

Introducción a Golang

Conociendo el lenguaje

Santiago Nicolás Risaro Sesar

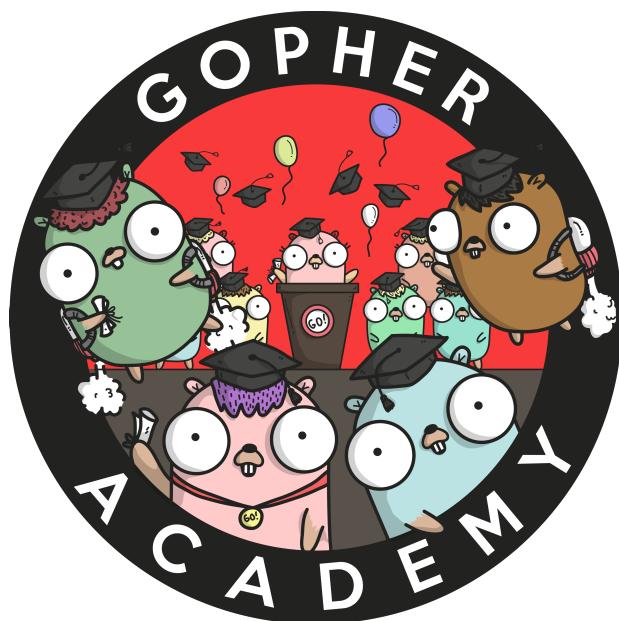
Las imágenes

<https://twitter.com/ashleymcnamara>

2

¿Quiénes crearon GO?

- Robert Griesemer, Ken Thompson y Rob Pike iniciaron el proyecto a fines de 2007
- A mediados de 2008 se finaliza el diseño y se termina la primera implementación



3

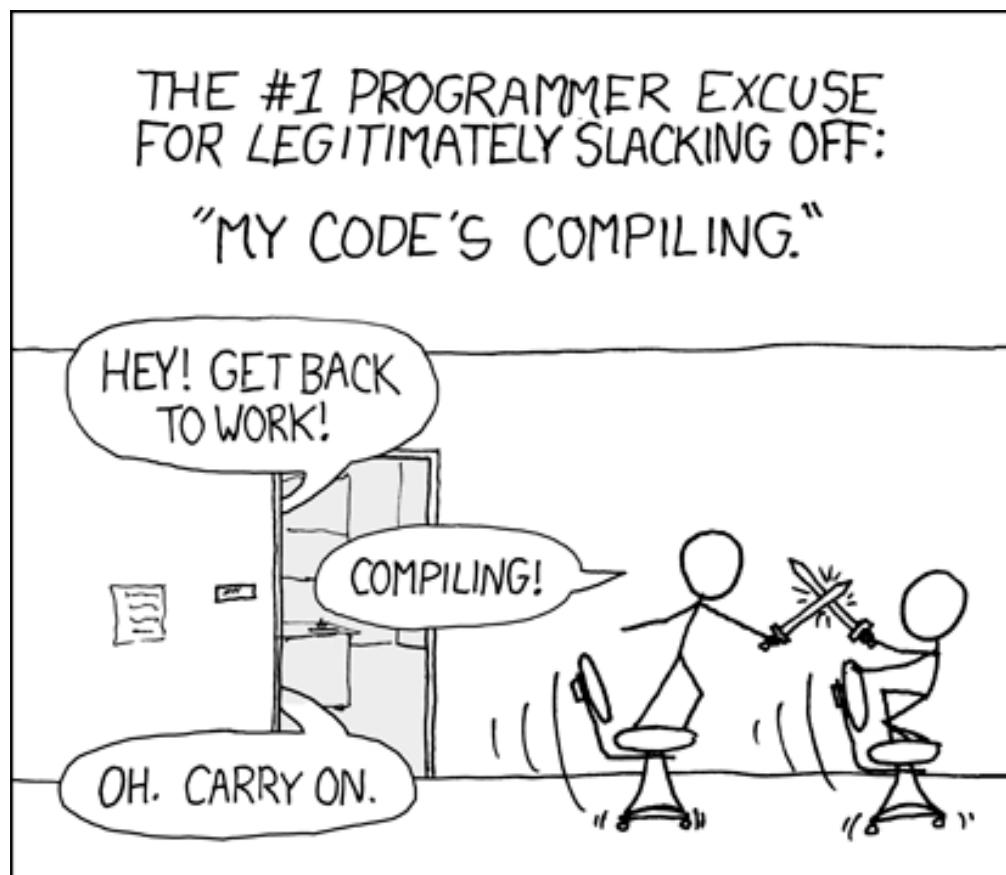
Filosofía y motivaciones

- Los sistemas cambiaron, los lenguajes no
- Velocidad de desarrollo
- Repensar el sistemas de tipos
- Agregar funcionalidad al lenguaje no lo hace mejor, sólo más grande

4

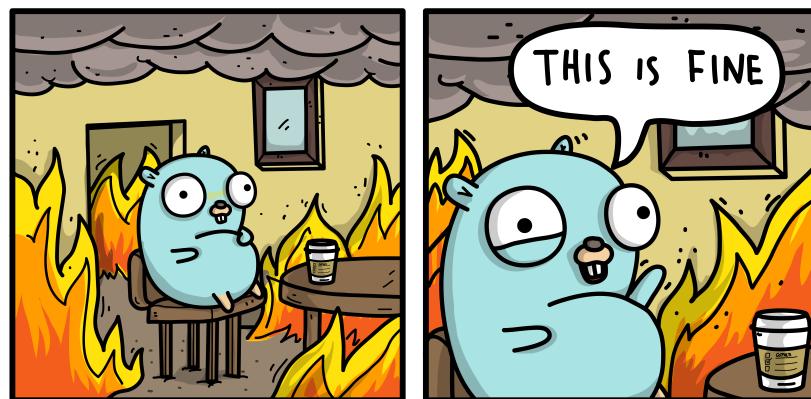
¿Por qué? Go fast!

"Make programming fun again" (Rob Pike)



Características principales

- Lenguaje procedural pero con funciones, métodos, interfaces y concurrencia
- La eficiencia de un lenguaje estáticamente tipado con la facilidad de un lenguaje dinámico
- Buen soporte para concurrencia y comunicación entre procesos
- Garbage collector eficiente y sin latencia
- Alta velocidad de compilación y ejecución
- Legibilidad y simplicidad



6

Entorno de desarrollo

- Descargar GO de golang.org/dl/ (<https://golang.org/dl/>)
- Descomprimirlo en /usr/local
- Editar .profile

```
# Define GOROOT
export GOROOT="/usr/local/go"

# add GOROOT to the PATH
PATH="$GOROOT/bin:$PATH"

# declare GOPATH
export GOPATH="$HOME/go"

# add GOPATH to the PATH
PATH="$GOPATH/bin:$PATH"
```

- Crear la carpeta \$GOPATH/src

```
nayla@Simona:~$ go version
go version go1.11 linux/amd64
```

Resultado de ejecutar go version

7

IDEs



Primer programa

```
package main

import "fmt"

func main() {
    fmt.Printf("Hola, mundo\n");
}
```

Run

9

Probar, compilar, ejecutar, instalar

```
# Compilar y ejecutar las pruebas que están en el directorio actual  
go test  
  
# Compilar <fuente>.go (El ejecutable queda en la carpeta en la que se encuentra el archivo .go)  
go build <fuente>.go  
  
# Compilar y ejecutar <fuente>.go  
go run <fuente>.go  
  
# Compilar e instalar el paquete (El ejecutable queda en $GOPATH/pkg/<paquete>)  
go install
```



10

Paquetes e imports

```
package main

import (
    "fmt"
    "math/rand"
)

func main() {
    fmt.Println("Mi número favorito es", rand.Intn(10))
}
```

- Todos los archivos de una carpeta deben pertenecer al mismo paquete
- El nombre del paquete no necesariamente debe coincidir con el nombre de la carpeta

11

Valores exportados

- Por defecto una variable global o función se define para todo un paquete (no para un fuente)
- Si quiero usar una variable o función fuera del paquete en el que fue definida su nombre debe empezar con mayúsculas
- Al importar un paquete sólo podés acceder a sus valores exportados

12

Inicialización de variables

```
package main

import "fmt"

var i, j int = 1, 2

func main() {
    var c, python, java = true, false, "no!"
    k := 3
    fmt.Println(i, j, c, python, java, k)
}
```

Run

13

Tipos básicos

- bool
- string
- int int8 int16 int32 int64
- uint uint8 uint16 uint32 uint64 uintptr
- byte // alias de uint8
- float32 float64
- complex64 complex128

14

Funciones

```
package main

import "fmt"

func Sumar(x int, y int) int {
    return x + y
}

func main() {
    fmt.Println(Sumar(42, 13))
}
```

Run

15

Funciones con múltiples valores de retorno

```
package main

import "fmt"

func intercambiar(x, y string) (string, string) {
    return y, x
}

func main() {
    a, b := intercambiar("Grupo", "Esfera")
    fmt.Println(a, b)
}
```

Run

16

Identificador vacío

Si no nos interesa alguno de los valores que retorna una función, se usa el underscore para ignorarlo

```
_ , b := intercambiar("Grupo", "Esfera")
```



17

Errores

En GO no existen las excepciones

```
i, err := strconv.Atoi("42")  
  
if err != nil {  
    fmt.Printf("no se puede convertir el numero: %v\n", err)  
    return  
}
```

Para crear un error usamos `fmt.Error()` o `errors.New()`

```
if divisor == 0 {  
    return fmt.Errorf("no se puede usar %v como divisor", divisor)  
}
```

18

If else

```
if x < 0 {  
    return x + 1  
} else {  
    return x - 1  
}
```

El if puede comenzar con una sentencia corta que se ejecuta antes de la condición

```
if v := math.Pow(x, n); v < lim {  
    return v  
}
```

19

Switch

```
switch os := runtime.GOOS; os {  
    case "darwin":  
        fmt.Println("OS X.")  
    case "linux":  
        fmt.Println("Linux.")  
    default:  
        fmt.Printf("%s.", os)  
}
```

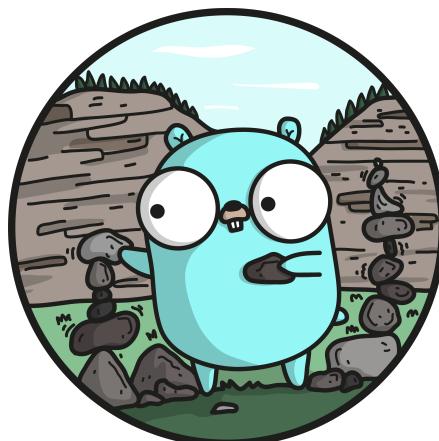
20

For

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

La inicialización y el incremento son opcionales

```
for ; sum < 1000; {  
    sum += sum  
}
```



21

While

```
for sum < 1000 {  
    sum += sum  
}
```

Ciclo infinito

```
for {  
}
```

22

Arrays

- El tipo $[n]T$ es un array de n valores de tipo T
- La longitud de un array es parte de su tipo, por eso, no pueden redimensionarse

```
var enteros [10]int
```

```
package main

import "fmt"

func main() {
    var a [2]string
    a[0] = "hola"
    a[1] = "mundo"
    fmt.Println(a[0], a[1])
    fmt.Println(a)

    primos := [6]int{2, 3, 5, 7, 11, 13}
    fmt.Println(primos)
}
```

Run

23

Slices

- El tipo `[]T` es un slice con elementos de tipo `T`
- Un slice se redimensiona dinámicamente. En la práctica son más comunes que los arrays
- El zero value de un slice es `nil`

```
func main() {  
  
    var s []int  
    fmt.Println(s)  
  
    if s == nil {  
        fmt.Println("nil!")  
    }  
  
    a := make([]int, 0) // Crea un slice vacío  
    fmt.Println(a)  
  
    b := make([]int, 5) // Crea un slice con 5 enteros (zero value)  
    fmt.Println(b)  
}
```

Run

24

Slices

Un slice es un puntero a un array, modificar un slice implica modificar el array y viceversa

```
package main

import "fmt"

func main() {

    nombres := [4]string{"John", "Paul", "George", "Ringo",}

    fmt.Println(nombres)

    a := nombres[0:2] // Creo un slice con los 2 primeros nombres
    b := nombres[1:3] // Creo un slice con el segundo y tercer no
    fmt.Println(a, b)

    b[0] = "XXX" // Modifico el primer nombre del slice b
    fmt.Println(a, b)
    fmt.Println(nombres)

}
```

Run

25

Agregar elementos a un slice

```
package main

import "fmt"

func main() {
    var s []int
    imprimirSlice(s)

    s = append(s, 0)
    imprimirSlice(s)

    s = append(s, 1)
    imprimirSlice(s)

    s = append(s, 2, 3, 4)
    imprimirSlice(s)
}

func imprimirSlice(s []int) {
    fmt.Printf("len=%d cap=%d %v\n", len(s), cap(s), s)
}
```

Run

26

Recorriendo un slice

```
package main

import "fmt"

var pow = []int{1, 2, 4, 8, 16, 32, 64, 128}

func main() {

    for indice, valor := range pow {
        fmt.Printf("2**%d = %d\n", indice, valor)
    }

    for _, valor := range pow {
        fmt.Printf("%d\n", valor)
    }
}
```

Run

27

Maps

```
package main

import "fmt"

var empresas map[string]int

func main() {
    empresas = make(map[string]int)

    empresas["Grupo Esfera"] = 40
    empresas["Mercado Libre"] = 40000

    fmt.Println(empresas)
    fmt.Println(empresas["Grupo Esfera"])
}
```

Run

28

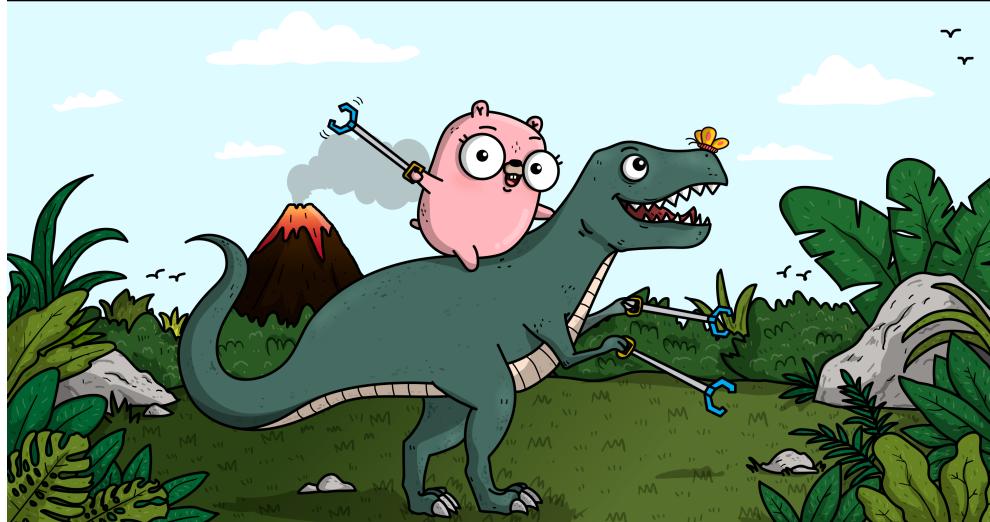
Operaciones sobre maps

Se pueden eliminar elementos de un mapa con `delete`

```
delete(mapa, clave) // si la clave no existe, la sentencia no fal
```

Se puede verificar si existe un elemento en un mapa

```
elemento, existe = mapa[clave]
```



29

Recorriendo un map

```
package main

import "fmt"

type Vertice struct {
    Lat, Long float64
}

var m = map[string]Vertice{
    "Bell Labs": Vertice{
        40.68433, -74.39967,
    },
    "Google": Vertice{
        37.42202, -122.08408,
    },
}

func main() {
    for clave, valor := range(m) {
        fmt.Println(clave, valor)
    }
}
```

Run

30

Tipos y métodos

- En GO no tenemos clases
- Si queremos agrupar datos y funciones podemos utilizar tipos
- Una función asociada a un tipo es un método, en este contexto al tipo lo llamamos receiver

```
type Vertice struct {  
    X, Y float64  
}  
  
func (vertice Vertice) Abs() float64 {  
    return math.Sqrt(vertice.X*vertice.X + vertice.Y*vertice.Y)  
}
```

- Invocamos un método con .

```
v := Vertice{3, 4}  
fmt.Println(v.Abs())
```

Pointer receivers

Si queremos que un método modifique los valores de su receiver usamos punteros

```
type Vertice struct {
    X, Y float64
}

func (v Vertice) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func (v *Vertice) Agrandar(f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}

func main() {
    v := Vertice{3, 4}
    fmt.Println(v.Abs())
    v.Agrandar(10)
    fmt.Println(v.Abs())
}
```

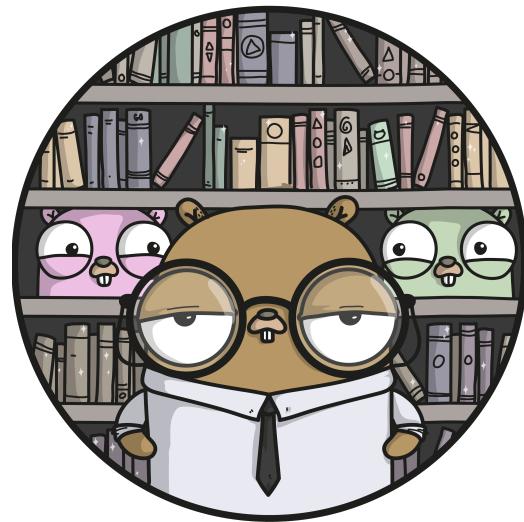
Run

32

Pointer receivers

No necesito un puntero para llamar a un método definido con un pointer receiver

```
var v Vertice
v.Agrandar(5) // OK
v.Abs()        // OK
p := &v
p.Abs()        // OK
p.Agrandar(10) // OK
```



33

Interfaces

Una interface es un conjunto de firmas de métodos

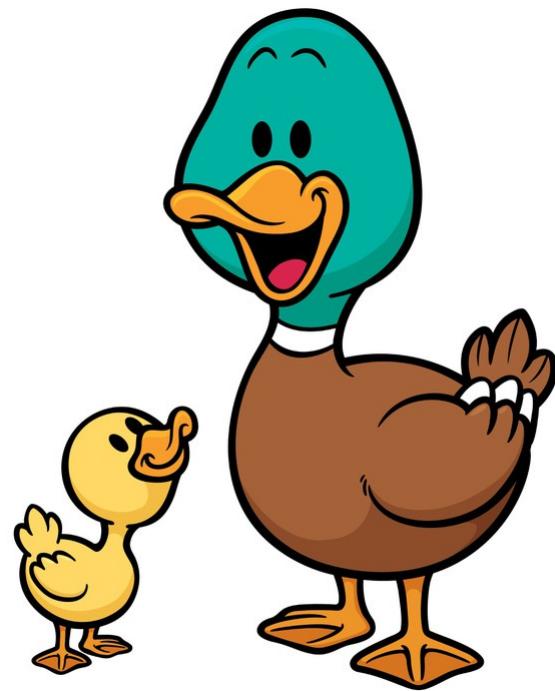
El paquete `fmt` define la interface `Stringer`, los tipos que la implementan se pueden convertir a string

```
type Stringer interface {
    String() string
}
```

34

Interfaces

Si camina como pato, nada como pato y hace cuak como un pato... es una implementación de la interfaz pato



35

Interfaces

Si quiero hacer referencia a cualquier tipo utilizo la interfaz vacía

```
var i interface{}
```

Si quiero saber si una variable es de un tipo determinado uso type assertions

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

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

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

Polimorfismo

```
type Animal interface {
    EmitirSonido() string
}

func (p *Perro) EmitirSonido() string {
    return "guau"
}

func (g *Gato) EmitirSonido() string {
    return "miau"
}

func main() {
    var animales []Animal
    gato := &Gato{}
    perro := &Perro{}
    animales = append(animales, gato, perro)

    for _,animal := range(animales) {
        fmt.Println(animal.EmitirSonido())
    }
}
```

Run

37

Estructuras embebidas

En GO no hay herencia, pero tenemos composición++

Una estructura puede tener a otra embebida, y poseer todas sus cualidades

```
type Motor struct {
    Cilindros int
}
type Auto struct {
    Motor
    Marca string
}
func (m Motor) Sonar() {
    fmt.Printf("Mis %v cilindros hacen poco ruido\n", m.Cilindros)
}
func (a Auto) Andar() {
    fmt.Printf("Con mis %v cilindros ando bien rapido\n", a.Cilindros)
}
func main() {
    fitito := Auto{Motor: Motor{Cilindros: 4}, Marca: "Fiat",}

    fitito.Sonar()
    fitito.Andar()
}
```

Run

38

Testing

Un test tiene que estar en un archivo cuyo nombre finalice en `_test.go` (i.e.: `auto_test.go`)

En general el package también finaliza con `_test`

Un test debe empezar con la palabra `Test` y debe recibir por parámetro un puntero a `testing.T`

```
package auto_test

import (
    "testing"

    "github.com/nickrisaro/workshop-go/auto"
)

func TestUnAutoTieneCilindros(t *testing.T) {
    fitito := auto.Auto{Motor: Motor{Cilindros: 4}, Marca: "Fiat"}

    if fitito.Cilindros() != 4 {
        t.Errorf("Esperaba 4 cilindros pero encontré %d", fitito.
            return
    }
}
```

39

Testing un poco más bonito

Con el framework default de testing hay que hacer muchas cosas a mano.

Testify al rescate github.com/stretchr/testify (<https://github.com/stretchr/testify>)

/testify)

```
assert.Equal(t, cilindrosEsperados, cilindrosObtenidos, "No se ob  
assert.NotNil(t, auto, "Esperaba que el auto no sea nulo")  
assert.Contains(t, lista, unElemento, "No se encontró el elemento")
```



40

Hablemos de dependencias

- Dependencias globales --> go get
- Dependencias locales --> vendoring (govendor, dep, y otras)
- Módulos, la nueva opción



41

Hilos y GO

La primera regla de la concurrencia: no usen concurrencia

Concurrencia no es paralelismo (Si tenés un solo procesador tu programa puede tener concurrencia pero no paralelismo)



42

GO Routines

Una goroutine es un hilo liviano administrado por GO

```
go f(x, y, z)
```

La sentencia go inicia una nueva rutina que corre concurrentemente con la ejecución principal

NO ES UN HILO

El runtime se encarga de crear y administrar los hilos necesarios para la ejecución de las rutinas.

Las rutinas de go corren en el mismo espacio de direcciones, por lo que el acceso a recursos compartidos debe ser sincronizado

43

GO Routines

```
package main

import (
    "fmt"
    "time"
)

func say(s *string) {
    for i := 0; i < 5; i++ {
        time.Sleep(100 * time.Millisecond)
        fmt.Println(*s)
    }
}

func main() {
    mensaje := "mundo"
    go say(&mensaje)

    mensaje2 := "hola"
    say(&mensaje2)
}
```

Run

44

Channels

Son conductos tipados de comunicación entre rutinas

Se envía o recibe información por un channel con el operador de canal <-

```
ch := make(chan int)

ch <- v    // Envía `v` al canal `ch`
v := <-ch  // Recibe del canal `ch` y asigna valor en `v`
```

Los canales son sincronizados: tanto el escribir como leer mensajes bloquean la ejecución de la rutina hasta que se realice la operación opuesta.

Ambas partes deben estar listas para continuar con la ejecución

45

Channels

```
func generaMensajes(c chan string) {
    for i := 0 ; ; i++ {
        c <- fmt.Sprintf("Mensaje generado %d", i)
        time.Sleep(1 * time.Second)
    }
}

func main() {
    c := make(chan string)
    go generaMensajes(c)

    for i := 0; i < 5; i++ {
        fmt.Println("Mensaje recibido: ", <-c)
    }

    fmt.Printf("Dejé de escuchar")
}
```

Run

46

Channels y buffers

Los canales pueden tener un buffer de mensajes. Tener un buffer evita que la escritura bloquee la ejecución

Cuando el buffer se llena, la ejecución bloquea al intentar agregar algo al canal

```
func main() {
    ch := make(chan int, 2)
    ch <- 1
    ch <- 2
    //ch <- 3

    fmt.Println(<-ch)
    fmt.Println(<-ch)
    //fmt.Println(<-ch)

    fmt.Println("FIN")
}
```

Run

47

Range y Close

Para leer infinitamente de un canal se utiliza el operador range

```
for v := range ch
```

Los canales se pueden cerrar con la función close(ch). Si el canal es cerrado, el range finaliza

Quien lee de un canal sabe que fue cerrado utilizando la forma

```
v, ok := <-ch
```

No se puede leer de un canal cerrado.

48

Thank you

Santiago Nicolás Risaro Sesar

[@NickRisaro](http://twitter.com/NickRisaro) (<http://twitter.com/NickRisaro>)

Desarrollador en Grupo Esfera

[@grupoesfera](http://twitter.com/grupoesfera) (<http://twitter.com/grupoesfera>)