

# Taller de Lenguajes de Programación

# **Introducción a GO**

---

Javier Villegas Lainas

09 de junio 2025



## Introducción

1. Título
2. Autor
3. Ilustrador
4. Ficción o no ficción
5. ¿Por qué elegiste este libro?

**Consejo:** ¿te resultó interesante el título? ¿Te llamó la atención la portada? ¿Lo elegiste por otro motivo? Menciona las razones por las que elegiste el libro para que tus compañeros puedan conocerte mejor.

---

## Contexto

**Consejo:** piensa en CUÁNDO y DÓNDE sucede la historia, y en cuánto TIEMPO transcurre desde el principio hasta el final. Describe el contexto como si tus compañeros estuvieran dentro de la historia.



**Consejo:** busca en Internet una imagen que refleje el entorno donde se desarrolla la historia y sustituye la de arriba.

## Instalación de GO

Para crear código Go, debe descargar e instalar las herramientas de desarrollo Go. Puede encontrar la última versión de las herramientas en la página de descargas del sitio web de Go. El instalador .pkg para Mac y el instalador .msi para Windows instalan automáticamente Go en la ubicación correcta, eliminan cualquier instalación antigua y colocan el binario de Go en la ruta de ejecución predeterminada.

La ruta del instalador la puedes encontrar en el siguiente enlace <https://go.dev/doc/install>

A continuación escogemos el paquete de instalación dependiendo del sistema operativo donde trabajemos

### Download and install

Download and install Go quickly with the steps described here.

For other content on installing, you might be interested in:

- [Managing Go installations](#) -- How to install multiple versions and uninstall.
- [Installing Go from source](#) -- How to check out the sources, build them on your own machine, and run them.

Download (1.24.4)

Una vez concluida la instalación, validamos la correcta instalación de Go. En nuestra línea de comandos o terminal escribimos lo siguiente:

Shell

```
go version
```

```
#go version go1.24.4 darwin/arm64
```

Ahora vamos a verificar las variables de entorno, para ello abrimos el **Símbolo del sistema** o **PowerShell** y ejecutamos lo siguiente:

Shell

```
echo %PATH%
```

```
echo %GOPATH%
```

```
echo %GOROOT%
```

Si las variables no están configuradas:

1. Presionar **Win + R**, escribir `sysdm.cpl` y presionar Enter
2. Ir a la pestaña **Avanzado**
3. Hacer clic en **Variables de entorno**
4. En **Variables del sistema**, agregar o editar:
  - `GOROOT: C:\Program Files\Go`
  - `GOPATH: C:\Users\%USERNAME%\go`
  - `PATH: Agregar C:\Program Files\Go\bin y %GOPATH%\bin`

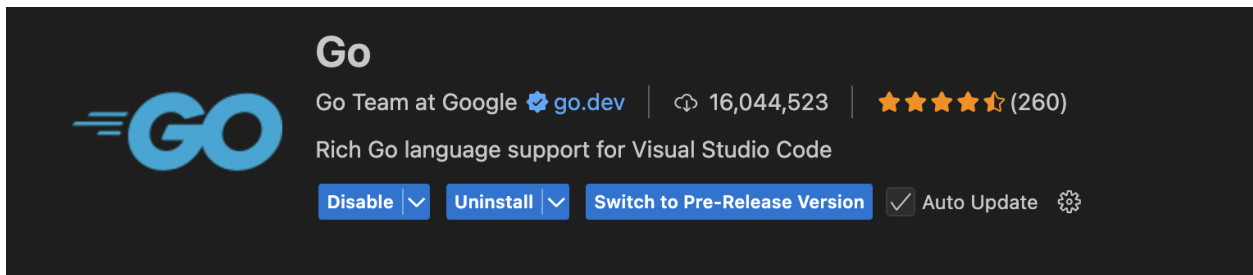
## Configuración de VSCode

### Paso 1: Instalar VSCode

- Descargar desde: <https://code.visualstudio.com/>

### Paso 2: Instalar Extensión de Go

1. Abrir VSCode
2. Ir a Extensions (**Ctrl+Shift+X**)
3. Buscar "Go" (por Google)
4. Instalar la extensión oficial



### Paso 3: Configurar la Extensión

bash

Shell

*# Abrir VSCode en un proyecto Go*

```
mkdir mi-primer-proyecto-go
```

```
cd mi-primer-proyecto-go
```

```
go mod init mi-primer-proyecto
```

```
code .
```

Al abrir un archivo `.go`, VSCode te pedirá instalar herramientas adicionales:

- Hacer clic en "Install All" cuando aparezca la notificación

### Paso 4: Verificar Configuración

1. Presionar `Ctrl+Shift+P`
2. Escribir "Go: Install/Update Tools"
3. Seleccionar todas las herramientas y hacer clic en "OK"

### Herramientas Que Se Instalan Automáticamente

- `gopls`: Language Server Protocol
- `dlv`: Debugger (Delve)
- `staticcheck`: Linter estático
- `go-outline`: Outline provider

- Y muchas más...

## Configuración Personalizada (settings.json)

json

JSON

```
{
  "go.useLanguageServer": true,
  "go.formatTool": "goimports",
  "go.lintTool": "golangci-lint",
  "go.testFlags": ["-v"],
  "go.coverOnSave": true,
  "go.coverOnSingleTest": true,
  "editor.formatOnSave": true,
  "editor.codeActionsOnSave": {
    "source.organizeImports": true
  }
}
```

## Nuestro primer programa en GO

Vamos a repasar los fundamentos de la escritura de un programa Go. A lo largo del camino, verás las partes que componen un programa Go simple.

Primero vamos a crear una carpeta para nuestro proyecto para ello vamos a la línea de comandos y creamos la carpeta llamada clase\_01\_go

Shell

```
md clase_01_go ##windows  
cd clase_01_go  
mkdir clase_01_go ##mac  
cd clase_01_go
```

Una vez dentro de nuestra carpeta vamos a ejecutar el siguiente comando

Shell

```
go mod init hello_world  
  
##esto aparecerá una vez ejecutado el comando  
  
go: creating new go.mod: module hello_world
```

Más adelante veremos en detalle acerca de los módulos en GO, pero por ahora, todo lo que necesita saber es que un **proyecto Go se llama módulo. Un módulo no es sólo código fuente. Es también una especificación exacta de las dependencias del código dentro del módulo.** Cada módulo tiene un archivo go.mod en su directorio raíz. Al ejecutar go mod init se crea este archivo. El contenido de un archivo go.mod básico es el siguiente:

Go

```
module hello_world  
  
go 1.24.4
```

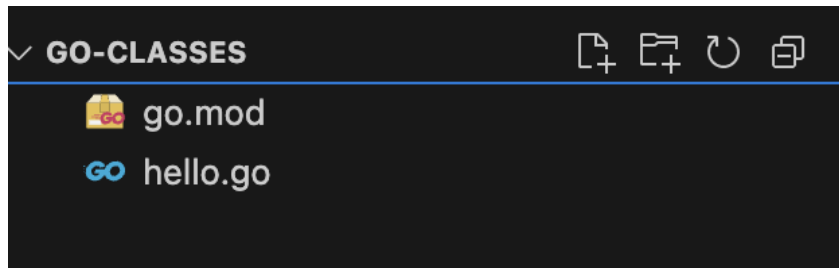
El archivo go.mod declara el nombre del módulo, la versión mínima soportada de Go para el módulo, y cualquier otro módulo del que dependa su módulo. Puedes pensar que es similar al archivo requirements.txt usado por Python o al Gemfile usado por Ruby.

**No deberías editar el archivo go.mod directamente. En su lugar, vamos a utilizar los comandos go get y go mod tidy para gestionar los cambios en el archivo.**



## Go Build

Ahora vamos a crear nuestro primer programa en Go, para ello vamos a nuestro editor y agregamos el archivo [hello.go](https://github.com/hello-go/hello-go) nos debe quedar de la siguiente manera



A continuación, escribimos el siguiente código:

```
Go


package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```

Repasemos rápidamente las partes del archivo Go que has creado. **La primera línea es una declaración de paquete. Dentro de un módulo Go, el código se organiza en uno o más paquetes. El paquete principal en un módulo Go contiene el código que inicia un programa Go.**

A continuación, hay una declaración import. La declaración import lista los paquetes referenciados en este archivo. En este ejemplo estamos usando una función en el paquete fmt (usualmente pronunciado «fumpt») de la biblioteca estándar, así que lista el paquete aquí. A diferencia de otros lenguajes, Go sólo importa paquetes completos. No puedes limitar la importación a tipos específicos, funciones, constantes o variables dentro de un paquete (Como por ejemplo Python)



Todos los programas Go comienzan con la función `main` en el paquete `main`. Esta función se declara con `func main()` y una llave izquierda. Al igual que Java, JavaScript y C, Go utiliza llaves para marcar el inicio y el final de los bloques de código.

El cuerpo de la función consiste en una sola línea. Dice que está llamando a la función `Println` del paquete `fmt` con el argumento «¡Hola, mundo!».

A continuación, vamos a ejecutar el siguiente comando:

Shell

```
go build
```

Esto crea un ejecutable llamado `hello_world` (o `hello_world.exe` en Windows) en el directorio actual. Ejecútalo y, como era de esperar, verá la palabra ¡Hola, mundo! impresa en la pantalla:

Shell

```
./hello_world
```

```
Hello, World!
```


El nombre del binario coincide con el de la declaración del módulo. Si desea un nombre diferente para su aplicación o si desea almacenarlo en una ubicación diferente, utilice el `-o`. Por ejemplo, si desea compilar el código en un binario llamado «hola», utilice lo siguiente:

Shell

```
go build -o hello
```

```
./hello
```

```
Hello, World!
```



Go fmt

`go fmt` es una herramienta **oficial** de Go que formatea automáticamente el código fuente siguiendo el **estilo estándar de Go**. Es uno de los comandos más importantes y utilizados en el ecosistema de Go.

### Funciones Principales:

- **Formateo automático:** Corrige indentación, espacios y saltos de línea
- **Estilo consistente:** Aplica las convenciones oficiales de Go
- **Sin configuración:** No necesita archivos de configuración
- **Parte del toolchain:** Viene incluido con la instalación de Go

Para hacer uso de `go fmt` debemos escribir lo siguiente

Shell

```
go fmt [archivos/paquetes]
```

Shell

# Formatear un archivo específico

```
go fmt main.go
```

# Formatear todos los archivos .go en el directorio actual

```
go fmt .
```

# Formatear recursivamente (todos los subdirectorios)

```
go fmt ./...
```

# Formatear un paquete específico

```
go fmt mipackage
```

# Formatear múltiples archivos

```
go fmt archivo1.go archivo2.go archivo3.go
```

## Ejemplos Prácticos: Antes y Después

### Ejemplo 1: Indentación y Espacios

✗ Antes de **go fmt**:

go

```
Go
package main
import "fmt"
func main(){
if true{
fmt.Println( "Mal formateado" )
}
for i:=0;i<5;i++){
fmt.Printf("Número: %d\n",i)
```

```
}}
```

✓ Después de `go fmt`:

go

```
Go
package main

import "fmt"

func main() {
    if true {
        fmt.Println("Mal formateado")
    }
    for i := 0; i < 5; i++ {
        fmt.Printf("Número: %d\n", i)
    }
}
```

## Ejemplo 2: Estructuras y Funciones

✗ Antes de `go fmt`:

go

```
Go
package main
```

```
import(  
    "fmt"  
    "time"  
)  
  
type Persona struct{  
    Nombre string  
    Edad int  
    Email string  
}  
  
func(p*Persona)Saludar()string{  
    return fmt.Sprintf("Hola, soy %s",p.Nombre)  
}  
  
func main(){  
    p:=&Persona{  
        Nombre:"Juan",  
        Edad:30,  
        Email:"juan@ejemplo.com",}  
    fmt.Println(p.Saludar())  
}
```

✓ Después de **go fmt:**

go

Go

```
package main
```

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

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

```
func (p *Persona) Saludar() string {  
    return fmt.Sprintf("Hola, soy %s", p.Nombre)  
}
```

```
func main() {  
    p := &Persona{  
        Nombre: "Juan",  
        Edad:   30,  
        Email:  "juan@ejemplo.com",  
    }
```

```
}  
  
fmt.Println(p.Saludar())  
  
}
```

### Ejemplo 3: Arrays, Slices y Maps

✗ Antes de go fmt:

go

```
Go  
  
package main  
  
import "fmt"  
  
func main(){  
    numeros:=[ ]int{1,2,3,4,5,}  
    mapa:=map[string]int{"uno":1,"dos":2,"tres":3,}  
    matriz:=[][]int{{1,2},{3,4},{5,6}},  
    for_,num:=range numeros{  
        if num%2==0{  
            fmt.Printf("Par: %d\n",num)  
        }else{  
            fmt.Printf("Impar: %d\n",num)  
        }  
    }  
}
```

✓ Después de go fmt:

go



Go

```
package main

import "fmt"

func main() {
    numeros := []int{1, 2, 3, 4, 5}
    mapa := map[string]int{"uno": 1, "dos": 2, "tres": 3}
    matriz := [][]int{{1, 2}, {3, 4}, {5, 6}}
    for _, num := range numeros {
        if num%2 == 0 {
            fmt.Printf("Par: %d\n", num)
        } else {
            fmt.Printf("Impar: %d\n", num)
        }
    }
}
```

## Go Vet

`go vet` es una **herramienta de análisis estático** oficial de Go que examina el código fuente en busca de **errores potenciales** que son **sintácticamente correctos** pero probablemente **erróneos en tiempo de ejecución**.

### Características Principales:

- **Análisis estático:** Examina código sin ejecutarlo
- **Detección de bugs:** Encuentra errores que el compilador no detecta
- **Sin falsos positivos:** Se enfoca en errores reales
- **Integrado:** Viene incluido con Go
- **Rápido:** Análisis en segundos

Para hacer uso de `go vet`, debemos escribir lo siguiente en la línea de comandos:

Shell

```
go vet [archivos/paquetes]
```

## Ejemplos de Uso

bash

Shell

*# Analizar un archivo específico*

```
go vet main.go
```

*# Analizar el paquete actual*

```
go vet .
```

*# Analizar recursivamente todos los subpaquetes*

```
go vet ./...
```

*# Analizar un paquete específico*

```
go vet mipackage
```

```
# Analizar con verbose (más información)
```

```
go vet -v ./...
```

```
# Mostrar todas las verificaciones disponibles
```

```
go vet help
```

## Tipo de Datos, Variables y Constantes en GO

### 1. Sistema de Tipos en Go

**Definición:** Un sistema de tipos es un conjunto de reglas que asigna una propiedad llamada "tipo" a diversos constructos de un programa (variables, expresiones, funciones, módulos).

Go implementa un **sistema de tipos estático fuerte**:

- **Estático:** Los tipos se verifican en tiempo de compilación
- **Fuerte:** No permite conversiones implícitas automáticas entre tipos incompatibles
- **Nominativo:** Los tipos se distinguen por su nombre, no solo por su estructura

### 2. Inferencia de Tipos

**Definición:** Capacidad del compilador de deducir automáticamente el tipo de una variable basándose en el valor asignado.

go

```
Go
```

```
var x = 42           // Go infiere que x es int
```

```
var y = 3.14         // Go infiere que y es float64
```

```
var z = "hello" // Go infiere que z es string
```

### 3. Zero Values (Valores Cero)

**Definición:** En Go, toda variable tiene un valor inicial por defecto llamado "zero value". Esto garantiza que no existan variables no inicializadas.

Tipo	Zero Value
bool	false
Numéricos	0
string	"" (cadena vacía)
Punteros	nil
Slices	nil
Maps	nil
Channels	nil
Interfaces	nil

### 4. Type Safety (Seguridad de Tipos)

**Definición:** Propiedad que garantiza que las operaciones se realicen solo en datos del tipo apropiado, previniendo errores de runtime.


go

Go

```
var a int = 10

var b float64 = 3.14

// var c = a + b // ERROR: no se puede sumar int + float64 sin
// conversión explícita

var c = float64(a) + b //  Correcto con conversión explícita
```

---

## Tipos Predeclarados (Predeclared Types)

### 1. Tipos Booleanos

go

Go

```
package main

import "fmt"

func main() {

    // Declaración explícita
    var activo bool = true
    var disponible bool = false

    // Inferencia de tipo
    var validado = true // Go infiere bool
```

```
// Zero value

var configurado bool // false por defecto


fmt.Printf("activo: %t, disponible: %t, validado: %t,
configurado: %t\n",
    activo, disponible, validado, configurado)


// Operaciones lógicas

resultado := activo && disponible
negacion := !activo


fmt.Printf("resultado: %t, negación: %t\n", resultado,
negacion)
}
```

## 2. Tipos Enteros

### Tipos con Tamaño Específico

```
Go

package main

import (
    "fmt"
```

```
"unsafe"

)

func main() {

    // Enteros con signo

    var a int8 = 127          // -128 a 127
    var b int16 = 32767       // -32,768 a 32,767
    var c int32 = 2147483647 // -2^31 a 2^31-1
    var d int64 = 9223372036854775807 // -2^63 a 2^63-1


    // Enteros sin signo

    var ua uint8 = 255        // 0 a 255
    var ub uint16 = 65535     // 0 a 65,535
    var uc uint32 = 4294967295 // 0 a 2^32-1
    var ud uint64 = 18446744073709551615 // 0 a 2^64-1


    // Tipos dependientes de la arquitectura

    var e int = 42            // 32 o 64 bits según la arquitectura
    var f uint = 42           // 32 o 64 bits según la arquitectura


    // Tipos especiales

    var g byte = 255          // Alias para uint8
}
```

```

var h rune = 'A'           // Alias para int32, representa un
punto de código Unicode

var i uintptr = 0x12345678 // Tamaño suficiente para almacenar
punteros

fmt.Printf("Tamaños en bytes:\n")

fmt.Printf("int8: %d, int16: %d, int32: %d, int64: %d\n",
    unsafe.Sizeof(a), unsafe.Sizeof(b), unsafe.Sizeof(c),
    unsafe.Sizeof(d))

fmt.Printf("uint8: %d, uint16: %d, uint32: %d, uint64: %d\n",
    unsafe.Sizeof(ua), unsafe.Sizeof(ub), unsafe.Sizeof(uc),
    unsafe.Sizeof(ud))

fmt.Printf("int: %d, uint: %d, uintptr: %d\n",
    unsafe.Sizeof(e), unsafe.Sizeof(f), unsafe.Sizeof(i))

// Operaciones aritméticas

suma := a + int8(10)
resta := b - int16(100)
producto := c * int32(2)

fmt.Printf("Operaciones: suma=%d, resta=%d, producto=%d\n",
    suma, resta, producto)

// Overflow example (comentado para evitar panic)

```



```
// var overflow int8 = 128 // ERROR: constant 128 overflows  
int8  
  
}
```

### Casos de Uso por Tipo

```
Go  
  
package main  
  
import "fmt"  
  
func main() {  
  
    // int8: Pequeños contadores, flags  
    var porcentaje int8 = 85  
  
    // int16: Puertos de red, pequeños identificadores  
    var puerto int16 = 8080  
  
    // int32: IDs de base de datos, timestamps Unix (hasta 2038)  
    var userID int32 = 1234567  
  
    // int64: Timestamps modernos, grandes contadores  
    var timestamp int64 = 1640995200000
```

```
// uint8/byte: Datos binarios, códigos de caracteres
var colorRed byte = 255

// rune: Caracteres Unicode
var emoji rune = '🚀'

// int: Uso general cuando el tamaño exacto no importa
var contador int = 42

fmt.Printf("Ejemplos de uso:\n")

fmt.Printf("Porcentaje: %d%%, Puerto: %d, UserID: %d\n",
porcentaje, puerto, userID)

fmt.Printf("Timestamp: %d, Color Rojo: %d, Emoji: %c,
Contador: %d\n",
    timestamp, colorRed, emoji, contador)
}
```

### 3. Tipos de Punto Flotante

```
Go

package main

import (
    "fmt"
```

```
"math"
)

func main() {
    // float32: Precisión simple (32 bits)
    var precio32 float32 = 19.99
    var temperatura32 float32 = 36.5

    // float64: Precisión doble (64 bits) - Por defecto en Go
    var precio64 float64 = 19.99
    var temperatura64 float64 = 36.5

    // Inferencia de tipo (siempre float64)
    var pi = 3.14159265359 // Go infiere float64

    // Zero value
    var descuento float64 // 0.0 por defecto

    fmt.Printf("Precisión float32: %.10f\n", precio32)
    fmt.Printf("Precisión float64: %.10f\n", precio64)

    // Operaciones matemáticas
```

```

area := pi * math.Pow(5.0, 2)

fmt.Printf("Área del círculo: %.2f\n", area)


// Valores especiales

var infinito = math.Inf(1)
var negInfinito = math.Inf(-1)
var noNumero = math.NaN()


fmt.Printf("Infinito: %f, -Infinito: %f, NaN: %f\n",
infinito, negInfinito, noNumero)


// Verificaciones especiales

fmt.Printf("Es infinito: %t, Es NaN: %t\n",
math.IsInf(infinito, 0), math.IsNaN(noNumero))


// Problemas comunes con punto flotante

a := 0.1
b := 0.2
c := 0.3

fmt.Printf("0.1 + 0.2 = %.17f\n", a+b)
fmt.Printf("0.1 + 0.2 == 0.3: %t\n", a+b == c)


// Comparación correcta con tolerancia

```

```
const epsilon = 1e-9

fmt.Printf("Comparación con tolerancia: %t\n",
math.Abs((a+b)-c) < epsilon)

}
```

## 4. Tipos Complejos

```
Go

package main

import (
    "fmt"
    "math/cmplx"
)

func main() {
    // complex64: Parte real e imaginaria como float32
    var c1 complex64 = 3 + 4i

    // complex128: Parte real e imaginaria como float64
    var c2 complex128 = 5 + 12i

    // Inferencia de tipo (siempre complex128)
    var c3 = 1 + 2i // Go infiere complex128
}
```

```

// Construcción usando complex()

var c4 = complex(3.0, 4.0) // complex128

var c5 = complex(float32(3.0), float32(4.0)) // complex64


// Zero value

var c6 complex128 // (0+0i) por defecto


fmt.Printf("c1: %v, c2: %v, c3: %v\n", c1, c2, c3)
fmt.Printf("c4: %v, c5: %v, c6: %v\n", c4, c5, c6)


// Extraer partes real e imaginaria

real := real(c2)

imag := imag(c2)

fmt.Printf("Parte real: %.2f, Parte imaginaria: %.2f\n",
real, imag)


// Operaciones

suma := c2 + c3

producto := c2 * c3

fmt.Printf("Suma: %v, Producto: %v\n", suma, producto)


// Funciones matemáticas complejas

```

```

modulo := cmplx.Abs(c2)

fase := cmplx.Phase(c2)

conjugado := cmplx.Conj(c2)

fmt.Printf("Módulo: %.2f, Fase: %.2f, Conjugado: %v\n",
modulo, fase, conjugado)

// Ejemplo práctico: Transformada de Fourier básica

exponencial := cmplx.Exp(1i * math.Pi) // e^(i*π) = -1

fmt.Printf("e^(i*π) = %v (debería ser aproximadamente
-1+0i)\n", exponencial)
}

```

## 5. Strings y Runes

```

Go

package main

import (
    "fmt"
    "unicode/utf8"
)

```

```

func main() {

    // Strings básicos

    var saludo string = "Hola"

    var mensaje = "¡Bienvenido a Go!" // Inferencia de tipo

    var vacio string // "" por defecto


    // String literal con comillas inversas (raw string)

    var ruta = `C:\Users\Juan\Documents\archivo.txt`

    var multilinea = `Este es un
string de múltiples
líneas`


    fmt.Printf("Saludo: '%s', Mensaje: '%s', Vacío: '%s'\n",
saludo, mensaje, vacio)

    fmt.Printf("Ruta: %s\n", ruta)

    fmt.Printf("Multilínea:\n%s\n", multilinea)


    // Propiedades de strings

    longitud := len(mensaje)

    bytes := []byte(mensaje)

    fmt.Printf("Longitud en bytes: %d, Bytes: %v\n", longitud,
bytes)

```



```

// Trabajo con Unicode

texto := "Hola 世界 🌍"

fmt.Printf("Texto: %s\n", texto)

fmt.Printf("Longitud en bytes: %d\n", len(texto))

fmt.Printf("Longitud en runes: %d\n",
utf8.RuneCountInString(texto))


// Iteración por bytes vs runes

fmt.Println("Iteración por bytes:")

for i := 0; i < len(texto); i++ {
    fmt.Printf(" Posición %d: %c (0x%02x)\n", i, texto[i],
texto[i])
}


fmt.Println("Iteración por runes:")

for i, r := range texto {
    fmt.Printf(" Posición %d: %c (U+%04X)\n", i, r, r)
}


// Runes individuales

var letra rune = 'A'

var unicode rune = '世'

var emoji rune = '🚀'

```

```

    fmt.Printf("Letra: %c (%d), Unicode: %c (%d), Emoji: %c (%d)\n",
        letra, letra, unicode, unicode, emoji, emoji)

    // Conversiones string <-> rune slice
    runes := []rune(texto)
    textoReconstruido := string(runes)
    fmt.Printf("Runes: %v\n", runes)
    fmt.Printf("Texto reconstruido: %s\n", textoReconstruido)

    // Escape sequences
    var especiales = "Línea 1\nLínea 2\tTabulada\n\"Comillas\"\n\\Backslash\\"
    fmt.Printf("Caracteres especiales:\n%s\n", especiales)
}

```

## Variables en Go

### 1. Declaración de Variables

#### Sintaxis Completa

Go

```
package main

import "fmt"

func main() {

    // Sintaxis: var nombre tipo = valor
    var edad int = 25
    var nombre string = "Juan"
    var activo bool = true

    // Sin inicialización (usa zero value)
    var contador int // 0
    var mensaje string // ""
    var disponible bool // false

    fmt.Printf("Con valores: edad=%d, nombre=%s, activo=%t\n",
edad, nombre, activo)

    fmt.Printf("Zero values: contador=%d, mensaje='%s',
disponible=%t\n",
        contador, mensaje, disponible)
}
```

## Inferencia de Tipo

```
Go

package main

import "fmt"

func main() {

    // Go infiere el tipo basándose en el valor

    var x = 42           // int
    var y = 3.14         // float64
    var z = "hello"     // string
    var w = true         // bool
    var v = 'A'          // rune (int32)

    fmt.Printf("x: %T = %v\n", x, x)
    fmt.Printf("y: %T = %v\n", y, y)
    fmt.Printf("z: %T = %v\n", z, z)
    fmt.Printf("w: %T = %v\n", w, w)
    fmt.Printf("v: %T = %v\n", v, v)
}
```

## Declaración Múltiple

```
Go

package main

import "fmt"

func main() {

    // Múltiples variables del mismo tipo
    var a, b, c int = 1, 2, 3

    // Múltiples variables de diferentes tipos
    var nombre, edad, activo = "Ana", 30, true

    // Declaración en bloque
    var (
        servidor string = "localhost"
        puerto    int    = 8080
        ssl       bool   = false
        timeout  int    = 30
    )

    fmt.Printf("a=%d, b=%d, c=%d\n", a, b, c)
```

```
    fmt.Printf("nombre=%s, edad=%d, activo=%t\n", nombre, edad,
activo)

    fmt.Printf("Configuración: %s:%d, SSL=%t, Timeout=%ds\n",
servidor, puerto, ssl, timeout)

}
```

## 2. Alcance de Variables (Scope)

```
Go

package main

import "fmt"

// Variables a nivel de paquete (package-level)
var globalCounter int = 0
var GlobalVisible int = 100 // Exportada (mayúscula inicial)

func main() {

    // Variable local a la función
    var localVar = "local"

    fmt.Printf("Global: %d, Local: %s\n", globalCounter,
localVar)
```

```

// Bloque anidado
if true {

    // Variable local al bloque
    var blockVar = "bloque"

    // Puede acceder a variables de niveles superiores
    fmt.Printf("Desde bloque - Global: %d, Local: %s, Bloque:
%s\n",
        globalCounter, localVar, blockVar)

    // Shadowing (sombreado)
    var localVar = "sombreada" // Nueva variable que sombrea
la exterior

    fmt.Printf("Variable sombreada: %s\n", localVar)
}

// blockVar no existe aquí
// fmt.Println(blockVar) // ERROR: undefined

fmt.Printf("Después del bloque - Local: %s\n", localVar) //
Valor original

// Loop scope

```

```

for i := 0; i < 3; i++ {
    var loopVar = fmt.Sprintf("iteración_%d", i)
    fmt.Printf("Loop: i=%d, loopVar=%s\n", i, loopVar)
}

// i y loopVar no existen aquí
// fmt.Println(i) // ERROR: undefined
}

func otherFunction() {
    // Puede acceder a variables globales
    globalCounter++
    fmt.Printf("Desde otra función - Global: %d\n",
globalCounter)

    // No puede acceder a variables locales de main
    // fmt.Println(localVar) // ERROR: undefined
}

```

### 3. Patrones de Inicialización

```

Go

package main

```



```
import (  
    "fmt"  
    "math"  
    "time"  
)  
  
func main() {  
    // Inicialización con valores calculados  
    var tiempo = time.Now()  
    var timestamp = tiempo.Unix()  
    var año = tiempo.Year()  
  
    // Inicialización con llamadas a funciones  
    var raizCuadrada = math.Sqrt(16)  
    var aleatorio = time.Now().Nanosecond() % 100  
  
    // Inicialización condicional  
    var mensaje string  
    if año%2 == 0 {  
        mensaje = "Año par"  
    } else {  
        mensaje = "Año impar"  
    }  
}
```

```
}

// Inicialización lazy (cuando se necesite)
var configuracion map[string]string
if configuracion == nil {
    configuracion = make(map[string]string)
    configuracion["env"] = "development"
}

fmt.Printf("Tiempo: %v\n", tiempo)
fmt.Printf("Timestamp: %d, Año: %d\n", timestamp, año)
fmt.Printf("Raíz cuadrada: %.2f, Aleatorio: %d\n",
raizCuadrada, aleatorio)
fmt.Printf("Mensaje: %s\n", mensaje)
fmt.Printf("Configuración: %v\n", configuracion)
}
```

# Constantes

## 1. Constantes Básicas

```
Go

package main

import "fmt"

// Constantes a nivel de paquete

const CompanyName = "TechCorp" // Exportada
const version = "1.0.0"         // No exportada

func main() {

    // Constantes locales

    const pi = 3.14159
    const mensaje = "¡Hola, Go!"
    const activo = true

    // Go infiere el tipo de las constantes

    fmt.Printf("Pi: %T = %v\n", pi, pi)
    fmt.Printf("Mensaje: %T = %v\n", mensaje, mensaje)
    fmt.Printf("Activo: %T = %v\n", activo, activo)

    // Las constantes deben ser evaluables en tiempo de
    compilación
```

```
const tiempoCompilacion = "Compilado el 2024"

// const tiempoEjecucion = time.Now() // ERROR: no es
constante

fmt.Printf("Constantes de empresa: %s v%s\n", CompanyName,
version)

fmt.Printf("Tiempo: %s\n", tiempoCompilacion)

}
```

## 2. Constantes Tipadas vs No Tipadas

go

```
Go

package main

import "fmt"

func main() {

    // Constantes no tipadas (untyped constants)

    const a = 42          // Constante numérica no tipada
    const b = 3.14        // Constante flotante no tipada
    const c = "hello"     // Constante string no tipada

    // Constantes tipadas (typed constants)
```

```
const d int = 42

const e float64 = 3.14

const f string = "hello"
```

*// Las constantes no tipadas pueden usarse con diferentes tipos compatibles*

```
var x1 int = a

var x2 int64 = a

var x3 float32 = a

var x4 float64 = a
```

```
fmt.Printf("Constante no tipada 'a' usada como:\n")

fmt.Printf("  int: %T = %v\n", x1, x1)

fmt.Printf("  int64: %T = %v\n", x2, x2)

fmt.Printf("  float32: %T = %v\n", x3, x3)

fmt.Printf("  float64: %T = %v\n", x4, x4)
```

*// Las constantes tipadas solo pueden usarse con su tipo exacto*

```
var y1 int = d

// var y2 int64 = d // ERROR: cannot use d (int) as int64

fmt.Printf("Constante tipada 'd': %T = %v\n", y1, y1)
```

```
// Precisión extendida en constantes no tipadas

const huge = 1e100
const tiny = 1e-100

fmt.Printf("Huge: %g, Tiny: %g\n", huge, tiny)
}
```

### 3. Enumeraciones con iota

```
Go

package main

import "fmt"

func main() {
    // Enumeración básica con iota
    const (
        Lunes = iota    // 0
        Martes           // 1
        Miercoles        // 2
        Jueves           // 3
        Viernes          // 4
    )
}
```

```

        Sabado           // 5
        Domingo          // 6
    )

    fmt.Printf("Días de la semana:\n")

    fmt.Printf("Lunes: %d, Martes: %d, Miércoles: %d\n", Lunes,
Martes, Miercoles)

// Enumeración con valores específicos
const (
    StatusInactivo = iota + 1 // 1
    StatusActivo          // 2
    StatusSuspendido      // 3
    StatusBloqueado        // 4
)

    fmt.Printf("Estados:\n")

    fmt.Printf("Inactivo: %d, Activo: %d, Suspendido: %d,
Bloqueado: %d\n",

        StatusInactivo, StatusActivo, StatusSuspendido,
StatusBloqueado)

// Enumeración con potencias de 2 (flags)

```

```

const (
    Read = 1 << iota // 1 << 0 = 1
    Write           // 1 << 1 = 2
    Execute         // 1 << 2 = 4
)

fmt.Printf("Permisos (flags):\n")

fmt.Printf("Read: %d, Write: %d, Execute: %d\n", Read, Write,
Execute)

// Combinación de permisos

const ReadWrite = Read | Write // 3
const FullAccess = Read | Write | Execute // 7

fmt.Printf("Permisos combinados:\n")

fmt.Printf("ReadWrite: %d, FullAccess: %d\n", ReadWrite,
FullAccess)

// Enumeración con saltos

const (
    Small = iota + 1 // 1
    -           // 2 (skip)
    -           // 3 (skip)
)

```



```

        Medium          // 4
        -               // 5 (skip)
        Large           // 6
    )

    fmt.Printf("Tamaños:\n")

    fmt.Printf("Small: %d, Medium: %d, Large: %d\n", Small,
Medium, Large)

    // Enumeración con tipos personalizados
    type Prioridad int
    const (
        Baja Prioridad = iota + 1
        Media
        Alta
        Critica
    )

    var taskPriority Prioridad = Alta
    fmt.Printf("Prioridad de tarea: %d\n", taskPriority)
}

```

## 4. Constantes Complejas

```
Go

package main

import (
    "fmt"
    "math"
)

func main() {
    // Constantes matemáticas
    const (
        Pi      = 3.14159265358979323846
        E       = 2.71828182845904523536
        Phi     = 1.61803398874989484820 // Golden ratio
        Sqrt2   = 1.41421356237309504880
    )

    // Constantes derivadas
    const (
        CircleArea = Pi * 5 * 5 // Área de círculo
radio 5
        Tau      = 2 * Pi // Tau = 2π
    )
}
```

```

    HalfPi      = Pi / 2
    DegreesToRad = Pi / 180
)

// Constantes con expresiones complejas
const (
    KiB = 1024
    MiB = KiB * 1024
    GiB = MiB * 1024
    TiB = GiB * 1024
)

// Constantes de configuración
const (
    MaxUsers      = 1000
    SessionTimeout = 30 * 60 // 30 minutos en segundos
    RetryAttempts  = 3
    BufferSize      = 8 * KiB
)

fmt.Printf("Constantes matemáticas:\n")
fmt.Printf("π = %.10f, e = %.10f, φ = %.10f\n", Pi, E, Phi)

```

```
fmt.Printf("Área círculo r=5: %.2f,  $\tau$  = %.6f\n", CircleArea,
Tau)
```

```
fmt.Printf("\nConstantes de almacenamiento:\n")
```

```
fmt.Printf("1 KiB = %d bytes\n", KiB)
```

```
fmt.Printf("1 MiB = %d bytes\n", MiB)
```

```
fmt.Printf("1 GiB = %d bytes\n", GiB)
```

```
fmt.Printf("1 TiB = %d bytes\n", TiB)
```

```
fmt.Printf("\nConstantes de configuración:\n")
```

```
fmt.Printf("Max usuarios: %d, Timeout: %ds, Buffer: %d
bytes\n",
```

```
MaxUsers, SessionTimeout, BufferSize)
```

```
// Uso en cálculos
```

```
angulo := 45.0
```

```
radianes := angulo * DegreesToRad
```

```
seno := math.Sin(radianes)
```

```
fmt.Printf("\nCálculo: %g° = %g radianes, sin(%g°) = %.4f\n",
```

```
angulo, radianes, angulo, seno)
```

```
}
```

# Conversiones Explícitas de Tipo

## 1. Conversiones Básicas

```
Go

package main

import "fmt"

func main() {

    // Go NO permite conversiones implícitas automáticas

    var i int = 42
    var f float64 = 3.14

    // var result = i + f // ERROR: mismatched types int and
    float64

    // Conversiones explícitas requeridas

    var result1 = float64(i) + f // Convertir int a float64
    var result2 = i + int(f)      // Convertir float64 a int
    (trunca)

    fmt.Printf("Conversiones numéricas:\n")
    fmt.Printf("int %d + float64 %.2f = %.2f\n", i, f, result1)
    fmt.Printf("int %d + int(%.2f) = %d\n", i, f, result2)
```

```

// Conversiones entre tipos enteros

var a int8 = 100
var b int16 = 200
var c int32 = 300
var d int64 = 400

// Cada conversión debe ser explícita

var suma16 = int16(a) + b
var suma32 = int32(suma16) + c
var suma64 = int64(suma32) + d

fmt.Printf("Conversiones entre enteros:\n")
fmt.Printf("int8(%d) + int16(%d) = %d\n", a, b, suma16)
fmt.Printf("Resultado anterior + int32(%d) = %d\n", c,
suma32)
fmt.Printf("Resultado anterior + int64(%d) = %d\n", d,
suma64)

// Conversiones que pueden perder datos

var grande int64 = 10000000

var pequeño int8 = int8(grande) // Overflow! Solo mantiene
los 8 bits menos significativos

```

```
fmt.Printf("Overflow: int64(%d) -> int8(%d)\n", grande,
pequeño)

// Conversiones entre signed y unsigned
var negativo int = -10

var positivo uint = uint(negativo) // Comportamiento
indefinido con números negativos

fmt.Printf("Signed a unsigned: int(%d) -> uint(%d)\n",
negativo, positivo)
}
```

## 2. Conversiones String ↔ Tipos Numéricos

```
Go

package main

import (
    "fmt"
    "strconv"
)

func main() {
```

```
// Conversiones usando strconv

// String a números
strNumero := "123"
strFloat := "3.14159"
strBool := "true"

// ParseInt(string, base, bitSize)
numero, err1 := strconv.ParseInt(strNumero, 10, 64)
if err1 != nil {
    fmt.Printf("Error convertir '%s' a int: %v\n", strNumero,
err1)
} else {
    fmt.Printf("String '%s' -> int64: %d\n", strNumero,
numero)
}

// Atoi es equivalente a ParseInt(s, 10, 0) pero retorna int
numeroInt, err2 := strconv.Atoi(strNumero)
if err2 != nil {
    fmt.Printf("Error en Atoi: %v\n", err2)
} else {
```



```
        fmt.Printf("String '%s' -> int: %d\n", strNumero,
numeroInt)
    }

    // ParseFloat
    flotante, err3 := strconv.ParseFloat(strFloat, 64)
    if err3 != nil {
        fmt.Printf("Error convertir '%s' a float: %v\n",
strFloat, err3)
    } else {
        fmt.Printf("String '%s' -> float64: %.5f\n", strFloat,
flotante)
    }

    // ParseBool
    booleano, err4 := strconv.ParseBool(strBool)
    if err4 != nil {
        fmt.Printf("Error convertir '%s' a bool: %v\n", strBool,
err4)
    } else {
        fmt.Printf("String '%s' -> bool: %t\n", strBool,
booleano)
    }
}
```

```

// Números a string
var entero int = 456
var flotante64 float64 = 2.71828
var booleano2 bool = false

// Itoa es equivalente a FormatInt(int64(i), 10)
strDesdeInt := strconv.Itoa(entero)
fmt.Printf("int %d -> string: '%s'\n", entero, strDesdeInt)

// FormatFloat(f, fmt, prec, bitSize)
strDesdeFloat := strconv.FormatFloat(flotante64, 'f', 3, 64)
fmt.Printf("float64 %.5f -> string: '%s'\n", flotante64,
strDesdeFloat)

// FormatBool
strDesdeBool := strconv.FormatBool(booleano2)
fmt.Printf("bool %t -> string: '%s'\n", booleano2,
strDesdeBool)

// Manejo de errores en conversiones
strInvalido := "abc123"
_, err := strconv.Atoi(strInvalido)
if err != nil {

```

```

        fmt.Printf("Error esperado al convertir '%s': %v\n",
strInvalido, err)
    }

    // Conversiones con diferentes bases
    strBinario := "1010"
    strOctal := "755"
    strHex := "FF"

    binario, _ := strconv.ParseInt(strBinario, 2, 64)    // Base 2
    octal, _ := strconv.ParseInt(strOctal, 8, 64)        // Base 8
    hexadecimal, _ := strconv.ParseInt(strHex, 16, 64)   // Base
16

    fmt.Printf("Conversiones con bases:\n")
    fmt.Printf("Binario '%s' (base 2) -> %d\n", strBinario,
binario)
    fmt.Printf("Octal '%s' (base 8) -> %d\n", strOctal, octal)
    fmt.Printf("Hexadecimal '%s' (base 16) -> %d\n", strHex,
hexadecimal)
}

```

### 3. Conversiones String ↔ Bytes ↔ Runes

Go

```
package main
```

```
import (
```

```
    "fmt"
```

```
    "unicode/utf8"
```

```
)
```

```
func main() {
```

```
    texto := "Hola 世界 🌍"
```

```
    // String a []byte
```

```
    bytes := []byte(texto)
```

```
    fmt.Printf("String: '%s'\n", texto)
```

```
    fmt.Printf("Bytes: %v\n", bytes)
```

```
    fmt.Printf("Longitud en bytes: %d\n", len(bytes))
```

```
    // []byte a string
```

```
    textoRecuperado := string(bytes)
```

```
    fmt.Printf("Bytes de vuelta a string: '%s'\n",  
textoRecuperado)
```

```
    // String a []rune
```

```
    runes := []rune(texto)
```

```

fmt.Printf("Runes: %v\n", runes)

fmt.Printf("Longitud en runes: %d\n", len(runes))

// []rune a string

textoDesdeRunes := string(runes)

fmt.Printf("Runes de vuelta a string: '%s'\n",
textoDesdeRunes)

// Conversión individual rune a string

var r rune = '🚀'
strDesdeRune := string(r)
fmt.Printf("Rune %c -> string: '%s'\n", r, strDesdeRune)

// Conversión byte a string (cuidado con UTF-8)

var b byte = 65 // ASCII 'A'
strDesdeByte := string(b)
fmt.Printf("Byte %d -> string: '%s'\n", b, strDesdeByte)

// Análisis detallado de UTF-8

fmt.Printf("\nAnálisis UTF-8 de '%s':\n", texto)
for i, r := range texto {
    fmt.Printf("Posición %d: rune %c (U+%04X)\n", i, r, r)
}

```

```

// Validación UTF-8

textoValido := "Texto válido"

textoInvalido := string([]byte{0xff, 0xfe, 0xfd}) // Bytes
inválidos UTF-8

fmt.Printf("'s' es UTF-8 válido: %t\n", textoValido,
utf8.ValidString(textoValido))

fmt.Printf("Texto inválido es UTF-8 válido: %t\n",
utf8.ValidString(textoInvalido))

// Contar runes sin conversión completa

cantidadRunes := utf8.RuneCountInString(texto)

fmt.Printf("Cantidad de runes en 's': %d\n", texto,
cantidadRunes)

}

```

#### 4. Conversiones Unsafe (Avanzado)

```

Go

package main

import (
    "fmt"
    "unsafe"

```

)

```
func main() {  
    // ADVERTENCIA: unsafe es peligroso y rompe garantías de Go  
    // Solo usar cuando sea absolutamente necesario y se entienda  
    // completamente  
  
    // Conversión de punteros  
    var i int64 = 0x0123456789ABCDEF  
  
    // Obtener puntero al int64  
    ptr := unsafe.Pointer(&i)  
  
    // Convertir a puntero de array de bytes  
    bytePtr := (*[8]byte)(ptr)  
  
    fmt.Printf("int64: 0x%016X\n", i)  
    fmt.Printf("Como bytes: %v\n", *bytePtr)  
  
    // Conversión string a []byte sin copia (PELIGROSO)  
    s := "Hello, World!"  
  
    // Estructura interna de string
```

```
type StringHeader struct {
    Data uintptr
    Len  int
}

// Estructura interna de slice
type SliceHeader struct {
    Data uintptr
    Len  int
    Cap  int
}

// Obtener header del string
sHeader := (*StringHeader)(unsafe.Pointer(&s))

// Crear slice header con los mismos datos
var b []byte
bHeader := (*SliceHeader)(unsafe.Pointer(&b))
bHeader.Data = sHeader.Data
bHeader.Len = sHeader.Len
bHeader.Cap = sHeader.Len
```



```
fmt.Printf("String: '%s'\n", s)
fmt.Printf("Slice (sin copia): %v\n", b)
fmt.Printf("Slice como string: '%s'\n", string(b))

// PELIGRO: Modificar el slice afectaría el string (inmutable)
// NO HACER: b[0] = 'h' // Esto causaría un crash

// Tamaños de tipos
fmt.Printf("\nTamaños de tipos (en bytes):\n")
fmt.Printf("bool: %d\n", unsafe.Sizeof(bool(true)))
fmt.Printf("int: %d\n", unsafe.Sizeof(int(0)))
fmt.Printf("int8: %d\n", unsafe.Sizeof(int8(0)))
fmt.Printf("int16: %d\n", unsafe.Sizeof(int16(0)))
fmt.Printf("int32: %d\n", unsafe.Sizeof(int32(0)))
fmt.Printf("int64: %d\n", unsafe.Sizeof(int64(0)))
fmt.Printf("float32: %d\n", unsafe.Sizeof(float32(0)))
fmt.Printf("float64: %d\n", unsafe.Sizeof(float64(0)))
fmt.Printf("string: %d\n", unsafe.Sizeof(string("")))
fmt.Printf("[]byte: %d\n", unsafe.Sizeof([]byte{}))

// Alineación de memoria
type Estructura struct {
```

```
a bool    // 1 byte
b int64   // 8 bytes
c bool    // 1 byte
}

var est Estructura

fmt.Printf("\nAlineación de estructura:\n")
fmt.Printf("Tamaño total: %d bytes\n", unsafe.Sizeof(est))
fmt.Printf("Offset de 'a': %d\n", unsafe.Offsetof(est.a))
fmt.Printf("Offset de 'b': %d\n", unsafe.Offsetof(est.b))
fmt.Printf("Offset de 'c': %d\n", unsafe.Offsetof(est.c))
}
```

---

## Operadores := vs =

### 1. Declaración Corta (:=) vs Asignación (=)

go

```
Go

package main

import "fmt"
```

```

func main() {

    // := es declaración corta (short variable declaration)
    // Solo puede usarse dentro de funciones
    // Declara e inicializa una nueva variable

    nombre := "Juan"      // Declara nueva variable 'nombre'
    como string

    edad := 25             // Declara nueva variable 'edad' como
    int

    activo := true         // Declara nueva variable 'activo' como
    bool

    fmt.Printf("Declaración corta: %s, %d, %t\n", nombre, edad,
    activo)

    // = es asignación (assignment)
    // Solo puede usarse con variables ya declaradas

    nombre = "Ana"         // Asigna nuevo valor a variable
    existente

    edad = 30              // Asigna nuevo valor a variable
    existente

    activo = false         // Asigna nuevo valor a variable
    existente

    fmt.Printf("Después de asignación: %s, %d, %t\n", nombre,
    edad, activo)

```

```
// Error común: tratar de usar := con variable ya declarada
// nombre := "Pedro" // ERROR: no new variables on left side
of :=

// Error común: tratar de usar = con variable no declarada
// salario = 50000 // ERROR: undefined: salario

// Correcto: declarar primero, asignar después
var salario int // Declaración
salario = 50000 // Asignación

fmt.Printf("Salario: %d\n", salario)
}
```

## 2. Declaración Múltiple con :=

```
Go

package main

import "fmt"

func main() {
    // Declaración múltiple con :=
}
```

```

x, y, z := 1, 2.5, "tres"
fmt.Printf("Múltiple: x=%d, y=%.1f, z=%s\n", x, y, z)

// Al menos una variable debe ser nueva para usar :=
a := 10

// a, b := 20, 30 // ERROR: no new variables (si 'a' ya
existe)
a, b := 20, 30 // OK: 'b' es nueva variable

fmt.Printf("a=%d, b=%d\n", a, b)

// Reasignación múltiple con =
a, b = 100, 200
fmt.Printf("Después de reasignación: a=%d, b=%d\n", a, b)

// Intercambio de variables (swap)
a, b = b, a
fmt.Printf("Después de intercambio: a=%d, b=%d\n", a, b)

// Funciones que retornan múltiples valores
cociente, resto := dividir(17, 5)
fmt.Printf("17 ÷ 5 = %d, resto = %d\n", cociente, resto)

```

```
// Ignorar valores con _
resultado, _ := dividir(20, 3) // Ignora el resto
fmt.Printf("Solo cociente: %d\n", resultado)

// Manejo de errores típico en Go
valor, err := convertirString("123")
if err != nil {
    fmt.Printf("Error: %v\n", err)
} else {
    fmt.Printf("Valor convertido: %d\n", valor)
}
}

func dividir(a, b int) (int, int) {
    return a / b, a % b
}

func convertirString(s string) (int, error) {
    // Simulación de conversión
    if s == "123" {
        return 123, nil
    }
}
```

```
    return 0, fmt.Errorf("no se puede convertir '%s'", s)
}
```

### 3. Scoping y Shadowing

```
Go

package main

import "fmt"

var global = "global" // Variable global

func main() {
    // Variable local que sombrea la global
    global := "local"
    fmt.Printf("Variable local: %s\n", global)

    // Crear nuevo scope con bloque
    {
        // Nueva variable que sombrea la local
        global := "bloque"
        fmt.Printf("Variable de bloque: %s\n", global)
    }
}
```

```

        // Declarar nueva variable en este scope
        temp := "temporal"
        fmt.Printf("Variable temporal: %s\n", temp)
    }

    // temp no existe aquí
    // fmt.Println(temp) // ERROR: undefined

    fmt.Printf("De vuelta a local: %s\n", global)

    // Acceder a la variable global original
    fmt.Printf("Variable global original: %s\n", getGlobal())

    // Caso especial: redeclaración parcial
    x, y := 1, 2
    fmt.Printf("Iniciales: x=%d, y=%d\n", x, y)

    // Solo y es nueva, x ya existe pero se permite
    x, z := 10, 30 // x se reasigna, z se declara
    fmt.Printf("Después: x=%d, y=%d, z=%d\n", x, y, z)

    // Shadowing en loops

```



```
for i := 0; i < 3; i++ {  
    // i es local al loop  
    for i := 10; i < 13; i++ { // Nueva i que sombrea la  
exterior  
        fmt.Printf("Loop interno i=%d\n", i)  
    }  
    fmt.Printf("Loop externo i=%d\n", i)  
}  
}
```

```
func getGlobal() string {  
    return global // Accede a la variable global  
}
```

#### 4. Casos de Error Comunes

```
Go  
  
package main  
  
import "fmt"  
  
func main() {  
    // ERROR 1: Usar := en nivel de paquete
```

```
// message := "hello" // ERROR: non-declaration statement  
outside function body
```

```
// ERROR 2: No declarar variables nuevas con :=
```

```
x := 10
```

```
// x := 20 // ERROR: no new variables on left side of :=
```

```
// CORRECTO: Usar = para reasignar
```

```
x = 20
```

```
fmt.Printf("x reasignado: %d\n", x)
```

```
// ERROR 3: Intentar asignar a variable no declarada
```

```
// y = 30 // ERROR: undefined: y
```

```
// CORRECTO: Declarar primero
```

```
var y int
```

```
y = 30
```

```
fmt.Printf("y declarado y asignado: %d\n", y)
```

```
// ERROR 4: Tipos incompatibles en asignación
```

```
var z int = 42
```

```
// z = "hello" // ERROR: cannot use "hello" (string) as int
```

```

// ERROR 5: Usar variable fuera de su scope
if true {
    temp := "temporal"
    fmt.Printf("En if: %s\n", temp)
}

// fmt.Println(temp) // ERROR: undefined: temp

// CORRECTO: Declarar en scope apropiado
var temp string
if true {
    temp = "temporal"
}

fmt.Printf("Fuera del if: %s\n", temp)

// ERROR 6: Mixing := y var en múltiple declaración
// var a, b := 1, 2 // ERROR: var declaration list syntax
error

// CORRECTO: Usar una u otra sintaxis
var a, b = 1, 2 // var con inferencia

// 0
c, d := 3, 4 // declaración corta

```

```
    fmt.Printf("a=%d, b=%d, c=%d, d=%d\n", a, b, c, d)
}
```

## Mejores Prácticas


### 1. Nomenclatura y Convenciones

go

```
Go

package main

import "fmt"

//  BUENAS PRÁCTICAS

// Constantes: SCREAMING_SNAKE_CASE o PascalCase
const (
    MAX_CONNECTIONS = 100
    DefaultTimeout  = 30
)

// Variables públicas: PascalCase (exportadas)
var GlobalConfig string
```

```
// Variables privadas: camelCase
var localCounter int

// Tipos: PascalCase
type UserAccount struct {
    ID          uint64    // Campos públicos: PascalCase
    userName    string    // Campos privados: camelCase
    isActive    bool
}

// Interfaces: PascalCase, preferiblemente terminadas en -er
type DataReader interface {
    Read() ([]byte, error)
}

// Funciones públicas: PascalCase
func ProcessData() {}

// Funciones privadas: camelCase
func validateInput() {}
```

```

// ❌ MALAS PRÁCTICAS (evitar)

// var snake_case_var int      // No usar snake_case
// var SCREAMING_VAR int      // No usar SCREAMING para variables
// const mixedCase = 10      // Inconsistente
// type userdata struct{}      // Debería ser UserData

func main() {
    // ✅ Variables locales: camelCase descriptivo
    userCount := 10
    isProcessing := true
    maxRetryAttempts := 3

    // ✅ Nombres descriptivos, no abreviaturas oscuras
    customer := UserAccount{ID: 1, userName: "john_doe"}

    // ❌ Evitar nombres muy cortos o crípticos
    // u := UserAccount{} // Muy corto
    // usrAcct := UserAccount{} // Abreviatura confusa

    fmt.Printf("User: %+v, Count: %d, Processing: %t, Max
Retries: %d\n",
        customer, userCount, isProcessing, maxRetryAttempts)
}

```

## 2. Manejo de Errores y Validación

go

```
Go

package main




import (
    "errors"
    "fmt"
    "strconv"
)

// ✅ Definir errores como variables o constantes
var (
    ErrInvalidInput      = errors.New("entrada inválida")
    ErrDataNotFound      = errors.New("datos no encontrados")
    ErrConnectionFailed = errors.New("falló la conexión")
)

// ✅ Errores personalizados con contexto
type ValidationError struct {
    Field string
    Value interface{}
    Msg    string
}
```

```
}
```

```
func (e *ValidationError) Error() string {  
    return fmt.Sprintf("validación falló en campo '%s' con valor  
'%v': %s", e.Field, e.Value, e.Msg)  
}
```

```
func main() {  
    //  Siempre verificar errores  
    result, err := divide(10, 0)  
    if err != nil {  
        fmt.Printf("Error en división: %v\n", err)  
    } else {  
        fmt.Printf("Resultado: %.2f\n", result)  
    }  
  
    //  Validación temprana  
    if err := validateUser("", 15); err != nil {  
        fmt.Printf("Error en validación: %v\n", err)  
        return  
    }  
  
    //  Usar conversiones seguras
```



```

safeValue, err := safeStringToInt("123")

if err != nil {
    fmt.Printf("Error en conversión: %v\n", err)
} else {
    fmt.Printf("Valor convertido: %d\n", safeValue)
}

// ✅ Inicialización defensiva
data := initializeData()

if data == nil {
    fmt.Println("Error: no se pudo inicializar datos")
    return
}

fmt.Printf("Datos inicializados: %v\n", data)
}

func divide(a, b float64) (float64, error) {
    if b == 0 {
        return 0, errors.New("división por cero")
    }
    return a / b, nil
}

```

```

}

func validateUser(name string, age int) error {
    if name == "" {
        return &ValidationError{Field: "name", Value: name, Msg:
"no puede estar vacío"}
    }

    if age < 18 {
        return &ValidationError{Field: "age", Value: age, Msg:
"debe ser mayor de 18"}
    }

    return nil
}

func safeStringToInt(s string) (int, error) {
    if s == "" {
        return 0, ErrInvalidInput
    }

    value, err := strconv.Atoi(s)
    if err != nil {

```

```
        return 0, fmt.Errorf("no se pudo convertir '%s' a entero: %w", s, err)
    }

    return value, nil
}

func initializeData() map[string]interface{} {
    //  Verificar condiciones antes de proceder
    data := make(map[string]interface{})

    if data == nil {
        return nil
    }

    data["initialized"] = true
    data["timestamp"] = "2024-01-01T00:00:00Z"

    return data
}
```

### 3. Performance y Memoria

```
Go

package main

import (
    "fmt"
    "strings"
    "time"
)

func main() {
    fmt.Println("🚀 OPTIMIZACIONES DE PERFORMANCE")
    fmt.Println("=====")

    // ✅ Pre-asignar slices cuando se conoce el tamaño
    demonstrateSlicePreallocation()

    // ✅ Usar strings.Builder para concatenación
    demonstrateStringBuilding()

    // ✅ Evitar conversiones innecesarias
    demonstrateTypeConversions()
}
```

```

// ✅ Reutilizar variables cuando sea posible
demonstrateVariableReuse()
}

func demonstrateSlicePreallocation() {
    fmt.Println("\n--- Preasignación de Slices ---")

    size := 1000

    // ❌ Malo: crecimiento dinámico
    start := time.Now()
    var badSlice []int
    for i := 0; i < size; i++ {
        badSlice = append(badSlice, i) // Múltiples
reasignaciones
    }
    badDuration := time.Since(start)

    // ✅ Bueno: preasignar capacidad
    start = time.Now()
    goodSlice := make([]int, 0, size) // Capacidad conocida
    for i := 0; i < size; i++ {
        goodSlice = append(goodSlice, i)
    }
    goodDuration := time.Since(start)
}

```

```

    }

    goodDuration := time.Since(start)

    // ✅ Mejor: asignar longitud exacta si es posible
    start = time.Now()
    bestSlice := make([]int, size)
    for i := 0; i < size; i++ {
        bestSlice[i] = i // Asignación directa
    }
    bestDuration := time.Since(start)

    fmt.Printf("Sin preasignación: %v\n", badDuration)
    fmt.Printf("Con capacidad: %v\n", goodDuration)
    fmt.Printf("Con longitud exacta: %v\n", bestDuration)
}

func demonstrateStringBuilding() {
    fmt.Println("\n--- Construcción de Strings ---")

    parts := []string{"Hola", " ", "mundo", " ", "desde", " ",
    "Go"}

    // ❌ Malo: concatenación directa (múltiples allocaciones)

```

```
start := time.Now()
var badResult string
for _, part := range parts {
    badResult += part // Crea nuevo string cada vez
}
badDuration := time.Since(start)

// ✅ Bueno: usar strings.Builder
start = time.Now()
var builder strings.Builder
builder.Grow(50) // Pre-asignar capacidad estimada
for _, part := range parts {
    builder.WriteString(part)
}
goodResult := builder.String()
goodDuration := time.Since(start)

// ✅ También bueno: strings.Join para este caso específico
start = time.Now()
bestResult := strings.Join(parts, "")
bestDuration := time.Since(start)
```

```

    fmt.Printf("Concatenación: %v -> '%s'\n", badDuration,
badResult)

    fmt.Printf("Builder: %v -> '%s'\n", goodDuration, goodResult)

    fmt.Printf("Join: %v -> '%s'\n", bestDuration, bestResult)
}

```

```

func demonstrateTypeConversions() {
    fmt.Println("\n--- Conversiones de Tipo ---")

    // ✅ Evitar conversiones innecesarias en loops
    var numbers []int64 = make([]int64, 1000)
    for i := range numbers {
        numbers[i] = int64(i)
    }

    // ❌ Malo: conversión en cada iteración
    start := time.Now()
    var badSum int64
    for _, num := range numbers {
        badSum += int64(num) // Conversión innecesaria
    }

    badDuration := time.Since(start)
}

```



```

// ✅ Bueno: evitar conversiones

start = time.Now()

var goodSum int64

for _, num := range numbers {
    goodSum += num // Sin conversión
}

goodDuration := time.Since(start)

fmt.Printf("Con conversiones innecesarias: %v (suma: %d)\n",
badDuration, badSum)

fmt.Printf("Sin conversiones: %v (suma: %d)\n", goodDuration,
goodSum)
}

func demonstrateVariableReuse() {
    fmt.Println("\n--- Reutilización de Variables ---")

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

    for i := range data {
        data[i] = map[string]int{"value": i}
    }

    // ❌ Malo: declarar variables en cada iteración

```


```

start := time.Now()
var badTotal int
for _, item := range data {
    tempValue := item["value"] // Nueva variable cada vez
    tempSquared := tempValue * tempValue
    badTotal += tempSquared
}
badDuration := time.Since(start)

// ✅ Bueno: reutilizar variables fuera del loop
start = time.Now()
var goodTotal int
var value, squared int // Declarar una vez
for _, item := range data {
    value = item["value"]
    squared = value * value
    goodTotal += squared
}
goodDuration := time.Since(start)

fmt.Printf("Declaración repetida: %v (total: %d)\n",
badDuration, badTotal)

```



```
        fmt.Printf("Reutilización: %v (total: %d)\n", goodDuration,  
goodTotal)  
    }
```