Golang Fundamentals

Introduccion

Go es un lenguaje de programación nuevo creado por Rob Pike, Robert Griesemer y Ken Thompson. Es un lenguaje estáticamente tipado y con un modelo de programación concurrente nativo del lenguaje. Algunas características muy marcadas del lenguaje es que implementa herencia por composición, está pensado para desarrollo en infraestructuras modernas con un modelo concurrente nativo y que tiene una comunidad muy grande en continuo crecimiento. Iremos viendo lo mejor de go en los siguientes encuentros. Primero lo primero.. la instalación y los fundamentos del lenguaje.

Instalación de Go

La instalacion de golang es super simple, hay que seguir pasos que se detallan en el siguiente link oficial: [Golang Installation]

Si sos de los que no le gusta leer mucho, te puedo resumir en:

1.- Te bajas la version que te corresponda segun tu sistema operativo desde: https://golang.org/dl/
2.- Lo instalas según tu sistema operativo, por ejemplo ejecutando el msi en windows.
3.- Configurar las variables de entorno que corresponden (en linux):

```
export GOPATH=$HOME/go
export PATH=$PATH:$GOPATH/bin
```

Para testear la instalación vamos a ejecutar el famoso hello world. Para esto vamos a crear una carpeta src/hello dentro del `$GOPATH` y dentro de esa carpeta un archivo con el nombre `hello.go` con el siguiente contenido:

```
package main
import "fmt"
func main() {
      fmt.Printf("hello, world\n")
}
```

Entonces ejecutamos los siguientes comandos (en linux):

```
$ cd $HOME/go/src/hello
$ go build
$ ./hello
```

Con estos comandos nos posicionamos dentro de la carpeta creada anteriormente, buildeamos el contenido (hello.go) y ejecutamos el binario que haya generado (según el sistema operativo puede ser un binario sin extensión o un exe, etc).

Encoding y Editor

El editor que más se usa para el desarrollo con golang actualmente es visual code de microsoft (download link) aunque también hay varios que usan VIM y otros editores. Yo recomiendo Visual Code.

El encoding en golang (o tan solo ***go***) es UTF-8. Esto nos permite escribir código ya en UTF sin tener que escapar o encodear los strings para poder visualizarlos como queremos, como así también nos permite escribir código en idiomas que antes no se podían, por ejemplo:

```
package main

import "fmt"

func 隨機名稱() {
  fmt.Println("It works!")
}

func main() {
  隨機名稱()
  źdźbło := 1
  fmt.Println(źdźbło)
}
```

Fundamentals!

Ya tenemos instalado el lenguaje y el editor, ya vimos cómo crear binarios de nuestros programas, ahora es tiempo de dar una recorrida rápida por el lenguaje para escribir nuestros propios programas. Antes de entrar en detalles del lenguaje, para todos aquellos que no hayan instalado aún el lenguaje y quieran comenzar con los fundamentals golang provee de un playground que es una plataforma que nos permite escribir pequeños programas y ejecutarlos online. Se puede acceder al playground desde el siguiente link

Ahora si.. vamos a lo básico pero divertido.

Packages

Un sistema escrito en go está estructurado por paquetes, que esencialmente son directorios y subdirectorios con archivos dentro. Como muchos lenguajes de programación, golang tiene un entry point (lo primero que se ejecuta cuando ejecutamos el binario) que es una función llamada `main` en un paquete llamado `main`. Los paquetes se importan con el keyword `import` (y como se iba a llamar??):

```
import (
  “math”
  “fmt”
)
```

También se puede separar en distintas sentencias `import`. Si queremos importar nuestros propios paquetes y no los que ya trae golang, simplemente ponemos la ruta relativa desde el comienzo del workspace.. ah, no lo dije pero el workspace se llama donde apunta $GOPATH. Entonces si queremos importar nuestra propia lib (llamada lib.go) que tenemos en `$GOPATH/src/tudai/2019/internal` debemos hacerlo del siguiente modo:

```
package main

import (
    "fmt"
    "tudai/2019/internal"
)

func main() {
    fmt.Println("hello world!")
```

```
    fmt.Println(internal.Sum(1, 1))
}
```

Noten que el paquete se puede usar con el nombre del último directorio en la ruta de importación, en este caso "`internal`".

Visibilidad de nombres

Si venís del mundo java seguramente te cruzaste con nombres interminables como `public static void main`. Go no tiene modificadores de acceso tales como público, privado, protegido, etc, simplemente en go los nombres (variables, funciones, métodos) son o públicos o con visibilidad de paquete. Para declararlos públicos se comienza el nombre con mayúsculas. Si es público hay que documentarlo. Por ejemplo, nuestra librería lib.go tiene el siguiente contenido:

```
package internal


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

Donde "`Sum`" es público.

Funciones

Las funciones en golang pueden tomar cualquier cantidad de argumentos y devolver cualquier cantidad de valores y el patrón típico es devolver los valores que uno quiere devolver un error. Los errores los veremos luego.

```
func Split(sum int) (x, y int) {
    x = sum * 4 / 9
    y = sum - x
    return
}
```

Por ejemplo la función anterior usa las variables resultado x e y definidas en la declaración de la función como variables para retornar el resultado (named result values). Yo particularmente prefiero usar variables en el cuerpo de la función, por ejemplo:

```go
func Split(sum int) (int, int) {
    x := sum * 4 / 9
    y := sum - x
    return x, y
}
```

Donde x e y son declarados en el cuerpo y retornados con el keyword `return`.

Algo interesante que tiene golang son las variadic functions. Estas son funciones que pueden tomar indeterminado número de parámetros de un mismo tipo al final de la lista de parámetros. Por ejemplo:

```go
func sum(nums ...int) {
    fmt.Print(nums, " ")
    total := 0
    for _, num := range nums {
        total += num
    }
    fmt.Println(total)
}

func main() {
    sum(1, 2)
    sum(1, 2, 3)

    nums := []int{1, 2, 3, 4}
    sum(nums...)
}
```

Si revisan bien el código de la función main verán que hay algo como una colección de elementos. Si bien parece un array no lo es, es un slice pero como muchas otras veces lo dijimos, lo veremos más adelante. Por otro lado pueden ver una sentencia for un tanto extraña, por ahora confíen en que recorre el slice elemento a elemento.

Sentencia de Control

For

En golang no existe tal cosa como while o repeat, solo existe la sentencia for. Esto simplifica mucho al compilador y a la legibilidad del código dando los mismos resultados que si tuviéramos repeat y while.

```
    for i := 0; i < 2; i++ {
        // code
    }
```

Como verán el for es similar al for del lenguaje C, solo que tiene algunas variantes como por ejemplo se pueden omitir la primer y tercer parte quedando básicamente un while:

```
    var i int = 1
    for i < 2 {
        // code
    }
```

Aunque también podemos omitir todos los argumentos y tenemos un loop infinito:

```
    for {
        // code
    }
```

Range

Primero veamos algo de código para analizarlo:

```
package main
```

```go
import "fmt"

func main() {

    // Here we use `range` to sum up the numbers in a slice.
    // Arrays work like this too.
    nums := []int{2, 3, 4}
    sum := 0
    for _, num := range nums {
        sum += num
    }
    fmt.Println("sum:", sum)

    // `range` on arrays and slices provides both the
    // index and value for each entry. Above we didn't
    // need the index, so we ignored it with the
    // blank identifier `_`. Sometimes we actually want
    // the indexes though.
    for i, num := range nums {
        if num == 3 {
            fmt.Println("index:", i)
        }
    }

    // `range` on map iterates over key/value pairs.
    kvs := map[string]string{"a": "apple", "b": "banana"}
    for k, v := range kvs {
        fmt.Printf("%s -> %s\n", k, v)
    }

    // `range` can also iterate over just the keys of a map.
    for k := range kvs {
        fmt.Println("key:", k)
    }

    // `range` on strings iterates over Unicode code
    // points. The first value is the starting byte index
    // of the `rune` and the second the `rune` itself.
    for i, c := range "go" {
```

```
        fmt.Println(i, c)
    }
}
```

Si bien esta explicado con comments vamos a ver algunos detalles para aclarar los slices y arreglos. Los slices son punteros a estructuras que almacenan un array, un length y un capacity, los veremos luego. Entonces:

```
    nums := []int{2, 3, 4}
```

Declara un slice de 3 elementos.

```
for _, num := range nums {
        sum += num
    }
```

Va sumando los valores e ignorando los índices con el _.

```
for i, num := range nums {
        if num == 3 {
            fmt.Println("index:", i)
        }
    }
```

Aca voy iterando hasta encontrar el elemento cuyo valor es 3 y muestro la posición en el slice. El del map es muy deducible.

```
 kvs := map[string]string{"a": "apple", "b": "banana"}
```

Declaré un map con el key del tipo string al igual que el value. Esto equivale a un Map<String, String> en java.

```
for k, v := range kvs {
        fmt.Printf("%s -> %s\n", k, v)
    }

    // `range` can also iterate over just the keys of a map.
    for k := range kvs {
        fmt.Println("key:", k)
    }
```

Recorro el mapa en el primer for y en el segundo solo los keys.

Una cadena puede ser vista como una secuencia de runas, entonces podríamos recorrerlas una a una con el siguiente código:

```
for i, c := range "go" {
        fmt.Println(i, c)
    }
```

If

La sentencia `if`, al igual que la sentencia for, no llevan paréntesis. En el `if` se pueden poner asignaciones antes de ejecutar la sentencia booleana propia del `if`. Por ejemplo:

```
if x, y := internal.Split(10); x > 1 && y < 10 {
        // something
    }
```

Switch

La sentencia `switch` se la puede ver como una secuencia de sentencias `if-else`. Esta sentencia ejecuta el primer caso cuya condición es cumplida. La diferencia entre el `switch` de c, c++, java y otros es que no hay que poner el horrible break para que no se ejecuten los demas casos, en golang se ejecuta el primer caso que cumple y finaliza la sentencia `switch`. Otra gran diferencia es que en golang los casos a comparar no necesitan ser constantes ni enteros, esto facilita mucho la aplicación.

```
    var day string = "lunes"
    switch day {
    case "lunes":
        fmt.Println("arranca la semana.. pilas!")
    case "martes":
        fmt.Println("maldicion!, falta toda la semana!")
    case "miercoles":
        fmt.Println("ya falta menos")
    case "jueves":
        fmt.Println("mañana es viernes!")
    case "viernes":
        fmt.Println("yey!!, arranca el finde!!")
    default:
```

```
        fmt.Println(":D")
    }
```

Defer

Esta es una sentencia que se usa mucho en golang. `Defer` es una sentencia que se ejecuta cuando finaliza la función en la que es contenida. Es importante destacar que los parametros de ejecución de `defer` se calculan inmediatamente pero al mismo tiempo se apila en el stack de ejecución para que al terminar la función que contiene al `defer`, se ejecuta este último con los parámetros correctos.

```
    fmt.Println("counting")
    for i := 0; i < 10; i++ {
        defer fmt.Println(i)
    }
    fmt.Println("done")
```

Si ejecuta el bloque anterior tendrán en la salida primero “done” y luego la lista de números en forma descendente (porque fueron apilados). La salida es:

```
done
9
8
7
6
5
4
3
2
1
0
```

Variables y tipos

En golang existen casi todos los tipos típicos de los lenguajes. Contiene:

```
bool
string
int  int8  int16  int32  int64
uint uint8 uint16 uint32 uint64 uintptr
byte // alias for uint8
rune // alias for int32
     // represents a Unicode code point
```

`float32 float64`
`complex64 complex128`

Como es de esperar estos tipos tienen operadores comunes de concatenación, suma, resta, operadores lógicos, etc. También podemos definir constantes con el keyword `const`. Estos tipos son utilizados para declarar variables y constantes. Las variables se pueden declarar de tres formas en golang como muestra el siguiente snippet de código:

```
// Declare with no initialization, go will give the default value
    // which in this case is 0.
    var someInt int
    // Giving it a specific initial value
    var someBool, hi = true, "hi"
    // Inferring the type from the right side of the expression
    hello := "hello world" // hello will be of type string

    // declaring constants. Type gets inferred
    const number = 2;
    const str = "some string";

    fmt.Printf("someInt: %v\n", someInt)
    fmt.Printf("someBool: %v\nhi: %v\n", someBool, hi)
    fmt.Printf("helloWorld: %v\n", hello)
    fmt.Printf("number: %v\n", number)
    fmt.Printf("str: %v\n", str)
```

Cuando se declara una variable sin valor inicial el compilador asigna el zero-value a la variable. En el caso de `int` es `0`, en `bool` es `false`, en punteros es `nil`, etc. Si necesitas saber algún otro cero value podes recurrir a la documentación online de golang.

Para realizar casting en golang simplemente encerramos entre paréntesis la variable a castear precedido por el tipo al cual queremos castear. Por ejemplo:

```
var x int = 10
   var f float32
   f = float32(x)
```

Pointers
Golang tiene punteros. Un puntero almacena la dirección de memoria de un valor. La declaración es la misma que en C, de modo que podemos crear un puntero a un tipo específico como:

```
  var ptr *int
```

De modo que la variable ptr almacena la posición de memoria que apunta a un entero. Por ejemplo:

```
var someInt *int
   otherInt := 2
   pointer := &otherInt
   fmt.Printf("zero-value of someInt: %v\n", someInt)
   fmt.Printf("value pointer points to: %v\n", *pointer)
```

En este caso la salida es <nil> y 2 respectivamente. Esto es porque el zero-value de un pointer es nil y porque *pointer es el valor almacenado en la posición de memoria pointer y en la sentencia pointer := &otherInt estamos copiando la posición de memoria de otherInt en pointer, de modo que ambas variables (pointer y otherInt) apuntan a la misma posición que almacena 2. Parece un trabalenguas pero es simple si se tiene en cuenta que:

- Definimos un puntero a un tipo con el *
- Obtenemos la dirección de memoria de una variable con &
- Accedemos al contenido de la memoria a la cual apunta un puntero con el *

Structs

Las estructuras son utilizadas para agrupar campos bajo un mismo nombre. La definición de un struct es similar a como se hace en C: `type name struct`. Cada campo dentro del `struct` va a tener su propio tipo y pueden ser inicializados de diferentes formas como muestra el siguiente ejemplo:

```
type someStruct struct {
    intField  int
    boolField bool
}

func main() {c
    // this will have the zero-value for all its fields
    s := someStruct{}
    // fields can get initialized by order
    s = someStruct{2, true}
```

```
    // we can name the fields we are initializing
    l := someStruct{
        boolField: false,
        intField:  2,
    }

    // We can then access the fields of the struct using the
    // dot notation:
    fmt.Printf("l bool field: %v\n", l.boolField)
    fmt.Printf("s int field field: %v\n", s.intField)
}
```

Noten que cuando inicializamos una estructura con los nombres de los campos y el valor, la última inicialización de campo lleva una coma al final (`intField:  2,`).

Si declaramos un puntero a una estructura lo natural es pensar que para acceder al valor del puntero deberíamos usar * pero no, golang es tan copado que nos evita poner los tres caracteres extra que deberíamos usar, por ejemplo (*algo).campo, entonces el siguiente ejemplo pero es totalmente válido:

```
   var x *someStruct
   (*x).boolField = true
   x.boolField = true
```

Nota: en la estructura `someStruct` los campos son “privados” o con visibilidad package ya que comienzan con minúsculas.

Las estructuras también pueden ser anónimas:

```
    // see that it has no name that can be used to reference
    // the "type" of this structure
    s := struct {
        x int
        y int
    }{
        x: 2,
        y: 3,
   }
    fmt.Printf("x: %v y: %v", s.x, s.y)
```

Esto es de mucha utilidad cuando uno hace testing.

Herencia vs Composición

Como ya dijimos antes, en golang todo tiene que ver con la composición, de hecho no existe la herencia en go. Esto puede parecer raro si venís del mundo de java, pero la composición es una técnica muy utilizada en oop y golang hace muchisimo uso de la misma. Veamos el siguiente ejemplo para ver la composición en acción:

```
package main

import "fmt"

type User struct {
    ID   int
    Name string
}

type Gamer struct {
    User
    FavoriteGame string
}

func main() {

    gamer := Gamer{}
    gamer.ID = 0
    gamer.Name = "juan pablo"
    gamer.FavoriteGame = "doom"

    fmt.Println(gamer)
}
```

Si vemos el ejemplo, la estructura `Gamer` está compuesta por un campo (field) anónimo `User` y un field `FavoriteGame`. Los campos anónimos como User pueden ser usados como lo muestra el ejemplo, que a simple vista parece ser parte de gamer como podemos ver en la linea `gamer.ID = 0`. Cuidado que si bien parece ser parte de la estructura Gamer, al imprimir la variable gamer por pantalla podemos ver la diferencia:

```
{{0 juan pablo} doom}
```

Como era de esperarse, la parte resaltado en rojo es el `User` y el resto es el `Gamer`.

Arrays, Slices y Maps

En go existe algo que puede confundir a los que se introducen al lenguaje y es la diferencia entre arrays y slices. Como pueden imaginar un array es una lista de tamaño fijo, indexada y con elementos del mismo tipo. La primera diferencia entre slices y arrays es que el slice no tiene un tamaño fijo sino que puede crecer dinámicamente. Algo que es relevante es que los slices no contienen los datos sino que tienen un puntero a un array que los contiene. En efecto un slice es un struct con 3 campos, cap (capacity), len (length) y data. Length cómo se puede esperar es el largo utilizado del array (data) y capacity es la capacidad máxima del array. Si vienen del lado de java, un slice es algo similar a un arraylist. En el siguiente ejemplo se puede ver cómo se manipulan los arreglos y los slices.

```
package main

import "fmt"

func main() {

    // This is how you would declare an array

    // var someArray [3]int

    // Slices can be declared in many different ways

    // var names []string

    // slice of strings with an initial size of 2 and unlimited capacity

    // otherNames := make([]string, 2)

    // slice of strings with an initial size of 2 and a maximum capacity of 4

    // capacity := make([]string, 2, 4)

    // We can initialize with values

    numbers := []int{1, 2, 3, 4, 5}

    // Slices let us do operations using the indices

    oneToThree := numbers[0:2]

    fmt.Println(oneToThree)

    // We can omit one of the indices and it will go to the last or the first
```

```
    threeToFive := numbers[2:]

    fmt.Println(threeToFive)

    fmt.Println(numbers)

    // Incrementing the size of slice.

    // If we want to append values to already declared slice we can use the append
function

    oneToThree = append(oneToThree, 4)

    fmt.Println(oneToThree)

    // append doesn't care about the receiver, so we could declare a new variable with
append:

    fiveToSix := append(threeToFive[len(threeToFive)-1:], 6)

    fmt.Println(fiveToSix)

}
```

La parte importante es de todo esto es saber que un array tiene un tamaño fijo y que se le otorga dicho valor al crearlo y que un slice es una estructura que tiene un puntero a un array (data) y guarda los valores len (la cantidad de elementos) y cap (la capacidad máxima). El slice se crea con la función make que toma tres parámetros que son el tipo del slice, el length y la capacidad. También se puede crear de la forma `slice := []int`.

```
package main



import "fmt"



func main() {

    s := make([]int, 0, 3)

    for i := 0; i < 15; i++ {

        s = append(s, i)

        fmt.Printf("cap %v, len %v, %p\n", cap(s), len(s), s)

    }
```

```
}
```

Si vemos el programa anterior, el slice tiene una capacidad inicial de 3 elementos, tiene un length inicial de 0 y es del tipo []int. El programa tiene la siguiente salida por consola:

```
λ go run main.go
cap 3, len 1, 0xc00008c000
cap 3, len 2, 0xc00008c000
cap 3, len 3, 0xc00008c000
cap 6, len 4, 0xc00009c000
cap 6, len 5, 0xc00009c000
cap 6, len 6, 0xc00009c000
cap 12, len 7, 0xc00009e000
cap 12, len 8, 0xc00009e000
cap 12, len 9, 0xc00009e000
cap 12, len 10, 0xc00009e000
cap 12, len 11, 0xc00009e000
cap 12, len 12, 0xc00009e000
cap 24, len 13, 0xc0000a0000
cap 24, len 14, 0xc0000a0000
cap 24, len 15, 0xc0000a0000
```

Como podemos ver, al llegar al límite de la capacidad del slice, golang duplica el tamaño del arreglo data y copia los elementos de un arreglo origen al arreglo destino.

Los mapas son básicamente diccionarios en python o hashmaps en java. Los mapas relacionan un key con un value, o sea, es una colección de key-values. El zero value de

un map es nil como los slices y también de igual forma los mapas se pueden crear con make.

```
package main


import "fmt"


type someStruct struct {

    intField  int

    boolField bool

}


func main() {

    // this a nil map since it's the zero-value

    // var m map[string]someStruct


    // declaring and initializing maps

    m := map[string]someStruct{

        "key1": someStruct{2, false},

        "key2": someStruct{3, true},

    }

    // We can reference the fields of the struct by accessing the value of the map

    fmt.Printf("m[\"key1\"].intField=%v\n", m["key1"].intField)

    // We can allocate new key-value pairs

    m["key3"] = someStruct{4, true}

    fmt.Printf("m[\"key3\"].boolField=%v\n", m["key3"].boolField)

}
```

Como se puede ver en el ejemplo anterior, creamos un mapa cuyo key es `string` y value `someStruct` que se definió anteriormente. En esta ocasión se creó utilizando el operador `:=` .

La forma de acceder a los valores es obviamente utilizando su key de la forma `m["the-key"].blah`. Este es otro ejemplo de como crear e iterar un mapa:

```
package main


import "fmt"


type someStruct struct {

    intField  int

    boolField bool

}


func main() {

    var m map[int64]string

    m = make(map[int64]string)


    m[0] = "some-value"

    for k, v := range m {

        fmt.Printf("the value associates to the key %v is \"%s\"\n", k, v)

    }

}
```

Como pueden ver definimos el mapa `m` sin inicializarlo, luego con el make le damos un valor inicial y con `m[0] = "some-value"` le damos el primer elemento. En el loop `for` podemos ver como ahora el operador `range` devuelve el `key` y el correspondiente `value`.

Funciones como tipos

En go las funciones son first class citizens, esto quiere decir que las funciones pueden estar, y de hecho están, en casi cualquier lado. Las funciones pueden ser tratadas como tipos tales como int, string, etc, esto nos habilita a poder pasar funciones por parámetros o crear tipos. El siguiente ejemplo ilustra cómo pueden usarse las funciones como tipos:

```
package main


import (
    "errors"
    "fmt"
)


func main() {
    var fun func() error
    fun = func() error {
        fmt.Println("hi from fun")
        return errors.New("error from fun")
    }
    someFunc(fun)
}


func someFunc(f func() error) error {
    fmt.Println("hi from someFunc")
    // return the result of calling f
    return f()
}
```

Como podemos ver, en la función `main()` se declara `fun` como una función que devuelve un `error`, luego de inicializar fun la paso como parametro a otra función llamada `someFunc`. La función `someFunc` simplemente imprime un valor por pantalla y luego ejecuta `fun()` y retorna el resultado (en una sola línea). Si inspeccionan el código fuente de cualquier librería de golang verán que las funciones son casi vitales para que todo funcione.

Metodos

Go, a diferencia de otros lenguajes orientados a objetos no posee clases, pero sí métodos que se pueden definir en tipos concretos. Esto puede ser un poco raro pero termina siendo una función con un *receiver* especial. El *receiver* puede ser cualquier tipo, int, structs, strings y cualquier cosa que imagines (pueden ser incluso otras funciones). Debido a que el *receiver* puede ser cualquier tipo, este puede tener cierto estado. Por ejemplo, si yo tengo una estructura con dos fields entonces esos fields van a ser accesibles desde el método que se definió para esta estructura. Si queres modificar esos estados y queres que se refieren fuera del método, tenes que definir el *receiver* como puntero, esto significa que el receiver almacenará la dirección de memoria del estado a ser manipulado. Por conveniencia golang nos deja llamar a métodos que tienen un puntero de receiver con un valor, esto quiere decir que no es necesario llamar de la forma (&v),Method() sino que solo de la forma v.Method(). Por ejemplo:

```
package main

import "fmt"

type mstr string

func (m *mstr) Hi() {

    // print the value of the receiver

    fmt.Printf("Hi %v\n", *m)

}

func main() {

    // we can call hi using a value, go will translate to
```

```
    // a memory address since that is what Hi expects as the
    // receiver
    name := mstr("Gopher")
    name.Hi()

    // we can call hi too when we have an actual memory address
    otherName := &name
    otherName.Hi()

}
```

Interfaces

Las interfaces nos permiten agrupar funciones. El valor de las interfaces puede ser cualquier valor que implemente dichos métodos. En go las interfaces son implementadas implícitamente, esto quiere decir que a diferencia de java no tenemos un keyword para anunciar que una clase implementa una interfaz (implements en java). Si un tipo dado implementa la/las funciones especificadas en la interfaz entonces ese tipo implementa dicha interfaz. Una característica cool de las interfaces en go es que pueden ser nil, entonces esencialmente podes llamar a un método en un valor nil lo que en otros lenguajes arrojaría un null pointer exception. Te podes estar preguntando: entonces qué pasa si la interfaz no tiene métodos?, no sería que todos los tipos implementan esa interface?, la respuesta es si. Esto a veces es conveniente cuando se manipulan valores de cualquier tipo. Por ejemplo, en el paquete json de la librería standard tenes el método Encoder.Encode que recibe un dato el cual queres convertir (encodear) y al estar el parámetro definido como interface{} este método puede recibir cualquier cosa. **CUIDADO** cuando uses interfaces vacías, no hagas abuso de las mismas ya que si estás definiendo comportamiento siempre conviene ponerlos bajo un mismo nombre, de este modo se puede usar el polimorfismo, es más legible el código, debugueable, etc. También, go recomienda reducir el número de métodos de las interfaces, esto es: cuanto más grande la interfaz más débil será la abstracción.

```
package main

import "fmt"

// Note the format. Typically one method interfaces have
// a the name of the interface be the name of the method
// + er. For example, the interface that has the method
// Write is called Writer, in this case the method name
// is greet so the interface will be called Greeter.
type Greeter interface {
    Greet(name string)
}
```

```
type str struct{}

// str type implements the Greeter interface implicitly.
func (str) Greet(name string) {
    fmt.Printf("Hi %v\n", name)
}

func main() {
    // s is of type str so it implements Greeter
    s := str{}
    s.Greet("gopher")

    // whatType receives an empty interface so we can
    // send whatever we want to the function
    whatType(s)
    whatType(3)
    whatType("hi")

    // greetAndBye expects a Greeter, so whatever type
    // that implements the Greet(name string) method
    // can be sent to this function
    greetAndBye(s, "gopher")
}

func whatType(v interface{}) {
    fmt.Printf("type of %v: %T\n", v, v)
}

func greetAndBye(g Greeter, name string) {
    g.Greet(name)
    fmt.Printf("Bye %v\n", name)
}
```

Assertions de tipos

Existe algo llamado type assertion.. Esto provee acceso a los valores de una interfaz bajo un tipo concreto. Esto básicamente significa que dada una interface podes extraer el valor de un tipo específico, por ejemplo:

```
package main

import "fmt"

type Greeter interface {
    Greet(name string)
}

type str struct{}

func (str) Greet(name string) {
    fmt.Printf("Hi %v\n", name)
}

func main() {
    // s is of type str so it implements Greeter
```

```
    s := str{}
    s.Greet("gopher")
    extract(s)
}

func extract(v interface{}) {
    // if v holds a value of type str then greeter will
    // have that value. If it does not then this will triger
    // a panic
    greeter := v.(str)
    greeter.Greet("gopher1")

    // instead of triggering a panic we can use a second
    // variable that will hold a boolean specifying if v
    // is of type str
    gr, ok := v.(str)
    if !ok {
        fmt.Println("v is not of type str")
        return
    }
    gr.Greet("gopher2")
}
```

En el ejemplo anterior poner especial atención a la linea `gr, ok := v.(str)`. En java sería hacer un casting con un valor (ok) booleano que me indica si el casting fue exitoso o no.

Por otro lado también podemos hacer type assertion de múltiples valores usando switches. Por ejemplo:

```
package main

import "fmt"

type Greeter interface {
    Greet(name string)
}

type str struct{}

func (str) Greet(name string) {
    fmt.Printf("Hi %v\n", name)
}

func main() {
    // s is of type str so it implements Greeter
    s := str{}
    extract(s)
    extract(2)
    extract("hello")
    extract(2.4)
}
|            1
func extract(v interface{}) {
    switch t := v.(type) {
    case str:
        t.Greet("GOPHER")
    case int:
        fmt.Printf("sent an int: %v\n", t)
```

```
    case string:
        fmt.Printf("sent a string: %v\n", t)
    default:
        fmt.Printf("unsupported type: %T\n", t)
    }
}
```

Como podemos ver (y como dijimos antes), la sentencia switch toma no solo constantes o enumerados como otros lenguajes sino también valores y en este caso se puede ver la flexibilidad que esto otorga. En switch pregunta por el tipo y evita el panic o al menos el `if ok` :)

Errores

Los errores son motivo de discusiones epicas en la comunidad, principalmente porque es diferente a como se usan típicamente en otros lenguajes como java, ruby, etc. Si vos queres expresar un error en esos lenguajes podes crear tu propia exception y darle un nombre y detalles propios del error. Por esto vas a necesitar estructuras típicas como try-catch-finally para manipular errores. En golang los errores se manejan como cualquier otro valor. Por esto, un error es simplemente una interfaz con un método Error() que devuelve un string.. Si, as simple as that.

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

En go las funciones típicamente devuelven valores y error, de este modo al ejecutar una función o método se puede chequear si el error es nil o no. Esto implica que en cualquier función que estés codeando debes devolver lo que quieras devolver y por último un error y si la función ejecutó exitosamente simplemente devolves nil como error. Por ejemplo:

```
// Handling errors
err := callFunc()
if err != nil {
    // handle the error in here
}
```

Como podemos ver, luego de ejecutar una función preguntamos por si hubo error o no.

```
// Returning errors.
// Return the data you want + an error
func importantFunc() (string, error) {
    s, err := importantOperation()
    if err != nil {
```

```
        return "", err
    }
    return s, nil
}
```

Esta función `importantFunc` devuelve un string y en caso de error también un `error`. Fijense como devuelve nil en caso exitoso y cómo propaga el error de `importantOperation` en caso de `error`.

Concurrencia (yeah!)

Si habías escuchado algo de golang previamente seguramente escuchaste que golang tiene un gran native support para concurrencia. Es extremadamente simple y eficiente. Para esto golang utiliza el concepto llamado `goruotines` (go-rutinas). Las go rutinas es un lightweight thread manejado por Go Runtime (similar a los de java). Esto implica que en un thread del sistema operativo podemos tener múltiples go routines, miles si uno quisiera. Estas goroutines corren en el mismo espacio de direccionamiento lo cual implica que hay que ser muy cuidadoso con la modificación del estado (variables en memoria) cuando se usan múltiples gorutinas. El siguiente es un ejemplo del uso de go rutinas:

```
package main

import (
    "fmt"
    "time"
)

func importantFeature() {
    fmt.Println("doing important work")
}

func main() {
    importantFeature()

    // you can spawn new goroutines with a simple
    // go funcName
    go importantFeature() // this now created a brand new goroutine

    s := "inside goroutine"
    // you can also spawn goroutines using an
```

```
    // anonymous functions
    go func(someString string) {
        fmt.Println("printing from the goroutine")
    }(s)

    for i := 0; i < 5; i++ {
        // we can spawn as many as we like. Try changing
        // i<5 to something like i<1000
        go func(i int) {
            time.Sleep(5 * time.Millisecond)
            fmt.Printf("i=%v\n", i)
        }(i)
    }
    // wait a bit so that we don't exit immediately
    // we need to wait so that at least some of the
    // goroutines get executed. Go doesn't automatically
    // wait for all goroutines, if we don't do synchronization
    // then we will exit and goroutines won't finish executing.
    time.Sleep(25 * time.Millisecond)
}
```

Como pueden imaginar, cada gorutina genera un lightweight thread que corre como si fuera un spin off del thread principal, esto quiere decir que al terminar cada gorutina termina su thread (en Go Runtime) y es en paralelo al thread principal y de las otras gorutinas, por tal motivo está la última línea, time.Sleep(25 * time.Millisecond) me asegura que el thread principal no termine antes que sus gorutinas. Super simple.. :D

Channels

Hacer una gorutina fue extremadamente fácil.. Se puede decir que si tengo que paralelizar el trabajo puedo hacer simplemente una gorutina y listo. Ahora bien.. Para comunicar y sincronizar diferentes gorutinas existe el concepto de channel. Los channels son como tuberías entre las gorutinas, esto quiere decir que puedo enviar un valor desde una gorutina y recibirlo en otra. Por ejemplo:

```
package main

import (
    "fmt"
    "time"
```

```
)

func main() {
    messages := make(chan string)

    go func() {
        time.Sleep(1 * time.Second)
        // The information flows in the direction of the arrow
        // In this case we are sending data into the messages
        // channel
        messages <- "hi from the goroutine"
    }()

    // now we are receiving the information that is sent to
    // the channel.
    // Note that this will block for a second until the
    // channel has something we can receive.
    m := <-messages
    fmt.Println(m)
}
```

En el ejemplo se puede ver como creamos una gorutina que escribe en un channel un string y es luego recibido por el thread principal (main). Algunas particularidades, el channel lo creamos con make (y un tipo) y escribimos con o leemos con <- según donde lo pongamos (hacia o desde el channel).

Como se puede ver en los comments del código, la sentencia `m := <-messages` bloquea el current thread (main en este caso) hasta que `m` recibe un valor. Leer de un channel SIEMPRE va a bloquear el current thread mientras que escribir en un channel puede o no bloquear según como se inicialice el mismo. En el ejemplo previo declaramos el channel como synchronous, como hicimos eso?, simplemente al declarar el channel y no darle un size lo hicimos síncrono. Para hacerlo asynchronous tenemos que declarar un size al momento de crearlo, por ejemplo:

```
package main

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

```go
func main() {
    messages := make(chan string, 2)

    // we send two messages that will make
    // the channel full.
    messages <- "hi"
    messages <- "bye"

    // spawn a new goroutine that will read
    // from the channel after three seconds
    go func() {
        time.Sleep(3 * time.Second)
        fmt.Println(<-messages)
    }()

    // if we try to add a message to the channel
    // before one or all of the messages are consumed
    // then the operation will block since there
    // is no more space for that message to fit.
    // So after the previous go routine reads, we
    // will be able to send "hi again"
    messages <- "hi again"

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

Un channel con size se llama buffered channel. Como se puede ver en el código anterior, escribimos dos mensajes en el channel, luego creamos y ejecutamos una función en un hilo paralelo que lee uno de los dos mensajes, luego añadimos un tercer mensaje y por último leemos los dos que quedaban. Como podemos notar, el código no se bloquea en la lectura porque teniendo 1 mensaje sin leer puedo añadir otro.

Podríamos usar una gorutina y channels para esperar resultados y generar la sincronía para asegurarnos que ejecuta y devuelve algo.