casos más general. También hay maneras de embeber tipos en otros tipos proporcionando algo similar – pero no idéntico – a la subclase. Además, los métodos en Go son más generales que en C++ o Java: se pueden definir para cualquier tipo de datos, incluso tipos integrados como enteros simples. No están restringidos a estructuras (clases).

Se puede decidir enfocar y utilizar Go como lenguaje OO, pero se tiene que tener en cuenta que será al estilo Go.

Diferencias

Adiós clases, hola structs

Al principio los structs pueden parecer clases, pero no tienen el mismo comportamiento y no son lo mismo. Podemos asignar a los structs métodos, dándole el comportamiento de una clase tradicional, donde la estructura solo contiene el estado y no el comportamiento, los métodos le proporcionarán el comportamiento al permitirle cambiar el estado.

Encapsulamiento

El encapsulamiento en Go funciona a nivel de paquetes y además se realiza de manera implícita dependiendo de la primera letra del método o atributo. Así que si declaramos un método o atributo con la primera letra en mayúscula, quiere decir que está exportado, es decir, es público fuera del paquete. Go no implementa protected, ya que no hay herencia, aunque tiene algo que se le podría asemejar que son los internas.

No existe herencia

Métodos

Los métodos son funciones que están asociadas a un tipo específico y pueden ser llamados por una instancia de ese tipo.

```
// Persona es un tipo struct que representa a una persona.
type Persona struct {
    Nombre string
}

// Saludar es un método que pertenece al tipo Persona.
func (p Persona) Saludar() {
    fmt.Printf("Hola, mi nombre es %s\n", p.Nombre)
}
```

https://goplay.tools/snippet/7ZzVP7Zsnk2

```
func (receiver) nombreFunc (parametros) (return/s) { ... }
```

Cuando tenés un receiver, este adjuntará la función al tipo definido en el receiver.

Ejercicios

Ejercicio 1

- Crear func foo que retorne un int.
- Crear func bar que retorne un int y un string.
- Llamar a ambas funciones.
- Imprimir resultados.
- **Solución**: https://goplay.tools/snippet/vStiq1dPxSP

Ejercicio 2

- Crear func foo():
  - Con un variadic parameter int.
  - Se pasa un []int func (unfurl el []int).
  - retornar sum de los ints.
- Crear func bar():
  - parámetro []int.
  - retornar sum todos los int.
- **Solución:** https://goplay.tools/snippet/obtQgQFhMyF

Ejercicio 3

- Usa la palabra clave “defer” para mostrar que una “deferred func” corre después de la que la función que la continúe termina.

- **Solución 1**: https://goplay.tools/snippet/0Ti7ocWISmG
- **Solución 2**: https://goplay.tools/snippet/RvobBJF_EwS

Ejercicio 4

- Crear struct persona:
  - nombre.
  - apellido.
  - edad.
- Agregar el método hablar() a persona:
  - hablar(): debe decir el nombre y la edad de la persona.
- Crear un valor de tipo persona.
- llamar al método hablar() del valor de tipo persona**.**
- **Solución:** https://goplay.tools/snippet/xYaZZinKeSB

Ejercicio 5

- Crear una función anónima.
- **Solución:** https://goplay.tools/snippet/sWmUOq2D0rL

Ejercicio 6

- Asignar como variable una función, y usarla el valor en otra.
- **Solución:** https://goplay.tools/snippet/k6GS1MDth17

Ejercicio 7

- Crear una func que retorne otra func
- **Solución:** https://goplay.tools/snippet/shhOWfYQlL7

Ejercicio 8

- Una función “callback” es una función que cuando pasa una func dentro de otra func como un argumento.
- Pasar una func dentro de otra func como argumento
- **Solución:** https://goplay.tools/snippet/uaPb-l4qQ2D

Ejercicio 9

- Un closure en programación es una función que retiene el acceso a las variables de su entorno léxico, incluso después de que la ejecución ha salido del bloque de código donde fueron declaradas. Esto permite que la función "recuerde" y acceda a esas variables en llamadas posteriores.
- Crear un Closure en Go
  - Crear una función que genere un closure en Go. Este closure deberá "capturar" una variable local y permitir su manipulación en llamadas sucesivas.
- **Solución:** https://goplay.tools/snippet/PXNntScP0GF

Punteros

¿Que son los punteros?

Un puntero es un apuntador a alguna parte de la memoria donde se almacena un valor.

Los punteros en Go son variables que contienen la dirección de memoria de otra variable en lugar del valor de la variable en sí. Esto permite realizar operaciones en la ubicación de memoria, lo que puede ser útil para optimizar el rendimiento, modificar valores fuera del alcance de una función o trabajar con estructuras de datos complejas.

https://goplay.tools/snippet/aZC7hxebRoL

https://golang.org/ref/spec#Operators_and_punctuation

```
a := 43
// a es de tipo int
//&a es de tipo puntero *int
```

Ambas son de TIPOS ABSOLUTAMENTE DIFERENTES, como int lo es con un string o con cq otro tipo.

```
var b int = &a // esto da ERROR, ya que son de tipos diferentes
var b *int = &a // esto es correcto, b apuntará a la misma dirección que &a
```

TODO en Go es pasado por valor, aquí se pasa la dirección como otro valor a otro puntero.

Declaración de punteros

Para declarar un puntero, utilizamos el operador `*` seguido del tipo de variable al que apunta. Por ejemplo, un puntero a una variable `int` se declararía como `*int`.

```
var punteroInt *int
```

```
var a int // tipo entero (int)
&a // tipo pointer (*int)
*&a // tipo entero (int)
```

```
var b *int // tipo pointer (*int)
b // tipo pointer (*int)
*b // tipo entero (int)
```

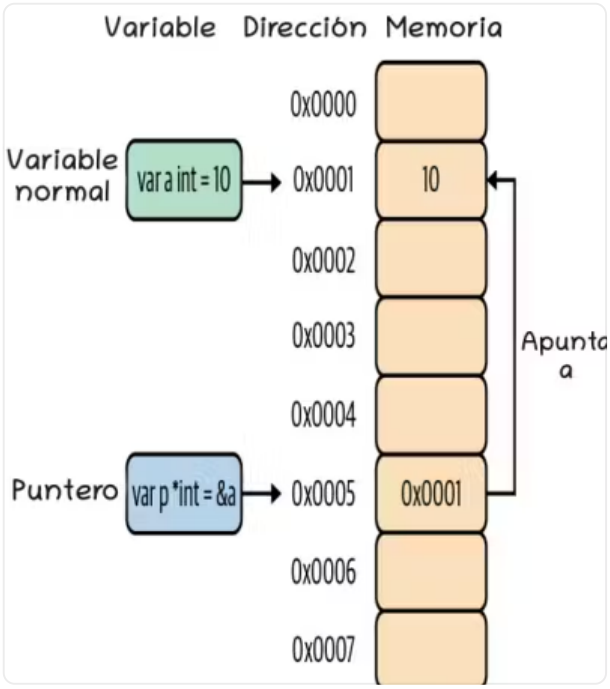
Obtener dirección de memoria

Para obtener la dirección de memoria de una variable, utilizamos el operador `&` antes del nombre de la variable.

```
numero :=10
punteroInt = &numero
```

& devuelve la dirección donde está almacenado el valor.

* devuelve el valor almacenado.



Aquí se puede ver claramente cómo el puntero apunta a dirección de memoria donde está el valor de la variable de la cual es puntero.

Acceder al valor apuntado

Para acceder al valor almacenado en la dirección de memoria apuntada por un puntero, utilizamos el operador `*` antes del nombre del puntero.

```
valor := *punteroInt
fmt.Println(valor) // imprime 10
```

Modificar el valor apuntado

También podemos modificar el valor almacenado en la dirección de memoria apuntada por un puntero utilizando el operador `*`.

```
*punteroInt =20
fmt.Println(numero) // imprime 20
```

https://goplay.tools/snippet/sjpTGDmX6yP

Al ejecutar este programa, obtendrás la siguiente salida:

```
Dirección de memoria de numero: 0xc00001e0e0
Valor apuntado por puntero: 42
Nuevo valor de numero: 99
```

La salida mostrará que la dirección de memoria de la variable **`numero`** es la misma que la almacenada en el puntero. Luego, accedemos al valor almacenado en esa dirección de memoria utilizando el puntero y modificamos el valor de la variable **`numero`** a través del puntero.

Es importante tener en cuenta que el uso excesivo de punteros puede hacer que el código sea más difícil de leer y mantener, por lo que es recomendable utilizar punteros solo cuando sea necesario.

En Go, los punteros son especialmente útiles al trabajar con estructuras y funciones que modifican el estado de las estructuras. Al pasar un puntero a una estructura a una función, se evita la copia de la estructura completa y se permite que la función modifique la estructura original en lugar de trabajar con una copia.

Ejemplo con slices

En Go, los slices son referencias a un segmento de un array subyacente. Por lo tanto, cuando pasas un slice a una función, en realidad estás pasando una referencia al array subyacente y no es necesario utilizar punteros explícitamente.

https://goplay.tools/snippet/an4hzMFnfcu

Como puedes ver, no necesitamos utilizar punteros explícitamente al trabajar con slices, ya que la función `modificarSlice` modifica el slice original directamente.

Un subslice es una referencia a un segmento del mismo array subyacente que el slice original. Por lo tanto, cualquier modificación en el subslice se reflejará en el slice original.

https://goplay.tools/snippet/VaOPFv6oUEk

*[]string, []string y []*string

Las diferencias entre `*[]string`, `[]string` y `[]*string` en Go (Golang) son conceptuales y prácticas, y se relacionan con cómo se manejan los datos en la memoria. Veamos cada uno de ellos:

1. `*[]string`: Es un puntero a un slice de strings. Este enfoque permite modificar el slice original en la memoria cuando se pasa a una función o método, ya que se está pasando la dirección de memoria del slice, en lugar del slice en sí. Esto es útil para evitar la copia del slice al pasarlo como argumento. Para acceder a los elementos del slice, se utiliza la notación `*`, que dereferencia el puntero.

```
slicePtr := &[]string{"Hola", "Mundo"}
fmt.Println(*slicePtr) // Imprime: [Hola Mundo]
```

1. `[]string`: Representa un slice de strings. Un slice es una estructura dinámica que puede cambiar su tamaño y es una referencia a un segmento de un array. Los slices son eficientes en términos de memoria, ya que solo almacenan la dirección de inicio, la longitud y la capacidad, sin importar el tamaño del array al que apuntan. Los elementos se acceden mediante índices, como `slice[índice]`.

```
slice := []string{"Hola", "Mundo"}
fmt.Println(slice) // Imprime: [Hola Mundo]
```

1. `[]*string`: Es un slice de punteros a strings. Cada elemento es un puntero a un string, lo que permite representar la ausencia de un string mediante el valor `nil`. Este enfoque es útil cuando los strings pueden ser opcionales o cuando se desea modificar los strings originales a los que apuntan los punteros.

```
ptrSlice := []*string{nil, nil}
strPtr := "Hola"
ptrSlice[0] = &strPtr
fmt.Println(ptrSlice) // Imprime: [0x<dirección_de_memoria> nil]
```

En el ejemplo, se ilustran estas diferencias creando y accediendo a un puntero a un slice de strings, un slice de strings y un slice de punteros a strings. Cada uno tiene usos específicos dependiendo de las necesidades de manejo de memoria y de las operaciones que se deseen realizar con los datos.

Ejemplo con structs

Cuando trabajamos con structs, es común utilizar punteros para evitar copiar toda la estructura y permitir que las funciones modifiquen el estado de la estructura original. Veamos un ejemplo con un struct `persona`.

https://goplay.tools/snippet/Xsw-3CYV0rk

¿Cuándo usar punteros?

- Cuando se maneja una gran porción de información, es más eficiente pasar la dirección, o sea el puntero.
- También cuando se necesita cambiar algo que está almacenado en una posición de memoria determinada.

https://goplay.tools/snippet/RCZyNPwitkT

Trabajando con Punteros

Cuando trabajan con punteros tienen que validar los nil

La elección entre:

- func xxxx(a *miTipo)
- func xxxx(a miTipo)

en Go depende en gran medida del contexto y de lo que estés tratando de lograr.

func foo(a *miTipo) utiliza un puntero a miTipo. Esto significa que la función puede modificar el estado de a. Esta es una opción común cuando tienes una estructura grande y quieres evitar el coste de la copia, o si quieres que la función sea capaz de modificar el estado del argumento.

func bar(a miTipo) por otro lado, pasa a por valor. Esto significa que la función recibe una copia del argumento y no puede modificar el estado original de a. Esta es una opción común cuando estás trabajando con tipos pequeños, como los tipos primitivos, donde el costo de copia es pequeño, o cuando no quieres que la función sea capaz de modificar el argumento.

En resumen, si necesitas modificar el estado de a dentro de la función, o si a es una estructura grande y quieres evitar el costo de la copia, entonces **func foo(a *miTipo)** es una opción más adecuada. Si estás trabajando con tipos más pequeños, o no quieres que la función pueda modificar el argumento, entonces **func bar(a miTipo)** puede ser más apropiado.

Errores en tiempo de ejecución como los errores de puntero nulo (nil pointer dereference error)

El manejo correcto de punteros en Go es crucial para evitar errores de tiempo de ejecución, como los errores de puntero nulo (nil pointer dereference). Cuando se intenta acceder a un campo o método a través de un puntero nulo en Go, ocurre un error y la aplicación se bloquea.

Para prevenir estos errores, se recomienda verificar siempre que los punteros no sean nulos antes de usarlos. Por ejemplo:

```
func miFuncion(a *MiTipo) {
    if a == nil {
        // Manejar el caso de puntero nulo aquí...
        return
    }
    // Continuar con la lógica de la función aquí...
}
```

Es importante notar que no es necesario verificar cada puntero en cada función. Esta necesidad depende de la lógica del programa y del uso de los punteros. En algunos casos, puedes estar seguro de que un puntero nunca será nulo gracias a la

lógica del programa. Sin embargo, en otros casos, la verificación es esencial.

Además, al trabajar con punteros en Go, no solo debes preocuparte por los punteros nulos. También es crucial considerar aspectos como la concurrencia y la sincronización de datos, especialmente cuando se utilizan punteros en múltiples goroutines.

El error de “nil pointer dereference” sucede al intentar acceder a un valor a través de un puntero nulo. Por ejemplo, si tienes un puntero `p` a un `int`, puedes acceder al valor con `*p`. Pero si `p` es `nil`, entonces `*p` causará un error.

```
var p *int
fmt.Println(*p)
// panic: runtime error: invalid memory address or nil pointer dereference
```

En este caso, `p` es un puntero a un `int` que no ha sido inicializado y, por defecto, es `nil`. Al intentar imprimir `*p`, se trata de dereferenciar `p`, lo que causa un error.

Para evitar errores de puntero nulo, asegúrate siempre de que un puntero no sea `nil` antes de intentar dereferenciarlo.

Este tipo de error es común en muchos lenguajes de programación que usan punteros o referencias. En Go, `nil` es el valor cero para punteros, interfaces, mapas, slices, canales y funciones. Si intentas acceder a un valor o método a través de uno de estos tipos cuando es `nil`, resultará en un error de puntero nulo.

Por ejemplo, en el siguiente código, se intenta imprimir el campo `Valor` de un puntero a `MiStruct` que es `nil`, lo que causa un error:

```
type MiStruct struct {
    Valor int
}

func main() {
    var ms *MiStruct
    fmt.Println(ms.Valor) // Error de nil pointer dereference
}
```

Para evitar este error, debes inicializar correctamente los punteros antes de usarlos. En el ejemplo anterior, podrías inicializar el puntero así:

```
type MiStruct struct {
    Valor int
}

func main() {
    ms := &MiStruct{} // ms es un puntero a una nueva estructura MiStruct
    fmt.Println(ms.Valor)
}
```

Aquí, `&MiStruct{}` crea e inicializa una estructura `MiStruct`, y `ms` apunta a ella, lo que evita el error de puntero nulo.

Type Embedding

En Go, Type Embedding es un mecanismo que permite que los tipos dentro de una estructura se incorporen en otra estructura para reutilizar su comportamiento y datos. En términos más simples, la Type Embedding es una forma de definir estructuras compuestas que contienen campos incrustados o anidados de otros tipos.

Por ejemplo, suponga que tiene dos estructuras "A" y "B", donde "A" tiene algunos campos, y "B" necesita esos campos. Para hacerlo, puede usar la Type Embedding escribiendo type B struct { A } donde "A" esta incrustado en "B". Con esto, "B"

ahora contiene todos los campos de "A", y los campos de "A" pueden ser accedidos como si fueran directamente pertenecientes a "B".

El Type Embedding es particularmente útil en combinación con interfaces y polimorfismo. Al colocar un tipo que implementa una interfaz como campo incrustado dentro de otra estructura, puede extender el conjunto de métodos disponibles en la estructura original y, por lo tanto, aumentar su capacidad en tiempo de ejecución sin tener que conocer exactamente qué tipo está incrustado dentro de ella.

Ejercicios

1#

Crear una variable, imprimir su valor y su posición de memoria.

https://goplay.tools/snippet/GKo12qnqgQI

2#

- Crear struct persona
- Crear func reto
  - persona es un parámetro
  - cambiar el valor de *persona
- Crear un valor persona
  - imprimir el valor
- En func main
  - llamar a reto

https://goplay.tools/snippet/p-NhCoRwkmj

Interfaces

Las interfaces en Golang son un tipo de dato que define un conjunto de métodos. Son muy útiles para abstraer la funcionalidad común entre diferentes tipos de estructuras de datos y para diseñar programas modulares y fácilmente extensibles. En Go, una estructura de datos no necesita declarar explícitamente que implementa una interfaz; si un tipo define todos los métodos especificados en una interfaz, entonces ese tipo implementa automáticamente la interfaz.

Definición de la interfaz:

Primero, definimos una interfaz llamada `Hablante` que especifica un único método llamado `Hablar()`:

type Hablante interface {

```
type Hablante interface {
    Hablar() string
}
```

Implementación de la interfaz:

A continuación, definimos dos structs, `Persona` y `Gato`, que implementan la interfaz `Hablante`. No necesitamos declarar explícitamente que estos structs implementan la interfaz; simplemente definimos el método `Hablar()` en ambos structs:

```
type Persona struct {
    Nombre string
}

func (p Persona) Hablar() string {
    return "Hola, mi nombre es " + p.Nombre
}

type Gato struct {
    Nombre string
}

func (g Gato) Hablar() string {
    return "Miau, soy " + g.Nombre
}
```

Utilizando la interfaz:

Ahora, puedes utilizar la interfaz `Hablante` para escribir funciones que acepten cualquier tipo que implemente la interfaz. Por ejemplo, aquí hay una función llamada `Saludar` que acepta un parámetro de tipo `Hablante`:

```
func Saludar(h Hablante) {
    fmt.Println(h.Hablar())
}
```

https://goplay.tools/snippet/x8huGwI2qP1

Implementación de la interfaz con un receptor de puntero:

Primero, definimos una interfaz llamada `Incrementador` que especifica un único método llamado `Incrementar()`:

```
type Incrementador interface {
    Incrementar()
}
```

A continuación, definimos un struct llamado `Contador` que implementa la interfaz `Incrementador` con un receptor de puntero en lugar de un receptor de valor. Esto permite que el método `Incrementar()` modifique el estado del struct `Contador`:

```go
type Contador struct {
    Valor int
}

func (c *Contador) Incrementar() {
    c.Valor++
}
```

Utilizando la interfaz

Ahora, puedes utilizar la interfaz `Incrementador` para escribir funciones que acepten cualquier tipo que implemente la interfaz. Por ejemplo, aquí hay una función llamada `IncrementarDosVeces` que acepta un parámetro de tipo `Incrementador`:

```go
func IncrementarDosVeces(i Incrementador) {
    i.Incrementar()
    i.Incrementar()
}
```

Ejemplo completo: https://goplay.tools/snippet/fl07_QwBCIc

Method sets

https://goplay.tools/snippet/SG1KrlPL5m4

https://golang.org/ref/spec#Method_sets

- Un reciber no NO-PUNTERO (t T):
  - Puede trabajar con valores que son no-punteros y punteros
- Un reciber PUNTERO (t *T)
  - Solo puede trabajar con valores que son punteros.

Error Handling

En Go, la interfaz `error` es una interfaz predefinida utilizada para representar errores. Es una interfaz simple que contiene un único método llamado `Error()`, que devuelve una cadena de texto representando el error:

```go
type error interface {
    Error() string
}
```

La mayoría de las bibliotecas y funciones estándar de Go retornan un valor de error como su último valor de retorno. Si un error ocurre, el valor de error será diferente de `nil`; de lo contrario, será `nil`.

Para crear un error personalizado, puedes implementar la interfaz `error` en tu propio tipo. Aquí tienes un ejemplo de cómo crear un error personalizado y manejar errores en Go:

Crear un error personalizado:

Definimos un error personalizado llamado `MiError` que contiene información adicional además del mensaje de error:

```go
type MiError struct {
    Mensaje string
    Codigo  int
}

func (e MiError) Error() string {
    return fmt.Sprintf("MiError: %s (código: %d)", e.Mensaje, e.Codigo)
}
```

Función que devuelve un error:

A continuación, vamos a escribir una función llamada `FuncionConError` que puede devolver un error personalizado `MiError`:

```go
func FuncionConError(condicion bool) (string, error) {
    if condicion {
        return "", MiError{Mensaje: "Algo salió mal", Codigo: 1}
    }
    return "Todo salió bien", nil
}
```

- https://goplay.tools/snippet/hGQNhN9NDcJ

En este ejemplo, primero llamamos a `FuncionConError` con un valor `true`, lo que provoca que la función retorne un error personalizado de tipo `MiError`. En el siguiente llamado a `FuncionConError`, pasamos un valor `false`, lo que hace que la función retorne un resultado exitoso y un error `nil`. En ambos casos, verificamos si el valor de `err` es diferente de `nil` para determinar si hubo un error y, de

Receiver de puntero o no puntero (de valor)

En Go, para definir un método hay que utilizar receivers, se pueden definir como de valor (no puntero) o de puntero.

Receptor de puntero

Modificar el receptor:

Si el método necesita modificar el estado del receptor, entonces necesitas usar un receptor de puntero. Las modificaciones que se hagan en un receptor de valor no persistirán después de que el método termine, ya que el receptor de valor es una copia del valor original.

Grandes estructuras de datos: Si estás trabajando con una estructura de datos grande, usar un receptor de puntero puede ser más eficiente en términos de rendimiento y uso de memoria, ya que evita la necesidad de copiar la estructura cada vez que se llama al método.

Consistencia: Si algunos métodos de tu tipo usan receivers de puntero, entonces es buena idea hacer que todos los métodos de ese tipo usen receptores de puntero para mantener la consistencia.

Receptor de valor:

Inmutabilidad: Si no quieres que un método pueda modificar el estado del valor original, puedes usar un receptor de valor. Esto puede ser útil si queres garantizar que el método no tenga efectos secundarios.

Tipos simples y pequeños: Para tipos simples y pequeños (como int, float, string, etc.), usar un receptor de valor puede ser más eficiente y más sencillo.

Valor semántico: Si el valor del tipo representa un valor semántico que no debe cambiar (como un número complejo, un punto en un plano, etc.), se suele usar un receptor de valor.

En resumen:
Si necesitas modificar el receiver o estás trabajando con estructuras de datos grandes, un receptor de puntero es probablemente la mejor opción.
Si estás trabajando con tipos simples y pequeños, o queres garantizar la inmutabilidad del valor, un receptor de valor puede ser la mejor opción.

Readers & Writers

En Go, los "readers" y "writers" son interfaces utilizadas para la lectura y escritura de datos en diferentes tipos de fuentes.

La interfaz io.Reader es utilizada para leer datos de una fuente de entrada. Esta interfaz define un método llamado Read() que lee datos de la fuente y los guarda en un búfer. El método devuelve el número de bytes leídos y un error si hubo algún problema durante la lectura. Algunos ejemplos de fuentes de entrada pueden ser un archivo, una conexión de red o una cadena de caracteres.

Por otro lado, la interfaz io.Writer es utilizada para escribir datos a una fuente de salida. Esta interfaz define un método llamado Write() que escribe datos en la fuente de salida. El método devuelve el número de bytes escritos y un error si hubo algún problema durante la escritura. Algunos ejemplos de fuentes de salida pueden ser un archivo, una conexión de red o la salida estándar.

Este patrón "readers-writers" es comúnmente utilizado en Go para operaciones de entrada/salida, ya que permite una abstracción genérica y flexible de diferentes tipos de fuentes de datos.

Ejercicios

Ejercicio 1: Formas geométricas

Crea una interfaz llamada **`Forma`** que tenga dos métodos: **`Area()`** y **`Perimetro()`**. Luego, crea structs que representen diferentes formas geométricas, como **`Cuadrado`**, **`Rectangulo`** y **`Circulo`**. Implementa la interfaz **`Forma`** para cada uno de estos structs.

Ejercicio 2: Ordenar

Crea una interfaz llamada **`Ordenable`** con un único método: **`MenorQue(otro Ordenable) bool`**. Luego, crea structs que representen diferentes tipos de objetos que puedan ser comparados, como **`Persona`** (comparando edades) y **`Producto`** (comparando precios). Implementa la interfaz **`Ordenable`** para cada uno de estos structs. Finalmente, escribe una función genérica **`Ordenar`** que acepte un slice de elementos **`Ordenable`** y los ordene utilizando el método **`MenorQue`**.

Ejercicio 3: Dispositivos electrónicos

1. **Definición de la Interfaz Dispositivo:**
   - Crear una interfaz llamada `Dispositivo` que actúa como un contrato para los structs que representan diferentes dispositivos.
   - Esta interfaz incluye tres métodos: `Encender()`, `Apagar()`, y `Estado() string`.
   - Los métodos `Encender()` y `Apagar()` no retornan valores, mientras que `Estado()` devuelve un string que indica el estado del dispositivo.
2. **Creación de Structs para Diversos Dispositivos:**
   - Definir structs para representar distintos dispositivos como `Televisor`, `Computadora`, y `Smartphone`.
   - Cada struct tiene atributos únicos. Por ejemplo, `Televisor` puede tener atributos como `marca`, `pulgadas`, `esSmart`, mientras que `Computadora` puede incluir `marca`, `modelo`, `sistemaOperativo`, etc.
3. **Implementación de la Interfaz en los Structs:**
   - Implementar los métodos de la interfaz `Dispositivo` en cada uno de los structs creados.
   - Esto significa que cada tipo de dispositivo cuenta con su propia versión de los métodos `Encender()`, `Apagar()` y `Estado()`.
   - En `Televisor`, por ejemplo, el método `Estado()` puede retornar “Encendido” o “Apagado” según su estado interno.
4. **Ejemplo de Implementación en Televisor:**
   - En el struct `Televisor`, el método `Encender()` cambiaría una variable interna `estaEncendido` a `true`, y `Apagar()` la cambiaría a `false`.
   - El método `Estado()` devolvería “Encendido” si `estaEncendido` es `true`, y “Apagado” si es `false`.
5. **Uso Práctico de la Interfaz:**

- Una vez implementada la interfaz en todos los structs, se puede usar la interfaz `Dispositivo` para referenciar a cualquier tipo de dispositivo.

Ejercicio 4: Lectores y escritores

Crea dos interfaces: **`Lector`** y **`Escritor`**. La interfaz **`Lector`** debe tener un método **`Leer() string`** y la interfaz **`Escritor`** debe tener un método **`Escribir(string)`**. A continuación, crea structs que representen diferentes tipos de lectores y escritores, como **`Archivo`**, **`Buffer`** y **`BaseDeDatos`**. Implementa las interfaces **`Lector`** y **`Escritor`** según corresponda para cada uno de estos structs.

Ejercicio 5: Animales

Crea una interfaz llamada **`Animal`** que tenga métodos como **`Hablar() string`**, **`Comer() string`** y **`Nombre() string`**. Luego, crea structs que representen diferentes animales, como **`Perro`**, **`Gato`** y **`Pajaro`**. Implementa la interfaz **`Animal`** para cada uno de estos structs.

Ejercicio 5

- Crear el tipo cuadrado.
- Crear el tipo círculo.
- Agregar un método para calcular el área y retornarla.
- Crear el tipo forma que define una interface para cualquier que use el método area().
- Crear func info() toma forma e imprime el área.
- Crear valor cuadrado.
- Crear valor círculo.
- Usar func info para imprimir el área del cuadrado.
- Usar func info para imprimir el área del círculo.
- **Solución:** https://goplay.tools/snippet/MrPyecw23IL

Errores

En Go, los errores se manejan a través de los valores de tipo `error`, que son simplemente interfaces que tienen un método `Error()` que devuelve una cadena de texto que describe el error.

Un valor de tipo `error` es `nil` si no hay error y no es `nil` si hay un error. Para manejar errores en Go, se utiliza el enfoque de "verificar errores" (check errors), lo que significa que después de realizar una operación que puede producir un error, se verifica si hay un error antes de continuar.

```
func divide(a, b float64) (float64, error) {
    if b == 0 {
        return 0, errors.New("cannot divide by zero")
    }
    return a / b, nil
}
.............
result, err := divide(4, 0)
if err != nil {
    // Manejar el error
    log.Fatal(err)
}
// Continuar con el resultado
fmt.Println(result)
```

En el ejemplo anterior, la función `divide` devuelve un error si el segundo parámetro es cero. Después de llamar a la función, se verifica si hay un error antes de continuar. Si hay un error, se maneja el error y se sale del programa.

Es importante tener en cuenta que Go no tiene excepciones como en otros lenguajes de programación, y los errores se manejan explícitamente a través de los valores de tipo `error`.

Ejemplo: https://goplay.tools/snippet/rZ7qSP4aDEh

Nota: La construcción `if err ! = nil` que se muestra en el último ejemplo es el caballo de batalla de la manipulación de errores en el lenguaje de programación Go. Donde sea que una función pueda producir un error, es importante utilizar una instrucción `if` para determinar su presencia

Esto se debe a que *error*, "el tipo" que devolvemos en nuestra función de arriba, se trata de una interfaz, que contiene un método *Error()*, el cual devuelve un *String*.

```
type error interface {
    Error() string
}
```

Creación de errores

La biblioteca estándar ofrece dos funciones incorporadas para crear errores: `errors.New` y `fmt.Errorf`..

`errors.New` tiene un solo argumento: un mensaje de error con una cadena que puede personalizar para avisarles a sus usuarios cuál fue el problema.

Ejemplo: https://goplay.tools/snippet/FKPB0XF9ttD

La función `fmt.Errorf` le permite crear un mensaje de error de forma dinámica. Su primer argumento es una cadena que contiene su mensaje de error con valores de marcadores de posición, como `%s` para cadenas y `%d` para enteros. `fmt.Errorf` interpola los argumentos que siguen esta cadena de formato en esos marcadores de posición en orden:

Ejemplo: https://goplay.tools/snippet/KXUTpwrGL2X

Devolución de errores junto con valores

Es frecuente escribir funciones que devuelven un valor si se completaron con éxito junto con un posible error si la función falló. Go permite que las funciones devuelvan más de un resultado

Ejemplo: https://goplay.tools/snippet/ie6v5tSaBEa

Errores personalizados

También en Go podemos controlar los errores de una forma más personalizada. Para ello podemos crearnos nuestro propio struct que implemente la interface error, y así devolver nuestro error personalizado:

```
type MyError struct{}

func (m *MyError) Error() string {
 return "boom"
}

func sayHello() (string, error) {
 return "", &MyError{}
}

func main() {
 s, err := sayHello()
 if err != nil {
  fmt.Println("unexpected error:", err)
  os.Exit(1)
 }

 fmt.Println("The string:", s)
}
```

Ejemplo: https://goplay.tools/snippet/hltXqj5jlwG

Paquete errors

En el paquete `errors` podemos encontrar funciones que nos permiten interactuar con los errores. La comunidad de Go piensa que lo mejor es usar errores personalizados, es por ello que estas nuevas funcionalidades van enfocadas en dar más herramientas para usarlos.

```
// El método Is informa cuando cualquier error de la cadena de errores es igual al target.
// Un error coincide con el el target, si es igual al target
func Is(err, target error) bool
func As(err error, target any) bool
```

Función Is()

Usando la función Is(), podemos detectar incluso si está envuelto, ya que esta función verifica si algún error en la cadena de errores envueltos coincide con el objetivo. Por lo tanto, esta forma debería ser preferible a la comparación `if err == TargetError`.

Ejemplo: https://goplay.tools/snippet/w2_C3I3-oq5

Función As()

Similar a Is(), As() comprueba si algún error en la cadena de errores envueltos coincide con el objetivo. La diferencia es que esta función verifica si el error tiene un tipo específico, a diferencia de , Is() que examina si es un objeto de error en particular. Debido a que As considera toda la cadena de errores, debería ser preferible al tipo afirmación `if e, ok := err.(*BadInputError); ok`.

Ejemplo: https://goplay.tools/snippet/AGJ2eW5YBKn

Ejercicios 2

Ejercicio 1

Crea una función que sume los números en un slice de enteros. Define una interfaz **`Operacion`** con un método **`Operar`** que procese un slice de enteros. Implementa esta interfaz en dos estructuras, **`Suma`** y **`Promedio`**, cada una con su método **`Operar`**. En la función principal, usa una instancia de **`Operacion`** para operar en un slice de enteros y muestra el resultado.

Ejercicio 2

Desarrolla un programa para manejar empleados de tiempo completo y con contrato. Usa dos estructuras, una para cada tipo, con campos como salario y tasa horaria. Ambas deben implementar una interfaz **`Employee`** con un método **`GetSalary()`**. Crea una función `TotalExpense` que calcule el costo total de los salarios de un slice de empleados, usando **`GetSalary()`**. En **`main`**, crea instancias de ambos tipos de empleados, agrégalas a un slice y usa **`TotalExpense`** para mostrar el costo total en consola.

https://goplay.tools/snippet/t0wMMMDcFTz

Ejercicio 3

Escribe una función en Go que tome un puntero a una estructura llamada `Forma` que tenga un método llamado `Area()` que calcule el área de la forma. La estructura `Forma` debe ser una interfaz que incluya dos tipos de formas: un círculo y un rectángulo.

Luego, crea dos tipos de estructuras para representar un círculo y un rectángulo. Ambos deben tener campos que representen sus dimensiones, como el radio para el círculo y la base y la altura para el rectángulo.

Finalmente, escribe una función que tome una `Forma` y devuelva su área. La función debe ser capaz de manejar cualquier tipo de forma (círculo o rectángulo) y usar el método `Area()` correspondiente para calcular su área.

https://goplay.tools/snippet/saSgHLXCcId

Ejercicio 4

Crea una estructura llamada "Alumno" con los campos "nombre", "edad" y "promedio". Luego crea una interfaz llamada "Aprobado" con un método "EstaAprobado() bool" que determine si el alumno tiene un promedio mayor o igual a 7.0. Luego crea una función llamada "VerificarAprobado" que reciba un puntero a una "Alumno" y una interfaz "Aprobado" y determine si el alumno está aprobado o no.

Ejercicio 5

Crea una estructura llamada "Libro" con los campos "titulo" y "autor". Luego crea una interfaz llamada "Leible" con un método "Leer() string" que devuelva una cadena de texto que diga "Leyendo [titulo] de [autor]". Luego crea una función llamada "LeerLibro" que reciba un puntero a un "Libro" y una interfaz "Leible" y lea el libro.

Ejercicio 6

Crea una estructura llamada "Fraccion" con los campos "numerador" y "denominador". Luego crea una interfaz llamada "Operable" con los métodos "Sumar(f Fraccion) Fraccion", "Restar(f Fraccion) Fraccion", "Multiplicar(f Fraccion) Fraccion" y "Dividir(f Fraccion) Fraccion" que permitan realizar operaciones matemáticas con fracciones. Luego crea una función llamada "OperarFracciones" que reciba un puntero a una "Fraccion" y una interfaz "Operable" y permita realizar operaciones matemáticas con la fracción.

GIT

Recursos

- https://learngitbranching.js.org/
- https://quickref.me/git
- https://www.hschne.at/git-aliases/

Configuración Github

- ssh-keygen -t id_ed25519 -b 4096 -C "git-username"
  - Simplemente “enter” a todas las opciones o agregar la seguridad deseada.
    - **MELI pide contraseña para seguridad!!**
  - Copiar la clave pública: cat ~/.ssh/ id_ed25519.pub
  - Añadirla en: https://github.com/settings/keys
- Config usuario y email
  - git config --global user.name "git-username" && git config --global user.email "mail-username@mail.com"
- Revisar la config
- git config --list

Comandos más usados

1. `pull`: Fusiona los cambios remotos en tu rama local. Básicamente, `pull` descarga los cambios de la rama remota y los fusiona automáticamente en tu rama local. Supongamos que estás trabajando en una rama llamada `develop` y un miembro del equipo ha hecho cambios en la rama remota `master`. Para actualizar tu rama local `develop` con los cambios de `master`, puedes utilizar el comando:
   a. `git pull origin master`
   b. `Alias: ggl`
2. `fetch`: Descarga los cambios remotos en tu rama local, pero no los fusiona automáticamente. En su lugar, debes utilizar otros comandos, como `merge` o `rebase`, para fusionar los cambios manualmente. Imaginá que queres revisar los cambios de la rama remota `nueva-rama`, pero no quieres fusionarlos en tu rama local todavía. Para descargar los cambios de la rama remota sin fusionarlos automáticamente, puedes utilizar el comando:
   a. `git fetch origin nueva-rama`
   b. `Alias:gfo nueva-rama`
3. `add`: Se utiliza para añadir cambios a un área de preparación (staging) en Git. Después de realizar cambios en tus archivos locales, debes utilizar `add` para preparar esos cambios para el siguiente paso, que es el `commit`. Supongamos que has realizado cambios en el archivo `index.html` y quieres preparar esos cambios para el siguiente `commit`. Para añadir los cambios del archivo `index.html` al área de preparación (staging), puedes utilizar el comando:
   a. `git add index.html`
   b. `Para guardar todos los cambios en todos los archivos del directorio: git add -A`
   c. `Alias: gaa`
4. `commit`: Guarda los cambios en tu rama local. Cada `commit` crea una instantánea de los cambios que has realizado, lo que permite realizar un seguimiento de los cambios y retroceder en el tiempo si es necesario. Imagina que has añadido una nueva función a tu aplicación y quieres guardar los cambios en tu rama local. Para hacer un `commit` de los cambios en tu rama local, puedes utilizar el comando:

    a. `git commit -m "mi nuevo commit"`
    b. `Alias: gcmsg “my commit”`
5. `push`: Sube tus cambios locales a la rama remota. Después de realizar un `commit`, debes utilizar `push` para compartir tus cambios con los demás miembros del equipo. Supongamos que has hecho un `commit` en tu rama local y quieres subir esos cambios a la rama remota `master`. Para subir los cambios de tu rama local a la rama remota, puedes utilizar el comando:
    a. `git push origin master`
    b. `Alias: ggp`
6. `stash`: este comando se utiliza para almacenar temporalmente los cambios que no están listos para ser confirmados. Puedes utilizar `stash` para ocultar los cambios que estás haciendo y luego recuperarlos más tarde. imagina que estás trabajando en una nueva función, pero recibes una tarea urgente que debes completar. Para ocultar temporalmente los cambios que estás haciendo, puedes utilizar el comando `git stash`. Después de completar la tarea urgente, puedes utilizar el comando
    a. `git stash pop` Para recuperar los cambios que estabas haciendo anteriormente.
    b. `Alias:gst`
7. `Crear rama`: Crea una nueva rama en Git. Las ramas te permiten trabajar en diferentes versiones de tu código sin afectar a la rama principal. Puedes crear ramas para nuevas características, experimentos o solución de problemas. supongamos que estás trabajando en una nueva función y quieres crear una nueva rama para trabajar en ella sin afectar a la rama principal. Para crear una nueva rama llamada `new-feature`, puedes utilizar el comando:
    a. `git checkout -b nombre-rama`
    b. `Alias: gcb nombre-rama`
8. `Cambiar de rama:` Git cambia tu directorio de trabajo a la última instantánea en la rama especificada y actualiza el índice de Git para que coincida con la última instantánea. Esto significa que todos los cambios que hagas después de cambiar a la nueva rama serán específicos de esa rama y no afectarán a otras ramas.
    a. `git checkout nombre-rama`
    b. `Alias:` gco nombre-branch
9. `merge`: Fusiona dos ramas de Git en una sola rama. Cuando trabajas en diferentes ramas, debes utilizar `merge` para combinar los cambios de las diferentes ramas en una sola rama. Imagina que has trabajado en una rama llamada `nueva-rama` y ahora quieres fusionar los cambios en la rama principal. Para fusionar los cambios de `nueva-rama` en la rama principal, puedes utilizar el comando
    a. `git merge feature-branch`
    b. `Alias (develop en current branch):gm nueva-rama`
10. `Conflictos`: Los conflictos en Git ocurren cuando dos personas realizan cambios en la misma línea de código. Git no sabe cuál de las dos versiones debe mantener, y te pide que resuelvas el conflicto manualmente. Para resolver conflictos, debes comparar las dos versiones de la línea de código, elegir la versión correcta y guardar los cambios. Supongamos que dos miembros del equipo han realizado cambios en la misma línea de código en la rama remota `master`. Cuando intentas fusionar los cambios locales con la rama remota, Git detecta un conflicto. Para resolver el conflicto, debes abrir el archivo en cuestión, comparar las dos versiones de la línea de código y elegir la versión correcta. Después de guardar los cambios, puedes hacer un `commit` y continuar fusionando los cambios.

PRs

Comentarios en PRs

Resolver comentarios prs

Algoritmia

La algoritmia es una parte fundamental de la programación que se refiere al proceso de diseñar y desarrollar algoritmos para resolver problemas específicos de manera eficiente y efectiva.

Un algoritmo es una secuencia de pasos bien definidos y ordenados que describen cómo realizar una tarea en particular. En la programación, los algoritmos se utilizan para resolver problemas y realizar operaciones, como ordenar una lista de elementos, buscar un elemento en una base de datos, calcular una fórmula matemática, entre otros.

La algoritmia implica analizar y diseñar algoritmos, así como implementarlos en un lenguaje de programación específico. Para crear algoritmos efectivos, los programadores deben comprender los principios matemáticos subyacentes, la complejidad computacional y las estructuras de datos relevantes.

En resumen, la algoritmia es una habilidad esencial para cualquier programador que desee desarrollar software eficiente y efectivo.

Ejercicio 1: Escribe un programa que calcule la suma de los números enteros positivos del 1 al 100.

Ejercicio 2: Escribe un programa que calcule el factorial de un número entero positivo dado.

Ejercicio 3: Escribe un programa que calcule el número de dígitos de un número entero positivo dado.

Ejercicio 4: Escribe un programa que encuentre el número mayor de un conjunto de números enteros

Ejercicio 5: Escribe un programa que determine si una cadena de texto dada es un palíndromo.

Un palíndromo es una cadena que se lee igual de adelante hacia atrás que de atrás hacia adelante, por ejemplo "reconocer" o "ana".dados.

Ejercicio 6: Escribe un programa que encuentre los números primos menores que un número entero positivo dado.

Ejercicio 7: Escribe un programa que encuentre la subcadena común más larga entre dos cadenas de texto dadas.

Ejercicio 8: Suma de dos matrices aleatorias (los numeros se cargan de forma aleatoria).

Ejercicio 9: Multiplicación de dos matrices aleatorias.

Ejercicio 10: Buscar la salida de un laberinto

Ejercicio 11: Pila

Escribe una función que implemente una pila que permita realizar las siguientes operaciones: apilar un elemento, desapilar un elemento y obtener el elemento en la cima de la pila.

Ejercicio 12: Cola

Escribe una función que implemente una cola que permita realizar las siguientes operaciones: encolar un elemento, desencolar un elemento y obtener el elemento en la cabeza de la cola.

Ejercicio 13: Validación de paréntesis

Escribe una función que reciba una cadena que contiene paréntesis y corchetes, y que determine si los paréntesis y corchetes están correctamente equilibrados. Es decir, si cada paréntesis o corchete de apertura tiene su correspondiente paréntesis o corchete de cierre.

Ejercicio 14: Búsqueda binaria

Escribe una función que implemente la búsqueda binaria en una lista de enteros ordenados de forma ascendente. La función debe tomar como argumentos la lista de enteros y el valor que se desea buscar. Si el valor se encuentra en la lista, la función debe devolver el índice de la primera ocurrencia. Si el valor no se encuentra en la lista, la función debe devolver -1

Ejercicio 15: Búsqueda lineal

Escribe una función que implemente la búsqueda lineal en una lista de enteros. La función debe tomar como argumentos la lista de enteros y el valor que se desea buscar. Si el valor se encuentra en la lista, la función debe devolver el índice de la primera ocurrencia. Si el valor no se encuentra en la lista, la función debe devolver -1.

Ejercicio 16: Búsqueda en árbol binario de búsqueda (BST)

Escribe una función que implemente la búsqueda en un árbol binario de búsqueda (BST). El árbol binario de búsqueda es un árbol en el que cada nodo tiene un valor, y los valores de los nodos izquierdos son menores que el valor del nodo padre, mientras que los valores de los nodos derechos son mayores que el valor del nodo padre. La función debe tomar como argumentos la raíz del árbol y el valor que se desea buscar. Si el valor se encuentra en el árbol, la función debe devolver el nodo que contiene el valor. Si el valor no se encuentra en el árbol, la función debe devolver `nil`.

Ejercicio 17: Búsqueda en grafo con BFS

Escribe una función que implemente la búsqueda en un grafo utilizando el algoritmo BFS (Breadth-First Search). El grafo se puede representar como un mapa donde las claves son los nodos y los valores son las listas de nodos adyacentes. La función debe tomar como argumentos el grafo, el nodo de partida y el nodo destino. Si el nodo destino se encuentra en el grafo, la función debe devolver `true`. Si el nodo destino no se encuentra en el grafo, la función debe devolver `false`.

Inyección de Dependencias

La ***inyección de dependencias (DI)*** es un patrón de diseño en el que los objetos *no crean sus dependencias directamente*, sino que se ***les proporcionan a través de algún mecanismo externo***. Esto permite que los objetos sean más independientes y flexibles, y facilita la prueba y el mantenimiento del código.

Este ejemplo usa **`MyRepository`** como repositorio:

```
type MyRepository struct {
    data []string
}

func NewMyRepository() *MyRepository {
    return &MyRepository{}
}

func(r *MyRepository) Save(item string) error {
    r.data = append(r.data, item)
    return nil
}

func(r *MyRepository) GetAll() ([]string, error) {
    return r.data, nil
}
```

En Go, una forma común es a través de la función **`New`**. Por ejemplo se podría usar la función **`New`** para inyectar un repositorio en una estructura **`Items`**:

```
type Items struct {
    repo MyRepository
}

// la inyección de dependencias ocurre al pasar como parametro de entrada a repo de tipo
// MyRepository, en este caso se inyecta repo con toda las dependencias necesarias para que
// este pueda funcionar correctamente, viene con "baterias incluidas"
func NewItems(repo MyRepository) *Items {
    return &Items{
        repo: repo,
    }
}

func(i *Items) DoSomething() {
  i.repo.GetAll // ejemplo de utilizacion de la parametro inyectado
  ...
}
```

En este ejemplo, la estructura **`Items`** tiene un campo **`repo`** de tipo **`MyRepository`.** En la función **`NewItems`**, se toma un argumento **`repo`** de tipo **`MyRepository`** y se utiliza para inicializar el campo **`repo`** de la estructura **`Items`**. Luego, se devuelve un puntero a la estructura **`Items`**.

La idea es que el repositorio se cree y se configure fuera de la estructura `Items`, y luego se pase como argumento a la función `NewItems`. De esta manera, la estructura `Items` no necesita saber cómo se crea el repositorio o qué dependencias tiene, simplemente lo recibe y lo utiliza. De nuevo aplica la idea de "baterias incluidas" ,o sea, tiene las dependecias necesarias para funcionar correctamente, sin la necesidad de importar nada mas.

Ejemplo de cómo se podría usar la función **`NewItems`** en el **`main`**:

```
func main() {
    repo := &MyRepository{} // crear una instancia del repositorio concreto
    items := NewItems(repo) // inyectar el repositorio en la estructura Items
    // utilizar la estructura Items con el repositorio inyectado
    items.DoSomething()
}
```

Primero se crea una instancia del repositorio concreto **`MyRepository`.** Luego, se utiliza la función **`NewItems`** para inyectar el repositorio en la estructura **`Items`** y se guarda el resultado en una variable llamada **`items`**. Finalmente, se utiliza la estructura **`items`** para realizar alguna acción, en este caso llamando a un método **`DoSomething`** que probablemente utiliza el repositorio inyectado para obtener o almacenar datos.

Arquitectura

Simple main.go a Estructura Hexagonal

Polimorfismo

Inversion de dependendecias

Inyeccion de dependencias

Puertos y Adaptadores

DDD

Capas (domain, application, infrastructure)

Responsabilidades de las Capas (Handler, Controller, UseCase, etc)

HTTP, APIs REST

https://roadmap.sh/backend

Internet, server/client

Http

Context

APIs

Request

endpoints

routes

verbos (post,get,...)

http Status

API Errors

Body / Headers

Seguridad (jwt,oauth,basic)

Json

queryparam

Context

https://blog.golang.org/context

https://medium.com/@matryer/context-has-arrived-per-request-state-in-go-1-7-4d095be83bd8

https://peter.bourgon.org/blog/2016/07/11/context.html

Context es una herramienta que se puede usar con los patrones de diseño de concurrencia.

CI / CD

Docker

Pipelines

Jenkins

Cloud services

Application Health

Datadog

Alertas

Indicadores

App = Kibana ? Elastic

SOLID

Base de Datos

Configuración

Pasos para levantar una base de datos MySQL con docker (se deben ejecutar en un terminal):

1. Crear volumen:
   - **docker volume create dbdata**

   El volumen sirve para que nuestros datos se mantengan. Si no lo creamos cada vez que levantemos o paremos la base de datos no tendremos los datos.
2. Crear un contenedor con mysql:
   - **docker run -dp 3306:3306 --name mysql-db -e MYSQL_ROOT_PASSWORD=secret --mount src=dbdata,dst=/var/lib/mysql mysql:5.7**

   Si tenemos MAC con chip M1 debe ejecutar el siguiente comando:
   - **docker run --platform linux/x86_64 -dp 3306:3306 --name mysql-db -e MYSQL_ROOT_PASSWORD=secret --mount src=dbdata,dst=/var/lib/mysql mysql:5.7**

   Lo que vemos marcado en rojo es como montamos nuestro volumen a la carpeta de mysql donde almacena los datos
3. Creamos la base de datos (opción ingresando a una shell del contenedor):
   - **docker exec -it mysql-db bash**
   - **mysql -h localhost -u root -p**
   - Escribimos la contraseña (en nuestro caso: **secret**)
   - **CREATE DATABASE sales;**
4. Otra opción es crear la base de datos usando la aplicación MySQL Workbench
   - Podes descargarla de la pagina oficial: **https://dev.mysql.com/downloads/workbench/**

Ejercicios

- Crear tablas en base de datos:

```
CREATE TABLE customers (
  id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  lastname VARCHAR(100) NOT NULL,
  city VARCHAR(100),
  category INT UNSIGNED
);

CREATE TABLE agents (
  id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  lastname VARCHAR(100) NOT NULL,
  email VARCHAR(100),
  commission FLOAT
);

CREATE TABLE orders (
```

```
  id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  total DOUBLE NOT NULL,
  id_customer INT UNSIGNED NOT NULL,
  id_agent INT UNSIGNED NOT NULL,
  created_at DATE,
  FOREIGN KEY (id_customer) REFERENCES customers(id),
  FOREIGN KEY (id_agent) REFERENCES agents(id)
);
```

- Insertar registros:

```
INSERT INTO customers VALUES('Aarón', 'Rivero', 'Almería', 100);
INSERT INTO customers VALUES('Adela', 'Salas', 'Granada', 200);
INSERT INTO customers VALUES('Adolfo', 'Rubio', 'Sevilla', NULL);
INSERT INTO customers VALUES('Adrián', 'Suárez', 'Jaén', 300);
INSERT INTO customers VALUES('Marcos', 'Loyola', 'Almería', 200);
INSERT INTO customers VALUES('María', 'Santana', 'Cádiz', 100);
INSERT INTO customers VALUES('Pilar', 'Ruiz', 'Sevilla', 300);
INSERT INTO customers VALUES('Pepe', 'Ruiz', 'Santana', 'Huelva', 200);
INSERT INTO customers VALUES('Guillermo', 'López', 'Gómez', 'Granada', 225);
INSERT INTO customers VALUES('Daniel', 'Santana', 'Loyola', 'Sevilla', 125);

INSERT INTO agents VALUES('Daniel', 'Sáez', 'daniel@gmail', 0.15);
INSERT INTO agents VALUES('Juan', 'Gómez', 'juan@gmail', 0.13);
INSERT INTO agents VALUES('Diego','Flores', 'flores@gmail.com', 0.11);
INSERT INTO agents VALUES('Marta','Herrera', 'marta@gmail.com', 0.14);
INSERT INTO agents VALUES('Antonio','Carretero', 'ancarre@gmail.com', 0.12);
INSERT INTO agents VALUES('Manuel','Domínguez', 'manuel@gmail.com', 0.13);
INSERT INTO agents VALUES('Antonio','Vega', 'ant@mercadolibre.com', 0.11);
INSERT INTO agents VALUES('Alfredo','Ruiz', 'alfredoruiz@gmail.com', 0.05);

INSERT INTO orders VALUES(1, 150.5, '2017-10-05', 5, 2);
INSERT INTO orders VALUES(2, 270.65, '2016-09-10', 1, 5);
INSERT INTO orders VALUES(3, 65.26, '2017-10-05', 2, 1);
INSERT INTO order VALUES(4, 110.5, '2016-08-17', 8, 3);
INSERT INTO orders VALUES(5, 948.5, '2017-09-10', 5, 2);
INSERT INTO orders VALUES(6, 2400.6, '2016-07-27', 7, 1);
INSERT INTO orders VALUES(7, 5760, '2015-09-10', 2, 1);
INSERT INTO orders VALUES(8, 1983.43, '2017-10-10', 4, 6);
INSERT INTO orders VALUES(9, 2480.4, '2016-10-10', 8, 3);
INSERT INTO orders VALUES(10, 250.45, '2015-06-27', 8, 2);
INSERT INTO orders VALUES(11, 75.29, '2016-08-17', 3, 7);
INSERT INTO orders VALUES(12, 3045.6, '2017-04-25', 2, 1);
INSERT INTO orders VALUES(13, 545.75, '2019-01-25', 6, 1);
INSERT INTO orders VALUES(14, 145.82, '2017-02-02', 6, 1);
INSERT INTO orders VALUES(15, 370.85, '2019-03-11', 1, 5);
INSERT INTO orders VALUES(16, 2389.23, '2019-03-11', 1, 5);
```

FURY

MeliTool logger y custom errors

Next Level in GO

Aplicación

Application

JSON documentation

https://pkg.go.dev/encoding/json

https://pkg.go.dev/encoding/json#example-Marshal

https://pkg.go.dev/encoding/json#example-Unmarshal

Json: forma de formatear la info en una estructura de datos, que se usa para transmitir información entre aplicaciones.

- func Marshal(v interface{}) ([]byte, error)
- func Unmarshal(data []byte, v interface{}) error

JSON marshal

https://play.golang.org/p/tGMOgWrq2V7

https://pkg.go.dev/encoding/json#Marshal

Lo que hace esto es transformar la info en un json y json es perfecto transmitir.

JSON unmarshal

https://pkg.go.dev/encoding/json#Unmarshal

https://github.com/GoesToEleven/golang-web-dev/tree/master/040_json

https://play.golang.org/p/ffga0wGREvL

json: [{"Nombre":"Homero","Apellido":"Simpson","Edad":39},{"Nombre":"Marge","Apellido":"Bouvier Simpson","Edad":33}]

Pegar el json en: https://mholt.github.io/json-to-go/

Resultado:

```
type AutoGenerated []struct {
Nombre   string `json:"Nombre"`
Apellido string `json:"Apellido"`
Edad     int    `json:"Edad"`
}
```

Son tags:

`json:"Nombre"`

`json:"Apellido"`

`json:"Edad"`

CAMBIAR: type AutoGenerated []struct

type persona struct {

Nombre string `json:"Nombre"`

Apellido string `json:"Apellido"`

Edad int `json:"Edad"`

}

Al revés del anterior, se toma un json y se la transforma en algo que se pueda usar en go.

Writer interface

ESTE CAPÍTULO AYUDA A ENTENDER CÓMO ENTENDER LA LIBRERÍA ESTÁNDAR.

Encode y decode, hacen el trabajo sin asignarlo a una variable como marshal o unmarshal.

https://pkg.go.dev/encoding/json#Encoder

Encode

- func NewEncoder(w io.Writer) *Encoder
- func (enc *Encoder) Encode(v interface{}) error

https://pkg.go.dev/encoding/json#Decoder

Decode

- func NewDecoder(r io.Reader) *Decoder
- func (dec *Decoder) Decode(v interface{}) error

Lo que ocurre con estas funciones es:

Go -> encode (se adjunta a un writer (ej. archivo o webconexion)) -> envío

Arribo de datos -> decode (se adjunta un reader (ej. archivo, webconexion, etc)) -> Go

Writer:

https://pkg.go.dev/io#Reader

type Writer interface {

Write(p []byte) (n int, err error)

}

Es una interface, cualquier otro tipo que tenga adjunto el método Write(p []byte), también es de tipo Writer.

Ej:

https://pkg.go.dev/os

func Create(name string) (*File, error)

func (f *File) Write(b []byte) (n int, err error) <-- método requerido para implícitamente implementar la interface Writer

Ya que File tiene el método Write, cualquier valor sea un puntero a un File, también es de tipo Writer. Este puntero puede ser pasarte a NewEncoder y de ahí, a Enconde.

Reader:

https://pkg.go.dev/io#Reader

type Reader interface {

Read(p []byte) (n int, err error)

}

Es una interface, cualquier otro tipo que tenga adjunto el método Read(p []byte), también es de tipo Reader.

io.Reader y io.Writer funcionan casi igual.

https://pkg.go.dev/fmt#Println

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

// Println formats using the default formats for its operands and writes to standard output.

// Spaces are always added between operands and a newline is appended.

// It returns the number of bytes written and any write error encountered.

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

return Fprintln(os.Stdout, a...)

}

https://pkg.go.dev/fmt#Fprintln

func Fprintln(w io.Writer, a ...interface{}) (n int, err error)

// Fprintln formats using the default formats for its operands and writes to w.

// Spaces are always added between operands and a newline is appended.

// It returns the number of bytes written and any write error encountered.

func Fprintln(w io.Writer, a ...interface{}) (n int, err error) {

p := newPrinter()

p.doPrintln(a)

n, err = w.Write(p.buf)

p.free()

return

}

https://pkg.go.dev/os#pkg-variables

var (

Stdin = NewFile(uintptr(syscall.Stdin), "/dev/stdin")

Stdout = NewFile(uintptr(syscall.Stdout), "/dev/stdout")

Stderr = NewFile(uintptr(syscall.Stderr), "/dev/stderr")

)

https://pkg.go.dev/os#NewFile

func NewFile(fd uintptr, name string) *File

Como tengo un valor de tipo *File, ese valor también es de tipo Writer.

https://play.golang.org/p/u3ZcZuNdwe0

Sort

https://pkg.go.dev/sort

https://play.golang.org/p/Uz8_omoKVkk

Sort = ordenar

Sort custom

https://play.golang.org/p/MTxWthiH8H9

https://pkg.go.dev/sort#example-package

bcrypt

https://play.golang.org/p/kOv9mc4Ncv4

https://pkg.go.dev/golang.org/x/crypto/bcrypt

bcrypt es una de las mejores formas para guardar datos encriptados.

Para obtener bcrypt localmente: go get golang.org/x/crypto/bcrypt

Exercises - Ninja Level 8

Hands-on exercise #1

https://play.golang.org/p/Xp-YooW59Fi

Hands-on exercise #2

Ir a:

- https://mholt.github.io/json-to-go/

para obtener struct.

https://play.golang.org/p/XLg__1lTgmv

Hands-on exercise #3

https://play.golang.org/p/zdb5OrdbUoY

Hands-on exercise #4

https://play.golang.org/p/lg3g-RPkLtp

Hands-on exercise #5

https://play.golang.org/p/GiU7wuO3uHE

Concurrency

Concurrency

Concurrency vs parallelism

Go nativamente toma ventaja del hardware multicore, fue diseñado para esto.

La concurrencia está en el núcleo del diseño de Go.

Concurrencia vs. paralelismo

Paralelismo no es concurrencia:

- El paralelismo se ejecuta cuando se corre el software en una computadora con el procesador de más de un núcleo,
- La concurrencia es un patrón de diseño, es una forma de escribir el código, que potencialmente, tiene la facultad de ejecutarse de forma paralela, en un procesador multicore,

O sea, la concurrencia no garantiza que se vaya a correr de forma paralela, lo que hace que se corra o no de forma paralela es que si se tiene más de un CPU.

https://youtu.be/oV9rvDllKEg

WaitGroup

Paquete runtime

https://pkg.go.dev/runtime

https://golang.org/pkg/runtime/#pkg-constants

IMPORTANTE: Cuando “func main()” termina, TODO termina. Por eso es necesario algo como los WaitGroups.

Package runtime contains operations that interact with Go's runtime system, such as functions to control goroutines. It also includes the low-level type information used by the reflect package; see reflect's documentation for the programmable interface to the run-time type system.

// Devuelve la cantidad de CPUs corriendo donde se ejecuta el programa.

runtime.NumCPU()

// Devuelve la cantidad de goroutines corriendo.

runtime.NumGouroutine()

// Is the running program's architecture target: one of 386, amd64, arm, s390x, and so on.

runtime.GOARCH

// Is the running program's operating system target: one of darwin, freebsd, linux, and so on. To view possible combinations of GOOS and GOARCH, run "go tool dist list".

runtime.GOOS

Existe “func init() {}” corre cosas en el programa antes que nada.

Existe siempre una “main goroutine”.

go foo() // go lanza otra goroutine

Esto es código concurrente, y si se tiene más de CPU, está corriendo de forma paralela.

El control de flujo no tiene que esperar por foo() pq tiene su propia goroutine.

Para que finalicen las 2 goroutines hay que utilizar sincronización.

Para eso se crea una variable wg WaitGroup.

https://pkg.go.dev/sync

https://pkg.go.dev/sync#WaitGroup

```
var wg sync.WaitGroup
...
wg.Add(1)
go foo()
...
wg.Wait(1)
```

```
func foot() {
    ...
    wg.Done()
}
```

Una buena analogía para el código concurrente es pensar en un director de orquesta, en como le indica a los instrumentos (gouroutines) empezar, esperar, terminar.

https://play.golang.org/p/UJKjNZvuWA5

Method sets revisited

https://golang.org/ref/spec#Method_sets

https://go.dev/play/p/w9LvtiPnqwO

WaitGroup no cumple con esto:

type WaitGroup

func (wg *WaitGroup) Add(delta int)

func (wg *WaitGroup) Done()

func (wg *WaitGroup) Wait()

var wg sync.WaitGroup <--- no es puntero

¿Pq funciona así?

Method sets determinan qué métodos se adjuntan a un tipo. Es exacto lo que el nombre dice. ¿Cuál es el set de un método para un tipo dado? Ese es su method set.

Importante: "El method set de un tipo determina las interfaces que el tipo implementa..."

"Si el receptor es un puntero, solo se puede usar como valor de tipo puntero"

~/FreeCourseSite.com] Udemy - Learn How To Code Google's Go (golang) Programming Language/0. Apps/20/3/3.go

Documentation

https://golang.org/doc/effective_go#concurrency

Race condition:

- Se produce cuando 2 o más goroutines actúan sobre las mismas variables.
- El proceso de lectura/escritura resulta erróneo por el acceso simultáneo a las variables de las distintas goroutines.

Go usa channels para transmitir las variables o valores compartidos, nunca se comparten activamente por hilos de ejecución separados. Solo una goroutine tiene acceso al valor en un momento dado. Las data race no pueden ocurrir, por diseño. Para fomentar esta forma de pensar lo hemos reducido a un eslogan:

No se comunique compartiendo memoria; en su lugar, comparta la memoria comunicándose.

Este enfoque puede llevarse lejos. Los recuentos de referencias se pueden realizar mejor colocando un mutex alrededor de una variable entera, por ejemplo. Pero como enfoque de alto nivel, el uso de channels para controlar el acceso facilita la escritura de programas claros y correctos.

Una forma de pensar en este modelo es considerar un programa típico de un solo subproceso que se ejecuta en una CPU. No necesita primitivas de sincronización. Ahora ejecute otra instancia similar; tampoco necesita sincronización. Ahora deja que esos dos se comuniquen; si la comunicación es el sincronizador, todavía no hay necesidad de otra sincronización. Las tuberías de Unix, por ejemplo, se ajustan perfectamente a este modelo. Aunque el enfoque de Go para la concurrencia se origina en los procesos secuenciales de comunicación (CSP) de Hoare, también se puede ver como una generalización segura de tipos de tuberías Unix.

Se denominan goroutinas porque los términos existentes (hilos, corrutinas, procesos, etc.) transmiten connotaciones inexactas. Una goroutine tiene un modelo simple: es una función que se ejecuta simultáneamente con otras goroutines en el mismo espacio de direcciones.

Las goroutines se multiplexan en varios subprocesos del sistema operativo, por lo que si uno se bloquea, mientras se espera la E/S, otros continúan ejecutándose. Su diseño oculta muchas de las complejidades de la creación y gestión de hilos.

NOTA: multiplexor o mux, se toman muchas entradas y se las ordena en una única salida.

go nombre_funcion o método <--- Ejecuta la llamada en una nueva goroutine.

Cuando se completa la llamada, la goroutine sale silenciosamente.

Una función literal puede ser útil en una invocación de goroutine.

```
func Announce(message string, delay time.Duration) {
go func() {
time.Sleep(delay)
        fmt.Println(message)
}() // Tenga en cuenta los paréntesis: debe llamar a la función.
}
```

En Go, los function literals son cierres: la implementación se asegura de que las variables a las que hace referencia la función sobrevivan mientras estén activas.

Estos ejemplos no son demasiado prácticos porque las funciones no tienen forma de indicar la finalización. Para eso, necesitamos channels.

https://golang.org/ref/spec#Go_statements

A "go" statement starts the execution of a function call as an independent concurrent thread of control, or goroutine, within the same address space.

GoStmt = "go" Expression .

The expression must be a function or method call; it cannot be parenthesized. Calls of built-in functions are restricted as for expression statements.

The function value and parameters are evaluated as usual in the calling goroutine, but unlike with a regular call, program execution does not wait for the invoked function to complete. Instead, the function begins executing independently in a new goroutine. When the function terminates, its goroutine also terminates. If the function has any return values, they are discarded when the function completes.

go Server()

go func(ch chan<- bool) { for { sleep(10); ch <- true }} (c)

https://play.golang.org/p/lBKFKCwrue

Race condition

~/FreeCourseSite.com] Udemy - Learn How To Code Google's Go (golang) Programming Language/0. Apps/20/5/5.go

código con una data race condition

$ go help

$ go help build

$ go run -race 5.go # lo corre y verifica si hay o no una data race condition

La situación “race” se da cuando múltiples goroutines intentan acceder a la misma variable. En este caso el contador. Se arregla con mutex.

Mutex

~/FreeCourseSite.com] Udemy - Learn How To Code Google's Go (golang) Programming Language/0. Apps/20/6/6.go

$ go run -race 6.go

https://play.golang.org/p/1v44E7bEKOk

Se usa mutex para lockear un pedazo de código para que múltiples goroutines no puedan acceder al mismo tiempo y prevenir una race condition. Una buena analogía sería, leer un libro de la biblioteca, ninguna goroutine puede usar el libro

hasta que la que lo está leyendo termine y lo devuelva.

https://pkg.go.dev/sync#Mutex.Lock

type RWMutex

func (rw *RWMutex) Lock() // lockea para L/E

func (rw *RWMutex) RLock() // lock sólo para escritura

func (rw *RWMutex) RLocker() Locker

func (rw *RWMutex) RUnlock()

func (rw *RWMutex) Unlock()

mu.Lock()

```
v := counter
runtime.Gosched()
v++
counter = v
```

// todo lo que esta entre mu.Lock() y mu.Unlock, no puede ser utilizado por nadie más

mu.Unlock()

Lo que está entre mu.Unlock() queda bloqueado hasta q termine, es como si estuviera leyendo el libro, hasta que no lo termine no podrá ser usado por otro lector.

Atomic

https://golang.org/pkg/sync/atomic/

~/FreeCourseSite.com] Udemy - Learn How To Code Google's Go (golang) Programming Language/0. Apps/20/7/7.go

Lo mismo que hace mutex, pero con atomic.

Siempre que se vea “int64” pensar en el paquete atomic.

Exercises - Ninja Level 9

Hands-on exercise #01

https://play.golang.org/p/yVooyV_KB7E

Hands-on exercise #02

https://play.golang.org/p/yL3xnUK92rl

https://play.golang.org/p/3w3ynCw_ids

Hands-on exercise #03

0. Apps/21/3/3.go

Hands-on exercise #04

0. Apps/21/4/4.go

Hands-on exercise #05

0. Apps/21/5/5.go

Hands-on exercise #06

0. Apps/21/6/6.go

Hands-on exercise #07

Na

Channels

Understanding channels

https://golang.org/doc/effective_go#concurrency

https://golang.org/ref/spec#Types

Con los paquetes como atomic o mutex se puede coordinar los diferentes partes de la que juegan con el paralelismo y la concurrencia.

LOS CANALES SE BLOQUEAN

~/0. Apps/22/1

Directional channels

Podes decidir si el channel solo envía o recibe.

c<-42 // el channel solo recibe

<-c // el channel solo envía

~/0. Apps/22/2

Using channels

~/0. Apps/22/3

Range

~/0. Apps/22/4

Cómo cerrar un channel.

Select

~/0. Apps/22/5

Los channels se bloquean!

Un range se bloquea hasta que el channel se cierra.

Comma ok idiom

~/0. Apps/22/6

https://stackoverflow.com/questions/10437015/does-a-channel-return-two-values

Fan in

https://play.golang.org/p/c6nC3Ifm20x

https://play.golang.org/p/78Wg_ojscR-

Fan in es un patrón de concurrencia.

Poner 2 o más channels en 1.

Fan out

https://play.golang.org/p/xbta1cQ3tks

https://play.golang.org/p/rloEQcXeHEU

Fan out es un patrón de concurrencia.

Poner 1 channel en 2 o más para trabajar en paralelo.

Context

https://blog.golang.org/context

https://medium.com/@matryer/context-has-arrived-per-request-state-in-go-1-7-4d095be83bd8

https://peter.bourgon.org/blog/2016/07/11/context.html

Context es una herramienta que se puede usar con los patrones de diseño de concurrencia.

Cancela TODAS las goroutines de un proceso, cuando este cancelado.

fan in

Background

https://play.golang.org/p/cByXyrxXUf

WithCancel

throwing away CancelFunc

https://play.golang.org/p/XOknf0aSpx

using CancelFunc

https://play.golang.org/p/UzQxxhn_fm

Example

https://play.golang.org/p/Lmbyn7bO7e

func WithCancel(parent Context) (ctx Context, cancel CancelFunc)

https://play.golang.org/p/wvGmvMzIMW

cancelling goroutines with deadline

func WithDeadline(parent Context, deadline time.Time) (Context, CancelFunc)

https://play.golang.org/p/Q6mVdQqYTt

with timeout

func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)

https://play.golang.org/p/OuES9sP_yX

with value

func WithValue(parent Context, key, val interface{}) Context

https://play.golang.org/p/8JDCGk1K4P

Exercises - Ninja Level 10

Hands-on exercise #1

https://play.golang.org/p/5gFYG08ZpLL

https://play.golang.org/p/zSla9Xxi7Zn

Hands-on exercise #2

https://play.golang.org/p/BhhgKXOYAgA

https://play.golang.org/p/QHxrG8UEiuq

Hands-on exercise #3

https://play.golang.org/p/xFOZGWvit4z

Hands-on exercise #4

https://play.golang.org/p/-WSTGZuqWn-

Hands-on exercise #5

https://play.golang.org/p/ckF6W1kF5mv

Hands-on exercise #6

https://play.golang.org/p/TMZK3ehBiWw

Hands-on exercise #7

https://play.golang.org/p/yjONCJccH3v

Error handling

https://golang.org/doc/faq#exceptions

https://blog.golang.org/error-handling-and-go

Para el manejo simple de errores, las devoluciones de valores múltiples de Go facilitan la notificación de un error sin sobrecargar el valor devuelto.

En Go se intentan poner las cosas justo en donde ocurren, para arreglar o manejar mejor todos los eventos.

```
func leerArchivo(nombreArchivo string) ([]byte, error) {
   contenido, err := ioutil.ReadFile(nombreArchivo)
   if err != nil {
      return nil, err //Devuelve error si no se pudo leer el archivo
   }
   return contenido, nil
}
```

En este caso, la función leerArchivo devuelve un valor []byte asociado con el contenido del archivo proporcionado y también un valor error en caso de que la lectura del archivo fuera imposible.

Además, para revisar si una operación a ocurrido con éxito sin un error, se puede comprobar si el valor retornado en estado "nil".

En Go, la interfaz de error es error. Esta interfaz predefinida en el paquete builtin se utiliza para representar un error que se produjo durante la ejecución del programa.

La interfaz error tiene solo un método, Error() string, que devuelve una descripción del error como una cadena. Todos los tipos que implementan este método son considerados tipos de errores en Go.

Si una función puede arrojar un error durante su ejecución, por lo general se espera que devuelva un valor error adicional para identificar y describir el error. Por ejemplo:

```
type error interface {
    Error() string
}
```

Simplemente es otro TIPO.

En Go, para personalizar la interfaz de error se debe implementar la interfaz error en una estructura propia.

```
type MyError struct {
    Msg string
}

func (e *MyError) Error() string {
    return fmt.Sprintf("Error occurred: %v", e.Msg)
}
```

En el ejemplo anterior, se define una estructura llamada MyError que contiene un campo Msg. A continuación, se añade un método llamado Error() a la estructura MyError, que devuelve el mensaje de error formateado como un string. Este método satisface la interfaz error.

Podemos usar esta estructura personalizada de errores en nuestro código:

```
func myFunc(input int) error {
    if input < 0 {
        return &MyError{"negative number"}
    }
    // ...
    return nil
}
```

Dentro de la función myFunc(), si el valor del parámetro input es negativo, se devuelve una instancia de MyError con el mensaje "negative number". En caso contrario, se devuelve nil.

Checking errors

(casi) SIEMPRE REVISAR LOS ERRORES

https://play.golang.org/p/ynWwRe34Osf

Printing and logging

https://play.golang.org/p/L5KT5v_Z5jh

- fmt.Println()
- log.Println()
- log.Fatalln()
    - os.Exit()
- log.Panicln()
    - deferred functions run
    - can use “recover”
- panic()

Recover

https://blog.golang.org/defer-panic-and-recover

https://play.golang.org/p/Ujf1dRatTMb

Go tiene los mecanismos habituales para controlar el flujo: if, for, switch, goto. También tiene la instrucción go para ejecutar código en una goroutine separada. Aquí me gustaría discutir algunos de los menos comunes: **defer, panic**, y **recover**.

Recover es una función incorporada que recupera el control de una goroutine en pánico. Recover sólo es útil dentro de funciones diferidas. Durante la ejecución normal, una llamada para recuperar devolverá cero y no tendrá ningún otro efecto. Si la goroutine actual está entrando en pánico, una llamada para recuperar capturará el valor dado al pánico y reanudará la ejecución normal.

Errors with info

Cómo añadir información a los errores.

- errors.New()
    - fmt.Errorf()
- builtin.error

“Error values in Go aren’t special, they are just values like any other, and so you have the entire language at your disposal.”

Exercises - Ninja Level 11

Hands-on exercise #1

https://play.golang.org/p/Q1Zt8mBN3Y6

Hands-on exercise #2

https://play.golang.org/p/xj1gM_279wj

Hands-on exercise #3

https://play.golang.org/p/M5OgYq1cldO

conversion: https://play.golang.org/p/LbsrPc-a1tC

assertion: https://play.golang.org/p/L-tVaWSmPdZ

Hands-on exercise #4

https://play.golang.org/p/-8IdPeNt0Jv

Hands-on exercise #5

http://www.golang-book.com/books/intro/12

https://pkg.go.dev/testing

Writing documentation

Writing documentation

Introduction

https://pkg.go.dev/

https://pkg.go.dev/search?q=fmt <-- standard library

https://pkg.go.dev/github.com/google/uuid <-- librería de terceros

https://golang.org/

https://golang.org/pkg/

https://golang.org/help/

go doc

Comando de la terminal

- go help
- go help comando
- go doc # prints package documentation for the package in the current directory
- go doc <pkg>
- go doc <sym>[.<method>] # go doc fmt.Printf
- go doc [<pkg>.]<sym>[.<method>]
- go doc [<pkg>.][<sym>.]<method>
- go help doc # para ver ejemplos

En la práctica es mejor ir a la web: https://pkg.go.dev/

godoc

https://golang.org/doc/cmd

https://pkg.go.dev/golang.org/x/tools/cmd/godoc

Lo mejor es que se puede correr desde un servidor local, si no se tiene internet todavía se tiene acceso a la documentación por este medio.

No me funca godoc, falta configurar, mirar después.

godoc.org

Ahora se usa https://pkg.go.dev/

- Push a github
- Se copia el url
- en pkg.go.dev se pega el url

- y listo!

Writing documentation

Simplemente los comentarios en el código son la documentación.

// comentario <-- en pkg.go.dev se usará como documentación cuando se suba el paquete

doc.go

ejemplo: https://golang.org/src/fmt/doc.go

Es importante hacer buenos y suficientes comentarios en el código.

La documentación es una gran parte para hacer que el software sea accesible y mantenible. Por supuesto, debe estar bien escrito y ser preciso, pero también debe ser fácil de escribir y mantener. Idealmente, debería estar acoplado al código mismo para que la documentación evolucione junto con el código. Cuanto más fácil sea para los programadores producir una buena documentación, mejor para todos.

Exercises - Ninja Level 12

Hands-on exercise #1

na

Testing & benchmarking

Testing & benchmarking

Introduction

https://www.golang-book.com/books/intro/12

https://github.com/GoesToEleven/go-programming/tree/master/code_samples/009-testing/01-test

Los tests son importante pq te aseguran que tu código hace lo que realmente queres q haga. Simplemente es más programación.

Los tests deben:

- El archivo debe terminar con "_test.go"
- El archivo debe estar en el mismo package que el que testea.
- Debe estar en una func de esta forma "func TestXxx(*testing.T)" (debe empezar con mayúscula, son buenas práticas)
- Se corren los tests con "go test"

Table tests

https://github.com/GoesToEleven/go-programming/tree/master/code_samples/009-testing/02-table-test

Una tabla de tests, es una tabla donde hay muchos distintos tests para hacer varias pruebas a la vez.

Example tests

Documentación y tests simultáneamente.

https://pkg.go.dev/strings#example-ToUpper

https://github.com/GoesToEleven/go-programming/tree/master/code_samples/009-testing/03-examples

https://blog.golang.org/examples

Golint

gofmt # formats go code

go vet # reports suspicious constructs

golint ./... # recomendaciones en este y todos los demás directorios

Benchmark

https://github.com/GoesToEleven/go-programming/tree/master/code_samples/009-testing/04-benchmark

Mide el performance del código.

Parte del paquete de prueba nos permite medir la velocidad de nuestro código. Esto también podría llamarse "medir el rendimiento" de su código, o "comparar" su código, es decir, averiguar qué tan rápido se ejecuta el código.

$ go test -bench <dir>

$ go help testflag

-bench ...

Coverag

Coverag: cuanto del código escrito fue cubierto por tests. Es buena práctica cubrir cuanto más sea posible. Un 70% es un buen valor.

code:

go test -cover

1. **go test -coverprofile c.out**
    a. **show in browser:**
        i. **go tool cover -html=c.out**
    b. **learn more**
        i. **go tool cover -h**

Benchmark examples

https://github.com/GoesToEleven/go-programming/tree/master/code_samples/009-testing/04-benchmark/03-cat

Para ver con ejemplos como distintos algoritmos que hacen lo mismo son más o menos eficientes.

Review

Review de los comandos de benchmark, ejemplos y tests:

- godoc -http=:8080
- go test
- go test -bench .
    - don't forget the "." in the line above
- go test -cover
- go test -coverprofile c.out
- go tool cover -html=c.out

Mocking

Mocks te permiten simular comportamientos y controlar las expectativas de las dependencias en tus pruebas, lo que facilita el aislamiento y la prueba de componentes individuales de tu código.

Antes de empezar, necesitarás instalar Gomock y su generador de mocks llamado "mockgen". Puedes hacerlo ejecutando los siguientes comandos:

go get github.com/golang/mock/gomock
go install github.com/golang/mock/mockgen@latest

Ejemplo unit test simple

unit test con handler

unit test con mock

Exercises - Ninja Level 13

---

1. **Hands-on exercise #1**

0. Apps/29/1

1. **Hands-on exercise #2**

0. Apps/29/2

1. **Hands-on exercise #3**

0. Apps/29/3

Generics

Chat GPT

NOTAS

hacer un crud y meter un kvs, agregar metricas,

el id es la fecha

abordar el tema de fury y kvs, hacer el rampup haciendo el onboarding

meter el rampup en la route:
https://learninghub-int.mercadolibre.com/courses/course-v1:it_boarding+rampup-BE+2022-Q2/courseware/c1d742bdf13d48748363bf00366ed153/f80115c379984507aaf8cc2024095fe3/?child=last

wave 3, desde punteros (incluido)

Principios SOLID y Clean Code

¿Por qué son importantes los principios SOLID y Clean Code?

Los principios SOLID y Clean Code son fundamentales para el desarrollo de software de alta calidad por varias razones:

Legibilidad y Mantenibilidad

El código que sigue los principios SOLID y de Clean Code es más fácil de leer y mantener. Los nombres claros y significativos, las funciones pequeñas y concisas, y los comentarios útiles hacen que el código sea más comprensible para los desarrolladores, lo que facilita su mantenimiento y actualización.

Modularidad y Reutilización

Los principios SOLID promueven la creación de sistemas modulares con responsabilidades bien definidas. Esto facilita la reutilización de código, ya que los módulos pueden ser acoplados y desacoplados fácilmente según sea necesario.

Escalabilidad

El cumplimiento de los principios SOLID también conduce a sistemas más escalables. A medida que tu código crece, la adherencia a estos principios te permitirá manejar esta creciente complejidad de manera más efectiva.

Menos Errores

El Clean Code minimiza los errores y los problemas de rendimiento. Cuando las funciones hacen una sola cosa y los efectos secundarios se minimizan, es menos probable que surjan problemas y errores difíciles de detectar.

Colaboración y Comunicación

El código limpio es también sobre comunicación. Un código bien estructurado y claro es más fácil de entender para otros desarrolladores. Esto es especialmente importante cuando trabajas en equipos grandes o cuando tu código va a ser mantenido por otros en el futuro.

Eficiencia en el Desarrollo

Finalmente, tanto SOLID como Clean Code pueden conducir a un desarrollo más eficiente. Aunque puede llevar más tiempo al principio adherirse a estos principios, a largo plazo se ahorra tiempo ya que el código es más fácil de mantener, reutilizar y ampliar.

Principios SOLID

SOLID

Es un acrónimo en inglés ingles de:

- **S** = El Principio de responsabilidad única (**S**ingle Responsibility Principle)
- **O** = El Principio Abierto-Cerrado (**O**pen-Closed Principle)
- **L** = El Principio de sustitución de Liskov (**L**iskov Substitution Principle)
- **I** = El Principio de segregación de interfaz **(I**nterface Segregation Principle)
- **D** = El Principio de inversión de dependencia (**D**ependency Inversion Principle)

S. Principio de Responsabilidad Única (Single Responsibility Principle, SRP)

Este principio dice que una clase o módulo debería tener solo una responsabilidad. En Go, que es un lenguaje sin clases, este principio se aplica a los tipos y funciones.

Una clase debería tener solo una razón para cambiar.

Ejemplo 1: En lugar de tener una estructura que maneje tanto la creación como la eliminación de archivos

```
type FileHandler struct {
     //...
}

func (f *FileHandler) Create() error {...}
func (f *FileHandler) Delete() error {...}
```

Podemos separar estas responsabilidades en dos estructuras

```
type FileCreator struct {
     //...
}
func (f *FileCreator) Create() error {...}

type FileDeletor struct {
    //...
}
func (f *FileDeletor) Delete() error {...}
```

Ejemplo 2: Cada metodo se encarga de solo una tarea.

```
type Order struct {
  customer Customer
}
func (o *Order) calculateTotalSum() {...}
func (o *Order) getItems() {...}
func (o *Order) getItemCount() {...}
```

```
func (o *Order) addItem(item Item) {...}
func (o *Order) deleteItem(item Item) {...}
```

O. Principio Abierto-Cerrado (Open Closed Principle, OCP)

Este principio establece que los módulos, funciones o entidades deberían estar abiertos para la extensión, pero cerrados para la modificación. En Go, esto puede lograrse mediante la implementación de interfaces.

Las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas para la extensión, pero cerradas para la modificación.

Ejemplo:

```
type Shape interface {
   Area() float64
}

type Rectangle struct {
    Width, Height float64
}

func (r Rectangle) Area() float64 {
    return r.Width * r.Height
}

type Circle struct {
    Radius float64
}

func (c Circle) Area() float64 {
    return math.Pi * c.Radius * c.Radius
}
```

La función `CalculateTotalArea` está cerrada para modificación pero abierta para extensión.

No necesitamos modificarla para calcular el área de nuevas formas.

```
func CalculateTotalArea(shapes []Shape) float64 {
   var totalArea float64
          for _, shape := range shapes {
              totalArea += shape.Area()
          }
  return totalArea
}
```

La función CalculateTotalArea recibe un slice de formas (shapes) y calcula el área total. La belleza de este diseño radica en que si necesitamos añadir una nueva forma (por ejemplo, un triángulo), no necesitamos cambiar la función CalculateTotalArea. Solo necesitamos asegurarnos de que la nueva estructura Triangle implemente la interfaz Shape. Esto es estar "abierto a la extensión".

Por otro lado, el código dentro de CalculateTotalArea no necesita ser modificado para manejar nuevos tipos de formas, lo que significa que está "cerrado para la modificación".

L. Principio de Sustitución de Liskov (Liskov Substitution Principle, LSP)

El principio dice que si una clase B es una subclase de la clase A, entonces deberíamos poder reemplazar A con B sin interrumpir el comportamiento del programa.

Para entenderlo de una forma más sencilla, supongamos que tenemos:

- Un pato de goma.
- Un pato real.

Ambos pueden flotar, hacer ruido, etc., por lo que puedes considerar que el pato de goma es un subtipo del pato real. Pero, ¿qué pasa si decides agregar la función de volar?

Los patos reales pueden volar, pero los patos de goma no.

En este caso, el pato de goma no puede ser sustituido por un pato real, por lo que viola el principio de sustitución de Liskov.

Ejemplo:

```
type Bird interface {
 Fly() string
}

type Duck struct{}

func (d Duck) Fly() string {
 return "Duck is flying"
}

type RubberDuck struct{}

func (rd RubberDuck) Fly() string {
 return "Rubber duck can't fly!"
}

func LetItFly(b Bird) {
 fmt.Println(b.Fly())
}
```

Aquí, tanto `Duck` como `RubberDuck` implementan la interfaz `Bird`, pero cuando intentas hacer que `RubberDuck` vuele, obtienes un resultado completamente diferente. Esto violaría el principio de sustitución de Liskov si estuvieras esperando que todas las implementaciones de `Bird` pudieran volar.

Por lo tanto, una manera de adherirse al principio de sustitución de Liskov sería garantizar que todos los métodos definidos por una interfaz sean soportados de la misma manera por todas las implementaciones de esa interfaz.

Para adherirse podríamos estructurar nuestros tipos y interfaces de manera que todas las implementaciones de una interfaz puedan soportar todos los métodos de la misma manera.

Veamos un ejemplo modificado del anterior para hacerlo adherirse al principio de sustitución de Liskov:

```
type Bird interface {
 Swim() string
 MakeNoise() string
}
```

```
type Duck struct{}

func (d Duck) Swim() string {
 return "Duck is swimming"
}

func (d Duck) MakeNoise() string {
 return "Duck says quack!"
}

type RubberDuck struct{}

func (rd RubberDuck) Swim() string {
 return "Rubber duck is floating"
}

func (rd RubberDuck) MakeNoise() string {
 return "Rubber duck says squeak!"
}

func InteractWithBird(b Bird) {
 fmt.Println(b.Swim())
 fmt.Println(b.MakeNoise())
}
```

En este caso, hemos eliminado el método `Fly()` que no podía ser soportado por `RubberDuck` de la misma manera que por `Duck`. Ahora, tanto `Duck` como `RubberDuck` son intercambiables desde el punto de vista de la interfaz `Bird`. Ambos pueden "nadar" y hacer un "ruido" sin problemas.

Este es un ejemplo simple, pero la idea principal es que cuando defines una interfaz, todas las implementaciones de esa interfaz deben comportarse de la misma manera para cumplir con las expectativas del código que utiliza esa interfaz. Esto facilita el razonamiento sobre el código y hace que el sistema sea más flexible y fácil de mantener.

I. Principio de Segregación de la Interfaz (Interface Segregation Principle, ISP)

Este principio establece que los clientes no deberían ser forzados a depender de interfaces que no utilizan. En Go, esto significa que deberíamos preferir muchas interfaces más pequeñas en lugar de una grande.

Los clientes no deben verse forzados a depender de interfaces que no utilizan.

En lugar de tener una interfaz grande.

Ejemplo 1:

```
type Horas interface {
    Trabajar()
    Descansar()
}
```

Podríamos tener dos interfaces más pequeñas

```
type HorasTrabajo interface {
    Trabajar()
}
```

```
type HorasDescanso interface {
    Descansar()
}
```

Los tipos pueden implementar una o ambas interfaces según sea necesario

```
type Humano struct{}

func (h Humano) Trabajar() {
    fmt.Println("Human is working")
}

func (h Humano) Descansar() {
    fmt.Println("Human is resting")
}

type Robot struct{}

func (r Robot) Trabajar() {
    fmt.Println("Robot is working")
}
```

El robot no necesita implementar Descansar(), por lo que no está obligado a implementar la interfaz HorasDescanso.

Ejemplo 2:

```
type Printer interface {
    Print()
}

type Scanner interface {
    Scan()
}

type Photocopier struct {}

func (p Photocopier) Print() {...}
func (p Photocopier) Scan() {...}
```

D. Principio de Inversión de Dependencias (Dependency Inversion Principle, DIP)

Este principio dice que las dependencias en los módulos, clases y funciones deberían ser en términos de abstracciones, no en términos de concreciones. En Go, esto se logra a través del uso de interfaces.

Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de abstracciones.

Ejemplo: En lugar de depender directamente del paquete sql.DB

```
type UserRepository struct {
```

```
    db *sql.DB
}
```

Dependemos de una abstracción:

```
type Datastore interface {
    Query(query string, args ...interface{}) (*sql.Rows, error)
}

type UserRepository struct {
    store Datastore
}
```

Clean Code

1. Nombres significativos

Los nombres de las variables, funciones, clases y módulos deben ser descriptivos y expresar el propósito de la entidad que están identificando. Un buen nombre es esencial para entender el código y su funcionamiento.

Los nombres en tu código deben ser descriptivos y significativos.

Ejemplo:

```
// En lugar de
var t int

// Usar
var timeoutInSeconds int
```

2. Funciones pequeñas con un solo trabajo

Las funciones deben ser pequeñas y hacer una sola cosa. Deben tener pocos argumentos y evitar efectos secundarios. Las funciones largas y complicadas son difíciles de entender y mantener.

Las funciones deben ser pequeñas y deben hacer una sola cosa.

Ejemplo: En lugar de tener una función que hace todo

```
func processOrder(id int) {
    // valida la orden
    // recupera detalles de la orden
    // procesa la orden
    // actualiza el estado de la orden
}
```

Tenemos varias funciones más pequeñas, cada una de las cuales hace una cosa

```
func validateOrder(id int) {...}
func retrieveOrderDetails(id int) {...}
func performOrderProcessing(details OrderDetails) {...}
func updateOrderStatus(id int, status string) {...}
```

3. Comentarios solo cuando sea necesario

Los comentarios pueden ser útiles para explicar el propósito del código, pero el mejor código es el que se explica por sí mismo. Además, los comentarios pueden volverse obsoletos si no se mantienen actualizados con el código.

Los comentarios pueden ser útiles, pero el mejor código es aquel que se explica por sí mismo.

Ejemplo: i es el índice

```
    for i := 0; i < len(items); i++ {
        // procesar item
    }
```

Usar nombres significativos para que los comentarios no sean necesarios

```
    for index := 0; index < len(items); index++ {
        processItem(items[index])
    }
```

4. Manejo de errores en lugar de ignorarlos

El código debe manejar errores explícitamente y no contener sentencias `panic()`. Las funciones deben devolver errores que indiquen claramente qué salió mal.

Siempre debes manejar los errores explícitamente.

Ejemplo 1:

```
    _, err := doSomething()
    if err != nil {
        // manejar error en lugar de ignorarlo
        log.Fatalf("Error doing something: %v", err)
}
```

Ejemplo 2:

```
file, err := os.Open("file.txt")
if err != nil {
    log.Fatal(err)
}
defer file.Close()
```

Este código intenta abrir un archivo y comprueba si se produjo un error. Si se produce un error, el programa registra el error y se detiene. Esta es una buena práctica porque permite al programa manejar errores en lugar de fallar silenciosamente.

5. Formato

El código debe estar correctamente formateado y ser consistente. El formato consistente facilita la lectura y comprensión del código. Go tiene la herramienta `go fmt` que formatea automáticamente el código.

El código debería estar correctamente formateado y consistente.

6. Principio DRY (Don't Repeat Yourself, No Te Repitas)

La idea es que cada pieza de lógica debe tener una única representación en el código. Si tienes bloques de código duplicados en varios lugares y necesitas realizar un cambio, tendrías que recordar actualizarlo en todos esos lugares. Esto aumenta la probabilidad de errores y hace que el código sea más difícil de mantener. Al seguir el principio DRY, te aseguras de que cada cambio solo necesita realizarse en un lugar.

Se centra en la eliminación de la redundancia en el código.

7. Principio KISS (Keep It Simple, Stupid, Manténlo Simple, Estúpido)

La idea es que un diseño simple es siempre mejor que un diseño complicado, siempre que se logren los objetivos deseados. Un código más simple es más fácil de entender, de mantener y de menos propenso a errores. Al seguir el principio KISS, te esfuerzas por evitar la complejidad innecesaria.

Se trata de mantener el código lo más simple y directo posible.

8. Principio YAGNI (You Aren't Gonna Need It, No Lo Vas a Necesitar)

Este principio sugiere que los desarrolladores no deben añadir funcionalidades hasta que realmente se necesiten. Esto puede evitar la creación de código innecesario y complicado que puede ser difícil de mantener.

No añadir nada, hasta necesitarlo.

9. Refactorización

La refactorización es el proceso de reorganizar o cambiar el código sin cambiar su comportamiento externo. El objetivo es hacer que el código sea más eficiente y/o legible. Este es un aspecto esencial del código limpio, ya que un código limpio y eficiente es más fácil de mantener y mejora la calidad del software.

Upgrade de calidad del codigo.

10. Pruebas unitarias y cobertura de pruebas

Las pruebas unitarias ayudan a asegurar que cada parte del código funcione correctamente y permiten una refactoring más segura. Un alto nivel de cobertura de pruebas indica que una gran parte del código está probada y es probable que esté libre de errores.

Testing, testing, testing y... testing coverage.

11. Consistencia

El código debe ser consistente en su estilo y convenciones. Esto incluye la nomenclatura, la indentación, la ubicación de los paréntesis, etc.

La consistencia hace que el código sea más fácil de leer y de entender.

REVISAR

Devolver nil o estructura vacia

La respuesta a esta pregunta puede depender del contexto y de las convenciones del código en el que estás trabajando. Sin embargo, aquí dejo algunos puntos a considerar:

Devolver nil: En Go, nil es un valor válido para cualquier tipo de interfaz, puntero, mapa, canal, función y slice. Devolver nil puede ser útil cuando quieres indicar EXPLÍCITAMENTE que no hay un valor válido para devolver, ejemplo:

Cuando una función no pudo encontrar un recurso o completar una tarea. En este caso, normalmente se devuelve un error junto con el nil para proporcionar más contexto.

Devolver una estructura vacía: Puede ser útil si queres evitar que el código que llama a tu función tenga que lidiar con punteros y pueda usar el valor devuelto inmediatamente. Devolver una estructura vacía puede ayudar a evitar errores de puntero nulo (nil pointer dereference). Sin embargo, puede ser más difícil para el código que llama a la función determinar si la función pudo completar su tarea correctamente, a menos que incluyas información de estado adicional en la estructura.

En resumen:
Si queres indicar la ausencia de un valor o un error, devolves nil.
Si quieres que el código que llama a tu función pueda usar el valor devuelto inmediatamente, o si quieres incluir información de estado adicional, devolves una estructura vacía.

Conversiones

Cuando pones el tipo entre paréntesis después de una expresión en Go, estás realizando una conversión explícita de tipo, similar a lo que se explicó anteriormente. La sintaxis es la siguiente:

```go
tipoDestino(expresión)
```

Donde `tipoDestino` es el tipo al que deseas convertir la expresión.

Esta sintaxis se utiliza cuando quieres realizar una conversión de tipo en una expresión específica y no quieres que afecte al tipo original de la expresión. En otras palabras, la conversión con paréntesis crea una nueva expresión con el tipo deseado sin modificar la expresión original.

Veamos un ejemplo para clarificarlo:

```go
package main

import (
    "fmt"
)

func main() {
    var a int32 = 42
    var b int64 = int64(a) // Conversión explícita de int32 a int64
    fmt.Printf("Valor de b: %d, Tipo de b: %T\n", b, b)
}
```

En este ejemplo, tenemos una variable `a` de tipo `int32`. Al utilizar la conversión explícita `int64(a)`, creamos una nueva variable `b` de tipo `int64` con el valor de `a`. Sin embargo, la variable original `a` sigue siendo de tipo `int32`. La salida sería:

```
Valor de b: 42, Tipo de b: int64
```

El valor de `b` es 42, y su tipo es `int64`, mientras que `a` mantiene su tipo original, que es `int32`. La conversión explícita no afecta a la variable original, solo crea una nueva variable con el tipo deseado.

Aserciones

```go
package main

import (
 "fmt"
)

func main() {
 var myInterface interface{} = 42 // Declaramos una variable de tipo interfaz con un valor entero.

 // Intentamos realizar una aserción para obtener el valor entero.
 if myInt, ok := myInterface.(int); ok {
  fmt.Println("La aserción fue exitosa.")
  fmt.Println("Valor entero:", myInt)
 } else {
  fmt.Println("La aserción falló.")
 }
}
```

En este ejemplo, creamos una variable `myInterface` de tipo interfaz y le asignamos el valor entero `42`. Luego, realizamos una aserción para intentar obtener el valor entero de la interfaz.

Si la aserción tiene éxito, imprimimos "La aserción fue exitosa" junto con el valor entero. En caso de que la aserción falle, imprimimos "La aserción falló".

Dado que `myInterface` contiene un valor entero y hemos realizado una aserción para obtenerlo como `int`, la aserción será exitosa y obtendremos la siguiente salida:

```
La aserción fue exitosa.
Valor entero: 42
```

Generics

Los generics, introducidos en Go 1.18, son una característica que permite escribir funciones y tipos parametrizados, lo que significa que pueden trabajar con diferentes tipos de datos sin duplicar el código. Antes de la introducción de los generics, los programadores de Go debían utilizar interfaces vacías (empty interfaces) y realizar conversiones de tipos en tiempo de ejecución para lograr un comportamiento genérico, lo cual era menos seguro y menos eficiente.

Con los generics, puedes definir funciones y tipos que acepten parámetros de tipo, lo que te permite escribir código más reutilizable y eficiente.

Ejemplo Completo: https://goplay.tools/snippet/fSEot03HI33

En este ejemplo, definimos una función genérica `Swap` que puede intercambiar dos valores de cualquier tipo que cumpla con la restricción `any` (cualquier tipo). Luego, en la función `main`, utilizamos la función `Swap` para intercambiar dos enteros y dos cadenas de texto.

La función genérica `Swap` se define utilizando parámetros de tipo (en este caso, `T`). Los parámetros de tipo se especifican entre corchetes angulares `[]` después del nombre de la función. La restricción `any` indica que la función genérica puede aceptar cualquier tipo de dato.

Este ejemplo muestra cómo los generics en Go permiten escribir código más reutilizable y eficiente, ya que puedes utilizar la misma función para diferentes tipos de datos sin duplicar el código.