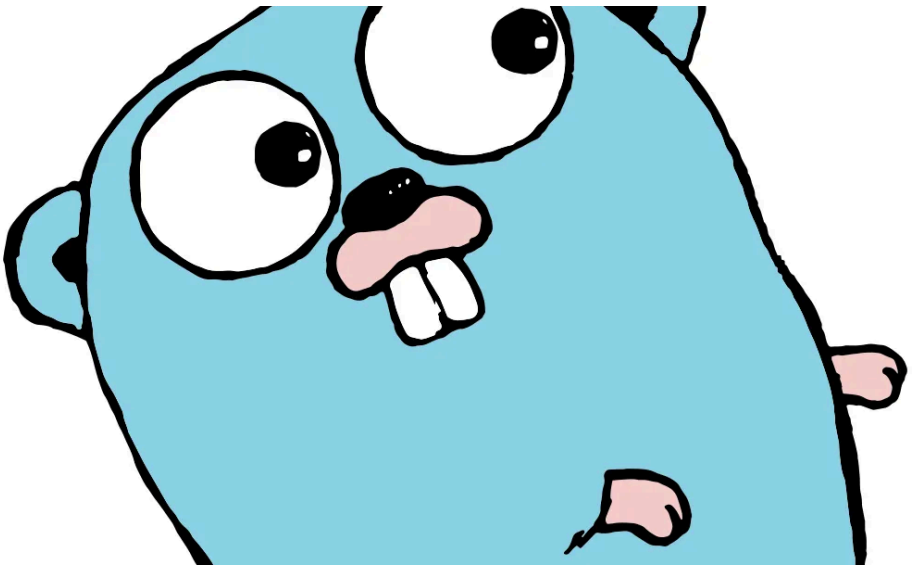


Go!

- clase 1 - lenguaje Go! → conceptos básicos
  - qué es Go?
    - lenguaje de programación multiplataforma de código abierto.
    - compilado, fuertemente tipado.
    - sintaxis basada en C/C++.
    - desarrollado por Google en 2007.
  - para qué se utiliza Go?
    - desarrollo web.
    - desarrollo de aplicaciones en red.
    - desarrollo de aplicaciones multiplataforma.
    - desarrollo nativo de la nube.
    - desarrollo de aplicaciones concurrentes.
  - por qué usar Go?
    - es fácil de aprender.
    - tiene tiempos de ejecución y compilación rápidos.
    - admite la concurrencia.

- admite genéricos.
- tiene administración de memoria.
- es portable a diferentes plataformas (windows, mac, linux, etc)

▼ estructura de un programa

```
package main
import "fmt"
func main {
   fmt.Println("Hello world")
}
```

▼ variables

```
// se declaran con la palabra clave var
var i int
var c,d,e bool

// se puede declarar e inicializar
var i int = 1

// Si hay una inicialización, el tipo puede ser “inferido” de los valores:
var c, d, e = true, false, "Texto"

// El tipo puede ser “inferido” usando “:= ” (short assignment):
k := 3
```

▼ tipos básicos

- bool
- string
- int, int8, int16, int32 int64, uint, uint8, uint16, uint32, uint64, uintptr: valor por defecto: 0. siempre usar int excepto que haya una razón específica para usar otro.
- byte (alias de uint8)
- rune (alias de int32)
- float32, float64: valor por defecto: 0

▼ conversión de tipos

- si T es un tipo y v es un valor, la expresión T(v) convierte el valor de v al tipo T.
- var i int = 42

  var f float64 = float64(i) + 0.000001

▼ named types

type Celsius float64

type Fahrenheit float64

- los tipos nombrados permiten que el compilador controle la mezcla no intencional.

▼ operadores

- se aplican sobre tipos numéricos; operandos del mismo tipo.

Operador	Descripción	Ejemplo
+	Suma	x + y
-	Resta	x - y
*	Multiplicación	x * y
/	División	x / y
%	Módulo	x % y
-	Menos unario	-x
+	Más unario	+x

- operadores sobre strings

Operador	Descripción	Ejemplo
+	Concatenación	s = s1 + s2

- operadores de comparación
  - se aplican sobre operandos del mismo tipo

Operador	Descripción	Ejemplo
==	Igualdad	x == y
!=	Desigualdad	x != y
>	Mayor que	x > y
<	Menor que	x < y
>=	Mayor o igual a	x >= y
<=	Menor o igual a	x <= y

- operadores lógicos
  - se aplican sobre operandos boolean

Operador	Descripción	Ejemplo
&&	And	x < 5 && x < 10
||	Or	x < 5 || x > 10
!	Not	!(x < 5 && x < 10)

▼ asignación

- es una instrucción o construcción del lenguaje (no es un operador).
- permite asignar valor a variables y constantes (en su declaración).
- sintaxis
  - variable = expresión (del mismo tipo que la variable)
  - x = x + y * 10

Instrucción	Ejemplo	Equivalente a
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

- operadores (asignación?)
  - operadores de incremento y decremento

Operador	Descripción	Ejemplo
++	Incremento	x++
--	Decremento	x--

▼ clase 2 - estructuras de control, funciones

▼ iteración

```
// forma1
for {
   // secuencia
}

// forma2
sum:= 1
for sum < 1000 {
   sum++
}

//forma3
sum:= 0
for i:= 0; i < 10; i++ {
   sum+=i
}

//forma4
sum:= 1
for; sum < 1000 {
   sum+=sum
}

// forma5
for i:= 1; i < 500 {
   i+=i
}
```

▼ iteración (repeat/ do-while)

```
i:=0
for {
   i++
   if i >= 10 {
      break
   }
}
```

▼ selección “if”

```
if x > y {
   fmt.Println(x)
   fmt.Println(y)
}
```

```
if x < y {
   fmt.Println(x)
} else {
   fmt.Println(y)
}
```

```
if x > y && x > z {
   fmt.Println("x")
} else if y > x && y > z {
   fmt.Println("y")
} else {
   fmt.Println("z")
}
```

- con sentencia de inicialización

```
if [variable := expresión;] condición {
   sentencias
} [else if condición {
   sentencias
}] [else {
   sentencias
}]
```

```
x := 3.0
n := 2.0
lim := 10.0

if v := math.Pow(x, n); v < lim {
   fmt.Println(v)
} else {
   fmt.Println(lim)
}

fmt.Println(v)
```

▼ selección “switch”

```
switch [[variable := expresión;] selector]
  {case expresión:
    sentencias}
  [default:
    sentencias]
}
```

▼ switch normal

```
package main
import (
```

```
    "fmt"
    "runtime"
)
func main() {
    switch runtime.GOOS {
        case "dar" + "win":
            fmt.Println("OS X.");
        case "li" + "nux":
            fmt.Println("linux");
        default:
            fmt.Println("other");
    }
}
```

▼ switch con sentencia de inicialización

```
package main
import (
    "fmt"
    "runtime"
)
func main(){
    switch os:= runtime.GOOS; os {
        case "darwin":
            fmt.Println("OS X.")
        case "linux":
            fmt.Println("linux")
        default:
            fmt.Println("other")
    }
}
```

▼ switch sin selector

```
package main
import (
    "fmt"
    "time"
```

```
)

func main () {
   t:= time.Now()
   switch {
      case t.Hour() < 12:
         fmt.Println("Good Morning")
      case t.Hour() < 17:
         fmt.Println("Good Afternoon")
      default:
         fmt.Println("Good evening")
   }
}
```

- ▼ funciones
  - ▼ sin return

```
func nombreFuncion() {
  fmt.Println("Esta es una función")
}
```

```
nombreFuncion()
```

```
func add(x int, y int) {
  fmt.Println(x + y)
}
```

```
add(2, 3)
```

```
func add(x, y int) {
  fmt.Println(x + y)
}
```

```
add(2, 3)
```

  - ▼ con return

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

```
z := add(2, 3)
```

```
func swap(x int, y int) (int, int) {
  return y, x
}
```

```
a, b = swap(a, b)
```

```
func swap(x1 int, y1 int) (x2, y2 int)
{
  x2, y2 = y1, x1
  return
}
```

```
a, b = swap(a, b)
```

- ▼ fmt

- `fmt.Println`, `fmt.Print` y `fmt.Printf` para mostrar datos. `Printf` permite controlar cómo se ven los valores (número de decimales, alineación, relleno). Los verbos como `%d`, `%f` y `%s` son los más usados.
- ejemplo

```
nombre := "Ana"
edad := 30
fmt.Printf("Nombre: %s, Edad: %d\n", nombre, edad)
```

▼ clase 3 - arrays, slices, for range, maps

▼ arrays

▼ definición

- secuencia indexada de elementos de un mismo tipo, de longitud fija, con primer índice en cero.

```
var x [5]int
x[4] = 100
fmt.Println(x) // [0,0,0,100]
```

▼ distintas maneras de declararlos

```
package main

import "fmt"

func main() {
    // Forma 1: declaración explícita y asignación individual
    var x [5]float64
    x[0] = 98
    x[1] = 93
    x[2] = 77
    x[3] = 82
    x[4] = 83
    fmt.Println(x)
```

```
    // Forma 2: declaración con inicialización directa
    x2 := [5]float64{98, 93, 77, 82, 83}
    fmt.Println(x2)

    // Forma 3: longitud inferida automáticamente con [...]
    x3 := [...]float64{98, 93, 77, 82, 83}
    fmt.Println(x3)
}
```

▼ distintas maneras de recorrerlos

```
package main

import "fmt"

func main() {
    x := []float64{1, 2, 3, 4, 5}

    var total float64 = 0
    for i := 0; i < 5; i++ {
        total += x[i]
        x[i] = total
    }
    fmt.Println(x)
    fmt.Println(total)

    total = 0
    for i, value := range x {
        total += value
        fmt.Println(i, value)
    }
    fmt.Println(total)

    total = 0
    for _, value := range x {
        total += value
    }
```

```
    fmt.Println(total)
}
```

▼ más

```
// inicialización posicional vs nombrada
arr := [5] int {1:10, 2:20, 3:30}
fmt.Println(arr)

// longitud
fmt.Println(len(arr))
```

▼ arrays multidimensionales

- los arrays pueden contener elementos de cualquier tipo, incluso arrays

```
package main

import "fmt"

func main() {
    // Forma 1: declaración y asignación manual
    var a [2][2]string
    a[0][0] = "Hello"
    a[0][1] = "World"
    a[1][0] = "Hola"
    a[1][1] = "Mundo"

    fmt.Println(a[0], a[1]) // [Hello World] [Hola Mundo]
    fmt.Println(a)          // [[Hello World] [Hola Mundo]]

    // Forma 2: inicialización directa
    b := [2][2]string{
        {"Hello", "World"},
        {"Hola", "Mundo"},
    }

    fmt.Println(b[0], b[1]) // [Hello World] [Hola Mundo]
```

```
    fmt.Println(b)          // [[Hello World] [Hola Mundo]]
}
```

▼ slices

▼ qué es un slice?

- la longitud de un tipo array es parte de la defición del tipo.
  - esto hace que el uso de arrays sea un poco incómodo.
- entonces es más frecuente usar slices que arrays
- un slice es un “segmento” de un array
- son indexables y tienen una longitud
  - pero esta longitud puede cambiar

▼ tres formas de crear slices

```
package main

import "fmt"

func main() {
    // 1. Slice desde un array
    a := [6]int{10, 11, 12, 13, 14, 15}
    s := a[2:4] // elementos con índices 2 y 3: [12 13]
    fmt.Println("s:", s, "len:", len(s), "cap:", cap(s))

    // 2. Declaración directa de slices
    var s1 []int               // nil slice
    s2 := []int{}              // slice vacío
    s3 := []int{1, 2, 3}       // slice inicializado
    fmt.Println("s1:", s1, "len:", len(s1), "cap:", cap(s1))
    fmt.Println("s2:", s2, "len:", len(s2), "cap:", cap(s2))
    fmt.Println("s3:", s3, "len:", len(s3), "cap:", cap(s3))

    // 3. Usando make()
    s4 := make([]int, 5, 10)    // longitud 5, capacidad 10
    s5 := make([]int, 5)        // longitud y capacidad 5
    fmt.Println("s4:", s4, "len:", len(s4), "cap:", cap(s4))
```

```
    fmt.Println("s5:", s5, "len:", len(s5), "cap:", cap(s5))
}
```

▼ slices multidimensionales

1. no existen arrays multidimensionales nativos - se implementan como slices de slices.
2. declaración

```
matriz := make([][]int, filas)
for i := range matriz {
    matriz[i] = make([]int, columnas)
}
```

3. inicialización literal

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

4. características
   - Cada slice interno puede tener diferente longitud (matriz irregular)
   - Se accede con doble índice: `matriz[fila][columna]`
   - El slice externo contiene referencias a slices internos
5. iteración

```
for i, fila := range matriz {
    for j, val := range fila {
        // trabajar con val
    }
}
```

▼ agregar elementos a un slice

```
slice = append(slice, elem1, elem2)
slice = append(slice, anotherSlice...)

var s []int
printSlice("s", s)  // s len=0 cap=0 []

s = append(s, 0)
printSlice("s", s)  // s len=1 cap=1 [0]

s = append(s, 1)
printSlice("s", s)  // s len=2 cap=2 [0 1]

s = append(s, 2, 3, 4)
printSlice("s", s)  // s len=5 cap=6 [0 1 2 3 4]

s = append(s, s...)
printSlice("s", s)  // s len=10 cap=12 [0 1 2 3 4 0 1 2 3 4]
```

▼ iteración

```
package main

import "fmt"

func main() {
    // Slice de ejemplo
    slice := []int{10, 20, 30, 40}

    // 1. Iteración básica (índice y valor)
    fmt.Println("1. Iteración con índice y valor:")
    for i, v := range slice {
        fmt.Printf("Índice: %d, Valor: %d\n", i, v)
    }

    // 2. Solo valores (ignorando índice)
    fmt.Println("\n2. Iteración solo valores:")
    for _, v := range slice {
        fmt.Println("Valor:", v)
```

```
    }

    // 3. Solo índices
    fmt.Println("\n3. Iteración solo índices:")
    for i := range slice {
        fmt.Println("Índice:", i)
    }

    // 4. Iteración tradicional con len
    fmt.Println("\n4. Iteración tradicional:")
    for i := 0; i < len(slice); i++ {
        fmt.Println("Elemento:", slice[i])
    }

    // 5. Modificación durante iteración
    fmt.Println("\n5. Modificación del slice:")
    for i := range slice {
        slice[i] *= 2 // Duplica cada elemento
    }
    fmt.Println("Slice modificado:", slice)

    /*
    Key Points:
    - range es la forma más limpia y segura
    - Usa _ para ignorar índices o valores
    - len(slice) es útil para control manual
    - Los cambios en v no modifican el slice original (usar slice[i])
    */
}
```

▼ maps

▼ que es un map?

- colección no ordenada de pares clave - valor.
- también llamados arreglos asociativos, tablas hash o diccionarios.
- no permite claves duplicadas.

- el valor por defecto es nil.
- se puede agregar, modificar y eliminar elementos, excepto que el map sea nil.

▼ operaciones

```
// Agregar o modificar
   m[key] = value
// Eliminar
   delete(m, key)
// Recuperar
   elem = m[key]       // si key no está en m, elem es el
   elem, ok = m[key]  // "zero" del tipo correspondiente
   elem, ok := m[key] // si elem y ok no están declaradas
```

▼ los maps son referencias

```
var a = map[string]string
       {"brand": "Ford",
    "model": "Mustang",
    "year": "1964",
}

b := a

fmt.Println(a)  // map[brand:Ford model:Mustang year:1964]
fmt.Println(b)  // map[brand:Ford model:Mustang year:1964]

b["year"] = "1970"

fmt.Println("After change to b:")

fmt.Println(a)  // map[brand:Ford model:Mustang year:1970]
```

▼ maps range

```
func createMap(a []string) map[string]string {
  result := make(map[string]string)
```

```
  for _, e := range a {
    result[string(e[0])] = e
  }
  return result
}
```

▼ clase 4 - punteros, tipos, métodos, structs, interfaces

▼ punteros

▼ qué son?

- un puntero es la dirección de memoria de un contenido de cierto tipo.
- *T es un puntero a un valor de tipo T.
- Zero value es nil.

```
var p *int
// fmt.Println(*p) // error en ejecución
if p != nil {
  fmt.Println(*p)
} else {
   fmt.Println("nil") // nil
}
i := 42
p = &i
fmt.Println(*p) // 42
```

▼ new

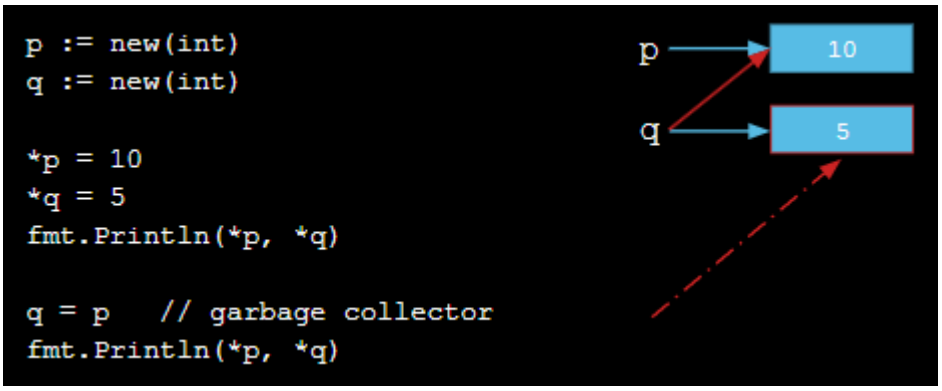
- Alocador new(T)

```
p := new(int)
q := new(int)

*p = 10
*q = 5
fmt.Println(*p, *q)  // 10 5
```

```
q = p
fmt.Println(*p, *q)  // 10 10
```

▼ garbage collector



▼ parámetros puntero

```
 func zero(xPtr *int) {
   *xPtr = 0
}

1.
func main() {
   x := 5
   zero(&x)
   fmt.Println(x) // 0
}

2.
func main() {
   xPtr := new(int)
   zero(xPtr)
   fmt.Println(*xPtr) // 0
}
```

▼ casting

- T(x)

- una conversión es permitida si ambos tipos (tipo origen y tipo destino) tienen el mismo tipo subyacente.

```
package tempconv

type Celsius float64
type Fahrenheit float64

func CToF(c Celsius) Fahrenheit {
  return Fahrenheit(c * 9 / 5 + 32)
}

func FToC(f Fahrenheit) Celsius {
  return Celsius((f - 32) * 5 / 9)
}
```

▼ métodos

- Go no tiene clases pero permite definir métodos de tipos nombrados.

```
type Celsius float64
type Fahrenheit float64

//función con un argumento “receiver"
func (c Celsius) String() string {
  return fmt.Sprintf("%g°C", c)
}

//función con un argumento “receiver"
func (f Fahrenheit) String() string {
  return fmt.Sprintf("%g°F", f)
}

func main() {
  var c Celsius = 35.0
  var f Fahrenheit = 95.0
  fmt.Println(c.String(), f.String()) // 35°C 95°F
```

```
  fmt.Println(c, f)          // 35°C 95°F
}
```

- si queremos que el el receptor se pueda modificar debemos usar un puntero.

```
type Persona struct {
    Nombre string
    Edad   int
}

// Método que usa un receptor por valor (copia)
func (p Persona) Saludar() {
    fmt.Printf("Hola, soy %s y tengo %d años\n", p.Nombre, p.Edad)
}

// Método que modifica el receptor (usa puntero *)
func (p *Persona) CumplirAños() {
    p.Edad++ // Modifica el valor original
}
```

▼ structs

- conjunto de campos de distinto tipos
  - (son los registros)

```
type Person struct {
  firstname string
  lastname string
  age int
}

var p1 Person
p2 := Person{"Pepe", "Sargento", 25}
p3 := new(Person)
p4 := Person{lastname: "Larralde",
          firstname: "José"}
```

```
p1.firstname = "John"
p1.lastname = "Doe"

p3 = &p2
```

- los structs son comparables si todos sus campos lo son

```
type Point struct{ X, Y int }
p := Point{1, 2}
q := Point{2, 1}
fmt.Println(p.X == q.X && p.Y == q.Y) // "false"
fmt.Println(p == q)                   // "false"
```

- los tipos structs comparables se pueden usar como clave de maps

```
type address struct {
    hostname string
    port int
}
hits := make(map[address]int)

hits[address{"golang.org", 443}]++
```

▼ interfaces

- un tipo interface está definido por un conjunto de firmas (encabezamiento) de métodos.
- un valor de un tipo interface puede ser cualquiera que implemente esos métodos.
- un tipo “implementa” un tipo interface, implementando sus métodos. No hay una declaración explicita de intención (como en java: implements...).

```
package main

import "fmt"

// Definición de una interfaz
```

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

// Tipo Perro implementa Animal (implícitamente)
type Perro struct{}

func (p Perro) Sonido() string {
    return "¡Guau!"
}

// Tipo Gato implementa Animal
type Gato struct{}

func (g Gato) Sonido() string {
    return "¡Miau!"
}

// Función que usa la interfaz
func HacerSonido(a Animal) {
    fmt.Println(a.Sonido())
}

func main() {
    perro := Perro{}
    gato := Gato{}

    HacerSonido(perro) // ¡Guau!
    HacerSonido(gato)  // ¡Miau!
}
```

▼ clase 5 - manejo de errores

🔷 Manejo de Errores

- En Go, los errores son parte del flujo normal del programa.
- Las funciones que pueden fallar retornan un valor adicional (`error`).
- Si hay una sola causa de error:

`v, ok := mapa["clave"]`

- Si hay múltiples causas:

`val, err := strconv.Atoi("42a")`

Estrategias:

▼ **1.Propagar el error**: retornar el mismo error al llamador.

```
package main

import (
    "errors"
    "fmt"
)

func dividir(a, b int) (int, error) {
    if b == 0 {
        return 0, errors.New("división por cero")
    }
    return a / b, nil
}

func main() {
    resultado, err := dividir(10, 0)
    if err != nil {
        // Aquí decidimos qué hacer con el error
        fmt.Println("Error:", err)
        return
    }
    fmt.Println("Resultado:", resultado)
}
```

▼ **2.Reemplazar el error**: usar `fmt.Errorf` para dar más contexto.

```
package main

import (
    "fmt"
    "os"
```

```
)

func leerArchivo(nombre string) ([]byte, error) {
    datos, err := os.ReadFile(nombre)
    if err != nil {
        return nil, fmt.Errorf("leerArchivo: no se pudo leer %s: %w", nomb
    }
    return datos, nil
}

func main() {
    _, err := leerArchivo("archivo_que_no_existe.txt")
    if err != nil {
        fmt.Println("Error:", err)
        // Se puede hacer un unwrap para detectar el error original
    }
}
```

▼ **3.Reintentar**: volver a ejecutar la operación fallida con back-off.

```
package main

import (
    "errors"
    "fmt"
    "time"
)

func operacionFallida() error {
    return errors.New("falló la operación")
}

func reintentar(maxIntentos int) error {
    var err error
    for i := 1; i <= maxIntentos; i++ {
        err = operacionFallida()
        if err == nil {
            return nil
```

```
        }
        fmt.Printf("Intento %d fallido: %v\n", i, err)
        time.Sleep(time.Duration(i) * time.Second) // back-off exponencia
    }
    return fmt.Errorf("todos los intentos fallaron: %w", err)
}

func main() {
    if err := reintentar(3); err != nil {
        fmt.Println("Error definitivo:", err)
    }
}
```

▼ **4.Finalizar el programa**: con `os.Exit(1)` o `log.Fatalf`.

```
package main

import (
    "log"
    "os"
)

func main() {
    _, err := os.ReadFile("config_critico.txt")
    if err != nil {
        log.Fatalf("Error fatal: no se pudo leer archivo de configuración: %
        // o bien:
        // fmt.Println("Error crítico, saliendo...")
        // os.Exit(1)
    }
    // resto del programa...
}
```

▼ **5.Registrar y seguir**: mostrar el error pero continuar ejecución.

```
package main

import (
```

```
    "fmt"
)

func procesarDato(dato int) error {
    if dato < 0 {
        return fmt.Errorf("dato inválido: %d", dato)
    }
    // procesar el dato normalmente
    return nil
}

func main() {
    datos := []int{10, -5, 20}
    for _, d := range datos {
        err := procesarDato(d)
        if err != nil {
            fmt.Println("Error, pero seguimos:", err)
            continue
        }
        fmt.Println("Dato procesado:", d)
    }
}
```

▼ **6.Ignorar el error**: usar un valor por defecto si hay error.

```
package main

import (
    "fmt"
    "strconv"
)

func main() {
    s := "abc" // no es número
    n, err := strconv.Atoi(s)
    if err != nil {
        n = 0 // valor por defecto
    }
```

```
    fmt.Println("Número:", n)
}
```

🔷 Paquete `errors` y `fmt.Errorf`

- `errors.New("mensaje")` crea un error simple.
- `fmt.Errorf("formato %s", var)` permite construir mensajes personalizados.

🔷 Function Values (Funciones como valores)

- Las funciones son valores de primera clase.
- Se pueden asignar a variables, pasar como parámetro o retornar.
- El valor por defecto de una función es `nil`.

🔷 Funciones Anónimas

- Se definen sin nombre, útiles para callbacks o lambdas:

```
rot13 := func(r rune) rune { ... }
strings.Map(rot13, "texto")
```

🔷 Funciones Variádicas

- Reciben una cantidad variable de argumentos:

```
func sum(vals ...int) int
```

🔷 `defer` (Llamadas diferidas)

- Se ejecutan al final de la función actual (aunque haya error).
- Útil para liberar recursos como archivos abiertos:

```
defer f.Close()
```

🔷 `panic` y `recover`

- `panic`: detiene el flujo por un error grave (no recuperable).
- `recover`: permite interceptar un `panic` en una función `defer`, y evitar que el programa se caiga.

```
defer func() {
  if p := recover(); p != nil {
    err = fmt.Errorf("internal error: %v", p)
  }
}()
```

▼ clase 6 - genéricos

🧠 ¿Qué son los Genéricos en Go?

Los **genéricos** en Go son una forma de escribir **funciones, estructuras o interfaces** que pueden trabajar con **distintos tipos de datos**, sin duplicar código.

🧰 ¿Cuándo se usan?

Se usan cuando:

- Necesitás que **una función** o **estructura** sea reutilizable con **diferentes tipos de datos**.
- Querés **evitar duplicar código** para tipos como `int`, `float64`, `string`, etc.
- Trabajás con **algoritmos** o **estructuras de datos** que no dependen del tipo exacto.

📦 Ejemplo Básico

```
func ImprimirElemento[T any](elemento T) {
  fmt.Println(elemento)
```

- `T` es un **parámetro de tipo genérico**.
- `any` significa "cualquier tipo" (alias de `interface{}`).
- Esta función puede imprimir un `int`, `string`, `bool`, etc.

Uso:

```
ImprimirElemento[string]("Hola")
```

📌 Con estructuras:

```
type Caja[T any] struct {
    Contenido T
}
```

Uso:

```
cajaEntero := Caja[int]{Contenido: 10}
cajaTexto := Caja[string]{Contenido: "texto"}
```

✅ Ventajas

- Más **reutilización** de código.
- Más **seguridad de tipos** que `interface{}`.
- Menos código repetido.

▼ clase 7 - concurrencia

▼ goroutines

- funciones capaces de ejecutarse concurrentemente con otras funciones.
- se llaman con la palabra clave → go f ( )

▼ WaitGroup

- Permite que una goroutine espere la terminación de otras goroutines
- El tipo sync.WaitGroup se puede pensar como un contador
- El tipo sync.WaitGroup define los métodos:
  - Add(delta int): incrementa (o decrementa) el contador

- Done(): decrementa en 1 el contador
- Wait(): bloquea a la goroutine que la ejecuta hasta que el contador llegue a cero

```
package main
import (
  "fmt"
  "io"
  "log"
  "net/http"
  "sync"
)

var wg sync.WaitGroup

var urls = []string{
  "https://www.golangprograms.com",
  "https://coderwall.com",
  "https://stackoverflow.com",
  "https://www.info.unlp.edu.ar",
}

func responseSize(url string) {
  defer wg.Done()

  fmt.Println("Getting ", url)
  response, err := http.Get(url)
  if err != nil {
    log.Fatal(err)
  }

  defer response.Body.Close()

  body, err := io.ReadAll(response.Body)
  if err != nil {
    log.Fatal(err)
  }
  fmt.Println(url, len(body))
```

```
}

func main() {
  for _, url := range urls {
    wg.Add(1)
    go responseSize(url)
  }
  wg.Wait()
}
```

▼ channels

- mecanismo que permite a las goroutines que se comuniquen y se sincronicen.
- conducto “tipado” a través del cual una goroutine evía datos a otra.
- por defecto, tanto la acción de enviar como la recibir bloquean a la goroutine que la ejecuta hasta que la del “otro extremo” esté lista.
- más...

```
// Se declaran antes de usarlos

   msg := make(chan string) | var msg chan string = make(chan str
   nums := make(chan int)   | var nums chan int = make(chan int)

// El “zero value” de un channer es nil

   var nums chan int // nil

// Send
   nums ← x

// Receive
   x = ←nums

// Se pueden cerrar
   close(nums)

// El receptor ...
```

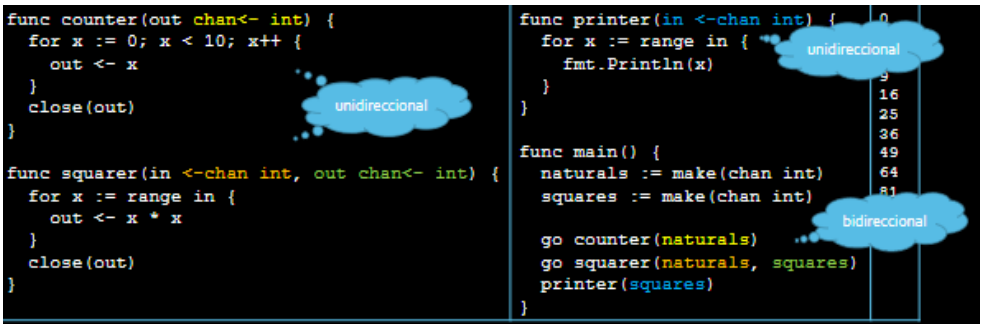
```
    x, ok := <-nums
       // si ok es false el channel no tiene más valores y está cerrad

// Range
    // Recibe valores repetidamente hasta que eventualmente el cha
    for x := range nums {
      fmt.Println(i)
    }
```

- los canales pueden ser “unidireccionales”

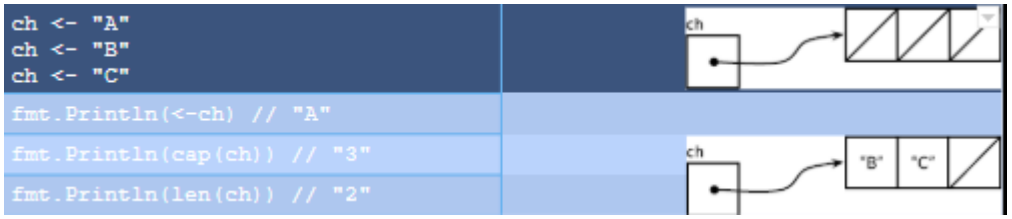
```
// Send-only channel
    chan<- int
// Receive-only channel
    <-chan int
```

  - solo la gourutine “sender” puede cerrar un send-only channel.
  - intentar cerrar un receive-only channel puede generar un error de en tiempo compilación.
  - ejemplos



▼ buffered channels

- tiene asociada una cola de elementos con la capacidad definida en la declaración.
- un “send” agrega un elemento al final de la cola y un “recieve” quita y devuelve un elemento del inicio.



▼ select

- permite que una gouroutine espere por más de un channel

```
ch1 := make(chan int)
ch2 := make(chan int)

go func() {
  for i := 1; i <= 10; i++ {
    ch1 ← i
  }
  close(ch1)
}()

go func() {
  for i := 1; i <= 10; i++ {
    ch2 ← i
  }
  close(ch2)
}()

func prnt(ch string, val int) {
  fmt.Printf("Received from %v: %v", ch, val)
}


// gouroutine que espera a más de un channel :
var val int
ok1 := true
ok2 := true

for ok1 && ok2 {
  select {
  case val, ok1 = ←ch1:
```

```
    if ok1 {
      prnt("ch1", val)
    }
  case val, ok2 = <-ch2:
    if ok2 {
      prnt("ch2", val)
    }
  }


if !ok2 {
  for val = range ch1 {
    prnt("ch1", val)
  }
}
if !ok1 {
  for val = range ch2 {
    prnt("ch2", val)
  }
}
```

- select puede utilizar una alternativa “default” para send o recieve sin bloqueo.

```
select {
  case x := <-ch1:
  case ch2 <- val:
  default:
    // operación no bloqueante
}
```

▼ exclusión mutua

- Problema: múltiples goroutines accediendo/modificando recursos compartidos → condición de carrera.
- Solución:
  - **sync.Mutex**: `Lock()` y `Unlock()` para secciones críticas.

        - **RWMutex**: para separar lectura (`RLock()`) de escritura (`Lock()`).
    - ejemplo clásico: problema de los fumadores
        - Tres fumadores con un recurso ilimitado (papel, tabaco o fósforos).
        - Un "dealer" selecciona aleatoriamente a quién le faltan los ingredientes.
        - Uso de canales y `select` para coordinar entre fumadores e ingredientes.

▼ resumen de proteccción de concurrencia

- `sync.Mutex`: protección básica.
- `sync.RWMutex`: optimiza acceso concurrente si hay muchas lecturas.
- **Monitores con canales**: alternativa basada en canales en lugar de locks.
- Evitar **deadlocks**: liberar los locks correctamente con `defer`.

▼ clase 8 - problemas clásicos de concurrencia

La presentación **"Problemas clásicos de concurrencia"** busca enseñar, usando el lenguaje **Go**, dos de los **problemas clásicos de sincronización en programación concurrente**: el **Problema de los Filósofos comensales** y el **Problema del Barbero durmiente**. Aquí va un **resumen básico**:

🍝 1. Problema de los Filósofos comensales (Dining Philosophers)

Objetivo: Simular varios procesos (filósofos) que comparten recursos (tenedores) y deben evitar errores comunes de concurrencia como *deadlock* o *inanición*.

🔄 Lógica básica:

- 5 filósofos sentados alrededor de una mesa.
- Hay 5 tenedores (uno entre cada filósofo).
- Para comer, un filósofo necesita ambos tenedores (izquierdo y derecho).
- Luego de comer, los libera.

🍣 Problemas que se ilustran:

1. **Deadlock (interbloqueo):**
   - Todos toman su tenedor izquierdo y esperan por el derecho → ninguno puede avanzar.
   - Se muestra cómo esto ocurre si todos bloquean en el mismo orden.
2. **Evitar Deadlock:**
   - Cambiar el orden de toma de tenedores según el índice (par o impar) evita el ciclo de espera circular.
3. **Inanición (Starvation):**
   - Un filósofo puede comer mucho menos si otros lo acaparan.
   - Se implementa una penalización para favorecer al que menos comió.

💈 2. Problema del Barbero durmiente

Escenario:

- Un barbero con una silla de corte y una sala de espera con espacio limitado.
- Si no hay clientes, el barbero duerme.
- Si llega un cliente y el barbero duerme, lo despierta.
- Si hay otros clientes esperando, se sientan (si hay lugar).
- Si no hay lugar, se van.

👨‍🔧 Enseñanzas:

- Coordinación entre múltiples clientes y un único recurso (el barbero).
- Uso de canales para modelar la sala de espera, el estado del barbero, y la sincronización.
- Ilustra **sincronización**, **comunicación mediante canales**, y **control del flujo concurrente**.

🧵 ¿Qué intenta enseñar la presentación?

1. **Conceptos clásicos de concurrencia**, pero aplicados con Go.

2. **Cómo usar** `sync.Mutex`**, canales y goroutines** para evitar problemas comunes como:
    - Deadlock (bloqueo mutuo)
    - Starvation (uno no avanza nunca)
3. **Modelar problemas del mundo real** con herramientas concurrentes de Go de forma segura y ordenada.

▼ deadlock

🔒 ¿Qué significa *deadlock* (interbloqueo)?

Un **deadlock** es una situación en programación concurrente donde **dos o más procesos (o goroutines)** quedan **bloqueados permanentemente**, esperando **unos por otros**, y **ninguno puede continuar**.

🧠 Ejemplo simple:

Imaginá que dos procesos quieren usar dos recursos (por ejemplo, dos tenedores en el problema de los filósofos):

1. **Proceso A** agarra el recurso 1 y espera por el recurso 2.
2. **Proceso B** agarra el recurso 2 y espera por el recurso 1.

🔁 Ambos se quedan esperando... para siempre. Ninguno libera lo que tiene, y ninguno puede seguir.

🧱 Para que ocurra un *deadlock*, suelen cumplirse estas 4 condiciones al mismo tiempo:

1. **Exclusión mutua:** los recursos no se pueden compartir.
2. **Retención y espera:** un proceso mantiene un recurso y espera por otro.
3. **No apropiación:** un recurso no puede ser forzado a liberarse.
4. **Espera circular:** hay un ciclo de procesos, cada uno esperando un recurso que otro tiene.

⚠️ En Go, puede pasar por ejemplo si:

- Usás `sync.Mutex` y dos goroutines tratan de **bloquear recursos en distinto orden**.
- Usás **canales** mal diseñados, que se quedan **esperando para siempre** a que alguien envíe o reciba.