FACULTAD DE INGENIERIA
Universidad de Buenos Aires

# Lenguaje Go

## Teoría de Lenguaje - 75.31

Adrián Mouly - Sebastián Torres

# Temario

- Introducción
- Paquetes y Variables
- Funciones
- Estructuras de control
- Tipos
- Objetos
- Concurrencia
- Manejo de memoria
- AppEngine y Compiladores
- Benchmarks
- Para qué sirve (y qué no)
- Ejemplos de uso

# Introducción

# Introducción


El lenguaje nace en 2007 en manos de Rob Pike, pero no toma importancia hasta 2009 cuando es impulsado por **Google**.

En la última década la informática cambió mucho:

- Gran importancia del Networking
- Computación en grandes clusters
- Servicios orientados a cliente/servidor
- Masificación de multi-core CPU

Pese a este avance de los sistemas informáticos, no existía un lenguaje diseñado con estos aspectos en mente. **Go** pretende ocupar ese lugar.

# Introducción


La fabricación de software actualmente presenta algunas dificultades:

- Lleva mucho tiempo compilar un programa
- Las herramientas de producción son cada vez más lentas
- Creciente cantidad de librerías y dependencias
- La complejidad de los programas aumenta con el tiempo

Algunos de los objetivos de **Go** son:

- Aumentar la productividad
- Rápida compilación
- Eficiencia y recolección de basura de baja latencia
- Buen soporte para concurrencia y comunicaciones
- Seguridad

# Introducción


Gramática simple

Pretende reducir el tipeo

- Dejar que el lenguaje realice más trabajo
- Sintaxis limpia, muy similar a C

Sistema de tipado simple y estático

- Tiene algunas características de lenguajes con tipado dinámico

Separar la *interfaz* de la implementación

Sistema de paquetes y dependencias

- El uso de paquetes es explícito y sólo se utiliza lo necesario

# Introducción


Es un lenguaje multi-paradigma.

**Go** utiliza interfaces y métodos, pero no es totalmente orientado a objetos ya que no existe la jerarquía de tipos, ni las clases.

Toma conceptos de distintos paradigmas y utiliza lo mejor de ellos.

No incluye soporte para:

- Herencia de tipos
- Sobrecarga de métodos
- Aritmética de punteros
- Manejo de excepciones

# Paquetes y Variables

# Paquetes


Todo programa hecho en **Go**, esta formado por paquetes.

Una aplicación comienza a ejecutarse desde el paquete "main".

Por convención, el paquete lleva el nombre de la función principal.

```
package main

import "fmt"

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

# Variables


Las variables son declaradas con *var*, aclarando el tipo al final.

Al declarar variables, estas pueden ser inicializadas. En este caso se puede omitir el tipo.

```
package main

import "fmt"

var x, y, z int
var c, python, java bool

func main() {
    fmt.Println(x, y, z, c, python, java)
}
```

# Ámbito de una variable


El ámbito de una variable define la visibilidad de la misma para otras funciones, métodos, o incluso paquetes.

Dentro de un paquete todas las variables globales, funciones, tipos y constantes son visibles desde todos los ficheros que formen dicho paquete.

Para los usuarios de dicho paquete, los nombres deben comenzar por una letra mayúscula para que sean visibles.

```
var prueba = "Una Prueba" //Variable visible en el paquete
var Prueba = "Otra Prueba" //Variable visible globalmente
```

# Funciones

# Funciones


Una función puede recibir cero o más parámetros.

Cuando dos o más parámetros consecutivos comparten un tipo, este se puede omitir.

Pueden retornar múltiples resultados.

```
func suma(x int, y int) int {
    return x + y
}
```

Ver "ejemplo-funciones.go"

# Funciones


Si la función devuelve varios valores, en la declaración de la función los tipos de los valores devueltos van en una lista separada por comas entre paréntesis.

```
func MySqrt(f float) (float, bool) {
    if f >= 0 {
        return math.Sqrt(f), true
    }
    return 0, false
}

func MySqrt2(f float) (v float, ok bool) {
    if f >= 0 {
        v, ok = math.Sqrt(f), true
    }
    return     // Retorna v,ok con sus valores
}
```

# Funciones Anónimas y Closures


Son típicas de lenguajes funcionales.

**Funciones anónimas**: Están definidas en tiempo de ejecución. Se ejecutan en el contexto en el que son declaradas. A menudo, son referenciadas por variable para poder ser ejecutadas en otro contexto.

**Closures**: Se pueden definir como funciones anónimas declaradas en el ámbito de otra función y que pueden hacer referencia a las variables de su función contenedora incluso después de que ésta se haya terminado de ejecutar.

# Funciones

```
func sumador(){
    var x int
    return func(delta int) int {
                x += delta
                return x
            }
}

var f = sumador() // f es una función ahora.
fmt.Printf(f(1))
fmt.Printf(f(20))
fmt.Printf(f(300))
```

Imprime los valores 1, 21 y 321, ya que irá acumulando los valores en la variable x de la función f.

# Funciones


**Alto orden**

**Go** soporta pasaje de funciones por parámetro

```
func printCool(){
    fmt.Println(" cool !")
}

func twice(foo func()){
    foo()
    foo()
}

func main(){
    twice(printCool)
}
```

# Estructuras de control

# If / Else


Admite instrucciones de inicialización, separadas por un punto y coma.

Si una sentencia condicional *if* no tiene ninguna condición, significa true.

```
if x < 5 { f() }

if x < 5 { f() } else if x == 5 { g() }

if v := f(); v < 10 {
    fmt.Printf("\%d es menor que 10\n", v)
} else {
    fmt.Printf("\%d no es menor que 10", v)
}
```

# For


**Go** sólo posee esta estructura de loop

Sintaxis similar a C pero simplificada

```
package main

import "fmt"

func main() {
    sum := 0

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

    fmt.Println(sum)
}
```

# Switch


En apariencia es similar al de C, pero tiene varias diferencias:

```
switch count {
    case 4, 5, 6: error()
    case 3: a *= v; fallthrough
    case 2: a *= v; fallthrough
    case 1: a *= v; fallthrough
    case 0: return a*v
    default: fmt.Printf("No es ninguno")
}
```

# Switch


También acepta expresiones lógicas:

```
// If-else
a, b := 3, 5;

// Un switch vacío significa true
switch {
    case a < b: return -1
    case a == b: return 0
    case a > b: return 1
}

// Este último switch sería equivalente a
switch a, b := 3, 5; { ... }
```

# Tipos

# Tipos Básicos


## Tipos numéricos

```
int  int8  int16  int32  int64
uint uint8 uint16 uint32 uint64 // Enteros sin signo

byte // Alias de uint8

rune // Alias de int32

float32 float64

complex64 complex128 // Números complejos
```

## Strings

El tipo string representa arrays invariables de bytes (texto). Los strings están delimitados por su longitud, no por un carácter nulo como suele ocurrir en la mayoría de lenguajes.

# Structs


Un *struct* es una colección de campos

```
package main

import "fmt"

type Vector struct {
    X int
    Y int
}

func main() {
    fmt.Println(Vector{1, 2})
    Vector.X = 4

    fmt.Println(Vector.X)
}
```

# Arrays


Los arrays de **Go** son más cercanos a los de Pascal que a los de C.

```
var ar [3]int
```

Si en algún momento deseamos averiguar el tamaño de un array, podemos hacer uso de la función len(). Los arrays son valores, no punteros implícitos como ocurre en C.

```
// Array de 10 enteros, los 3 primeros no nulos
[10]int { 1, 2, 3 }

// Si no queremos contar el número de elementos
[...]int { 1, 2, 3 }

// Si no se quiere inicializar todos,'clave:valor'
[10]int { 2:1, 3:1, 5:1, 7:1 }
```

# Slices


Un slice es una referencia a una sección de un array. (Más parecido a C)

```
var a []int
```

Un slice puede crearse “troceando” un array u otro slice.

Esta instrucción nos generaría un slice a partir de un array, tomando los índices 7 y 8 del array.

```
a = ar[7:9]
```

Podemos inicializar un slice asignándole un puntero a un array:

```
a = &ar // Igual que a = ar[0:len(ar)] o a = ar[0:]
```

# Slices


Podemos también reservar memoria para un slice (y su array correspondiente) con la función predefinida make():

```
var s100 = make([]int, 100) // Slice de 100 enteros
var sl = make([]int, 0, 100) // Len == 0, cap == 100
```

Un slice se refiere a un array asociado, con lo que puede haber elementos más allá del final de un slice que estén presentes en el array. La función cap() nos indica el número de elementos que el slice puede crecer.

```
var ar = [10]int {0,1,2,3,4,5,6,7,8,9}
var a = &ar[5:7] // Referencia al subarray {5, 6}
// len(a) == 2 y cap(a) == 5.
a = a[0:4] // Referencia al subarray {5,6,7,8}
// len(a) == 4. cap(a) == 5.
```

# Maps

Un *map* mapea claves a valores

```
type Vector struct {
    Lat, Long float64
}

var m map[string]Vector

func main() {
    m = make(map[string]Vector)

    m["FIUBA"] = Vector{ -34.617639, -58.368497 }

    fmt.Println(m["FIUBA"])
}
```

# Maps


Insertar elementos

```
m[clave] = elemento
```

Obtener un elemento

```
elemento = m[clave]
```

Borrar elemento

```
delete(m, clave)
```

# Objetos

# Objetos


En este apartado, surge una discusión. Existen distintos consensos sobre la teoría de "orientación a objetos". Si consideramos a los objetos como ententes individuales, con la capacidad de intercambiar información entre ellos, podríamos decir que **Go** presenta estas características a través de los canales y corutinas. En cambio, si consideramos la teoría tradicional de objetos, donde existen mecanismos de clases, herencia, interfaces, entonces **Go** no es orientado a objetos.

Podríamos decir que Go permite programar basándose en objetos, pero de una forma no convencional, o no como la realizan otros lenguajes.

# Métodos


**Go** no posee clases, pero se pueden crear métodos específicos para un tipo de datos concreto, los cuales deben estar definidos dentro del mismo paquete.

Los métodos en **Go** se declaran de forma independiente de la declaración del tipo.

```
type Punto struct { x, y float }

//Un método sobre *Punto
func (p *Punto) Abs() float {
    return math.Sqrt(p.x*p.x + p.y*p.y)
}

p := &Punto { 3, 4 }
fmt.Print(p.Abs()) // Imprimirá 5
```

# Métodos


**Go** automáticamente indirecciona o derreferencia los valores cuando se invoca un método.

```
    p1 := Punto { 3, 4 }
    fmt.Print(p1.Abs()) // Azúcar sintáctico para (&p1).
Abs()
```

Se puede emular el mecanismo de herencia de métodos, esto ocurre cuando tenemos atributos anónimos dentro de un struct:

```
type Punto struct { x, y float }
func (p *Punto) Abs() float { ... }
type PuntoNombre struct{
    Punto
    nombre string
}
n := &PuntoNombre { Punto { 3, 4 }, "Pitágoras" }
fmt.Println(n.Abs()) // Imprime 5
```

# Interfaces


**Definición:** Una interfaz es algo completamente abstracto y que por sí solo no implementa nada.

Son parecidas a las interfaces de Java, pero con bastantes diferencias.

Una interfaz es un conjunto de métodos.

**Tipo Interface:** Un tipo interface es una especificación de una interfaz, es decir, un conjunto de métodos que son implementados por otros tipos.

```
type AbsInterface interface {
    Abs() float // El receptor es implícito
}
```

El tipo Punto implementa AbsInterface pues implementa el método Abs().

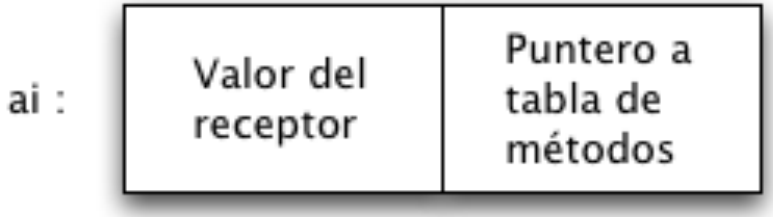
# Interfaces


**El valor interfaz:** Una vez una variable es declarada con un tipo interface, puede almacenar cualquier valor que implemente dicha interfaz.

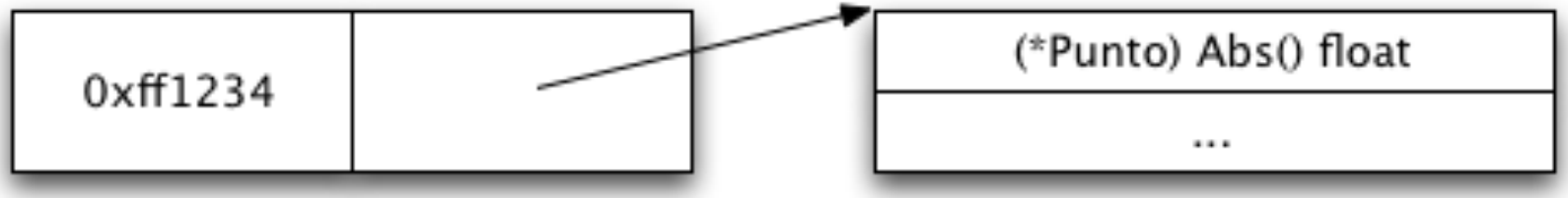
```
var ai AbsInterface
pp := new(Punto)
ai = pp // OK: *Punto tiene Abs()
ai = 7. // Error en tiempo de compilación:
          float no tiene Abs()
```

# Interfaces


La variable **ai** no es un puntero, sino que es una estructura de datos de varias palabras:




```
ai = &Punto { 1, 2 }
```



# Interfaces


**Contenedores y la interfaz vacía**

Los contenedores son estructuras de datos que permiten almacenar elementos de cualquier tipo o de un determinado tipo.

```
type Element interface {}

//Vector es el propio contenedor
type Vector struct {
    a []Element
}

// At() devuelve el i-ésimo elemento
func (p *Vector) At(i int) Element {
    return p.a[i]
}
```

# Concurrencia

# Concurrencia


La concurrencia y la programación multi-hilos presenta algunas dificultades a la hora de programar o diseñar un programa.

**Go** trata de simplificar el uso de concurrencia, ya que lo considera indispensable para los programas actuales.

Se utiliza el concepto de *goroutine*, abstrayendo la utilización de *threads*, de los cuales se ocupa el *run-time system* de forma eficiente.

Una *goroutine* es una función corriendo independientemente en el mismo espacio de memoria de otras *gouroutines*.

# Concurrencia


El *run-time system* se encarga de separar las rutinas en ejecución en distintos hilos.

> Cuando una rutina que esta siendo ejecutada se bloquea, el sistema automáticamente mueve la rutina siguiente a otro hilo, de esta forma evita que se paralice la ejecución del programa.

Los hilos no comparten memoria para comunicarse, como hacen otros lenguajes. El proceso es inverso: para compartir información entre procesos, estos deben comunicarse.

> La comunicación (intercambio de valores) se realiza mediante canales.

Este concepto evita que la información se pierda o sea sobreescrita, aunque genera una carga mayor de memoria.

# Canales


Los canales son un conducto por el cual se puede enviar y recibir valores usando el operador canal (<-).

```
ch := make(chan int) // Crea un canal de enteros
ch <- v // Envía información al canal
w := <- ch // Recibe del canal y asigna el valor
```

Un canal puede tener un buffer.

Los envíos en un canal con buffer solo se bloquean si el buffer está lleno.

Las recepciones se bloquean cuando el buffer está vacío.

```
ch := make(chan int, 100)
```

# Manejo de Memoria

# Manejo de Memoria


**Go** no posee aritmética de punteros, según los creadores, por razones de seguridad. Sin aritmética de punteros no es posible obtener una dirección de memoria ilegal y que sea utilizada de forma incorrecta.

Por otro lado, la falta de aritmética de punteros simplifica la implementación del recolector de basura.

# Recolección de Basura


Actualmente, el compilador de **Go** posee un Garbage Collector muy simple pero efectivo basado en el ”marcado de barrido” (mark and sweep), es decir, marca aquello que puede ser eliminado, y al ser activado, se borra.

Este recolector, se basa en los siguientes conceptos fundamentales:

**Stop the World**: El recolector detiene la ejecución de la aplicación para realizar su trabajo. La frecuencia de estas pausas es proporcional al tamaño del *heap*.

**Non-Compacting**: El recolector nunca mueve un objeto dentro de la memoria. Una vez que el objeto es creado en una dirección determinada, este persiste hasta que es liberado. Esta característica no permite compactar la memoria utilizada por una aplicación.

# Recolección de Basura


**Non-Generational**: Otros recolectores separan los objetos en Jóvenes y Viejos. Pero el recolector de Go los considera a todos por igual, no diferencia entre objetos recién creados u objetos que llevan más tiempo ejecutándose.

**Conservativo**: Un recolector de basura conservativo asume que cualquier dato puede ser un puntero. De esta forma el recolector realiza una búsqueda en memoria de un dato que pueda parecer un puntero y realiza una copia de dicho puntero en una zona de memoria donde impedir que sea reciclado. Este proceso puede producir un falso/positivo ya que el puntero puede apuntar a un objeto que ya fue borrado previamente, y de todas formas el puntero persiste.

# AppEngine y Compiladores

# AppEngine


**AppEngine** es una plataforma creada por Google, orientada a crear servicios web y distribuirlas en base a la infraestructura de Google.

Al ser un sistema pensado para la *nube*, provee acceso a todos los servicios de Google mediante APIs (*Datastore*, *Mail*, *Users*, *XMPP*) y sobre todo provee un entorno muy seguro.

> Las aplicaciones corren en un *sandbox* y se encuentran distribuidas en múltiples data-centers.

No es un framework, sinó que es un conjunto de frameworks, librerías y utilidades, con soporte para **Go**, Python y Java.

# Compiladores


Existen dos compiladores de **Go**:

**6g/8g/5g** - Compilan a AMD64 / x86 / ARM respectivamente. Comandos:

"go run ejemplo.go": Compila y ejecuta el programa

"go build ejemplo.go": Compila el programa y dependencias y genera un ejecutable

"go get paquete": Permite descargar paquetes y agregarlos al Runtime

**gccgo** - Frontend para GCC escrito en C++.

- Se utiliza de la misma forma que el compilador de C

Ambos compiladores son multiplataforma.

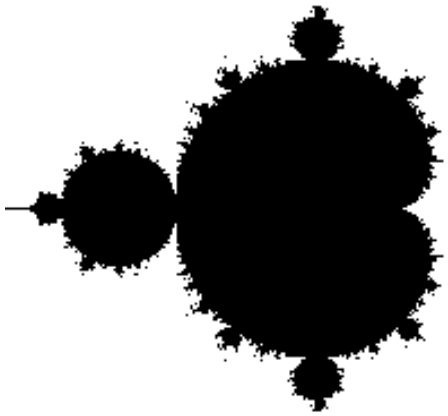
# Benchmarks

# Benchmarks


**Go** se caracteriza por estar diseñado para compilar y ejecutar sus programas de forma eficiente y veloz

Uno de los benchmarks tomados como referencia para medir la performance de un lenguaje, es el cálculo de los fractales de Mandelbrot

Lo que debe hacer el programa de pruebas, es graficar la imagen del conjunto [-1.5-i,0.5+i] y escribirlo a un archivo externo



# Benchmarks


Tabla comparativa (Intel® Q6600® one core)

| Lenguaje | CPU (seg) | Memoria (KB) | Código (B) | Carga CPU |
|---|---|---|---|---|
| **Go** | 129.03 | 31,412 | 700 | 100% |
| Lua | 794.37 | 1,044 | 353 | 100% |
| Scala | 46.38 | 67,500 | 796 | 100% |
| Ruby 1.9 | 3,944.74 | 3,400 | 307 | 100% |
| Mozart/Oz | 2,164.90 | 6,876 | 559 | 100% |
| Haskell | 58.61 | 36,148 | 782 | 100% |
| C | 23.31 | 29,920 | 799 | 100% |
| Fortran Intel | 14.68 | 30,936 | 967 | 100% |

# Para qué sirve
## (y qué no)

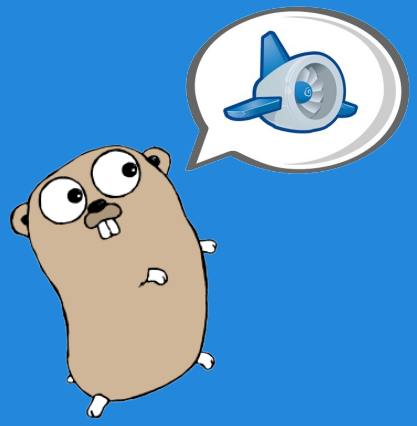
# Para qué sirve (y qué no)


**Go** es muy eficiente a la hora de realizar aplicaciones modernas, orientadas a los servicios web, donde existe un ambiente de constante cambio.

Sirve para crear soluciones en poco tiempo, ya que su sintaxis busca justamente eso.

No sirve para programar aplicaciones de bajo nivel, ya que no provee las estructuras necesarias.

# Ejemplo de uso



# Gracias

**Adrián Mouly | adrian@mouly.com.ar**
**Sebastián Torres | sebasmil986@gmail.com**

**Repo: http://code.google.com/p/amouly-codes/**