

Taller de Lenguajes de
Programación

Estructuras de Control en Go

Javier Villegas Lainas

16 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.

Estructuras de control el GO

¿Qué son las Estructuras de Control?

Las **estructuras de control** determinan el orden en que se ejecutan las instrucciones en un programa. En Go, tenemos:

- **Secuencial:** Ejecución línea por línea (por defecto)
- **Condicional:** Ejecutar código basado en condiciones (if, switch)
- **Repetitiva:** Ejecutar código múltiples veces (for, range)
- **Salto:** Alterar el flujo normal (break, continue, goto, return)

Características Únicas de Go

1. **No hay paréntesis obligatorios** en condiciones
2. **Llaves obligatorias** siempre, incluso para una línea
3. **No hay while y do-while** - solo **for**
4. **Switch sin break** automático (no fall-through por defecto)
5. **Inicialización en condicionales** permitida

```
// ✅ Estilo Go
if x := getValue(); x > 0 {
    // usar x
}
```

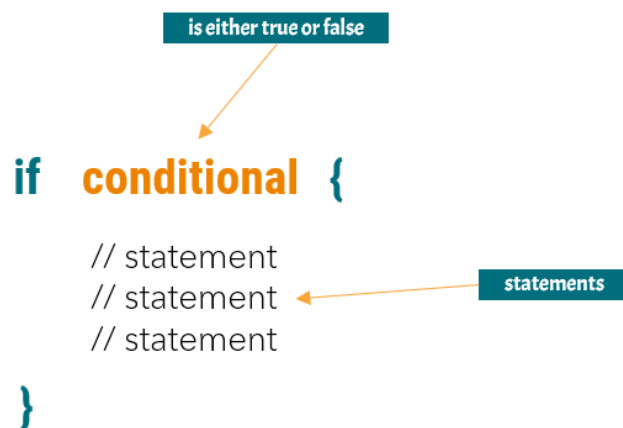
```
// ❌ No es Go (paréntesis innecesarios)
if (x > 0) {
    // código
}
```

```
// ❌ Error en Go (llaves requeridas)
if x > 0
    doSomething()
```

Estructuras condicionales

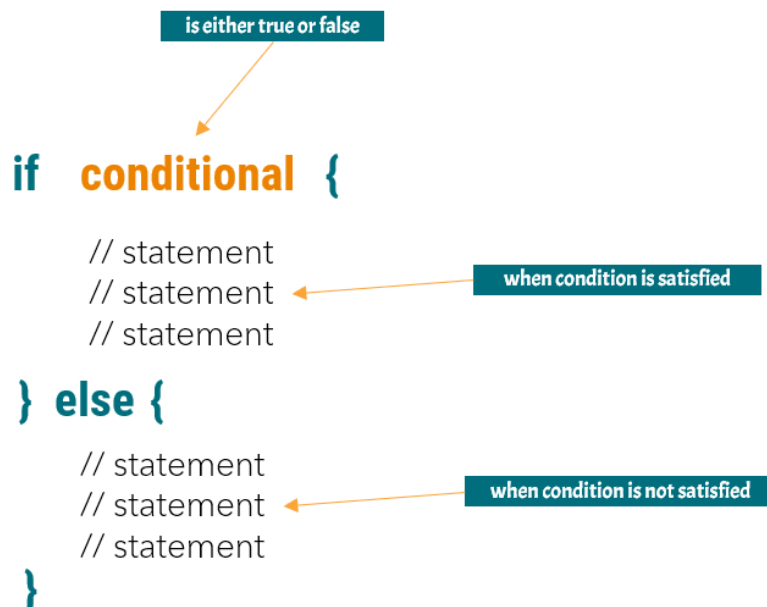
If Sentence

Go proporciona las variantes if, if-else, if-else if-else de la sentencia if/else con la que estamos familiarizados. Se utiliza para comprobar una condición, y ejecutar algún código cuando la condición es verdadera o falsa.



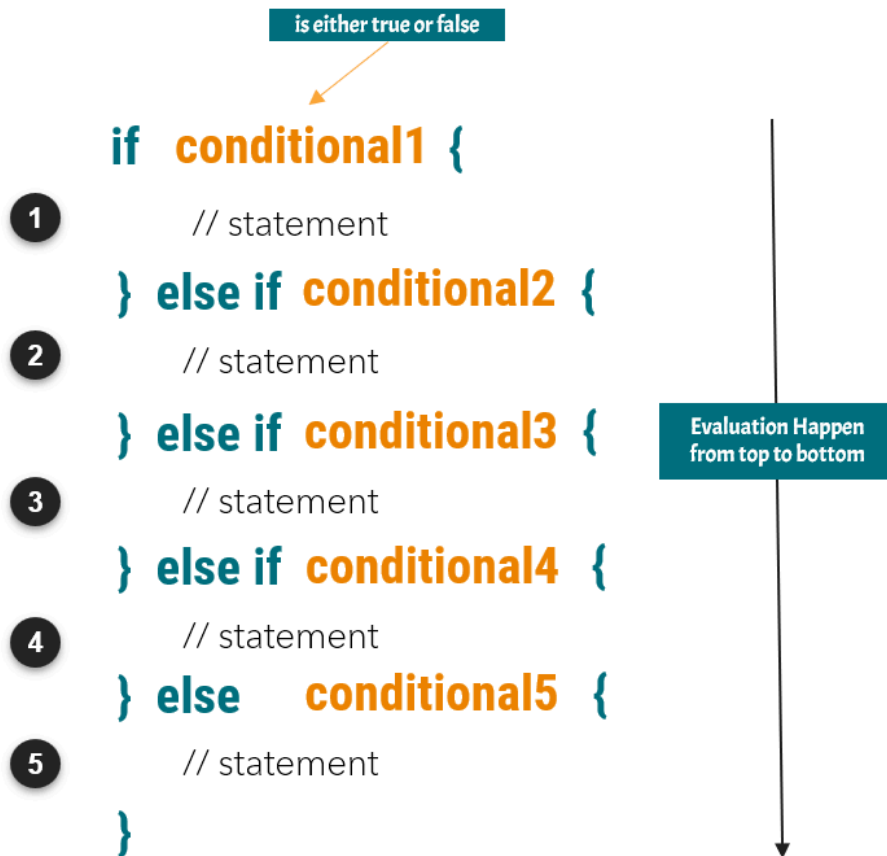
The diagram illustrates the basic syntax of an if statement. It shows the keyword **if** followed by a **conditional** in curly braces. An annotation box labeled "is either true or false" points to the **conditional**. Inside the braces, there are three lines of code, each preceded by a comment `// statement`. An annotation box labeled "statements" points to these three lines of code.

```
if conditional {  
    // statement  
    // statement  
    // statement  
}
```



The diagram illustrates the syntax of an if-else statement. It shows the keyword **if** followed by a **conditional** in curly braces. An annotation box labeled "is either true or false" points to the **conditional**. Inside the first set of braces, there are three lines of code, each preceded by a comment `// statement`. An annotation box labeled "when condition is satisfied" points to these three lines. This is followed by the keyword **else** and another set of curly braces. Inside the second set of braces, there are three lines of code, each preceded by a comment `// statement`. An annotation box labeled "when condition is not satisfied" points to these three lines.

```
if conditional {  
    // statement  
    // statement  
    // statement  
} else {  
    // statement  
    // statement  
    // statement  
}
```



```
Go  
  
package main  
  
import (  
    "fmt"  
    "strconv"  
    "strings"
```

```
    "time"
)

func main() {
    fmt.Println("=== ESTRUCTURAS IF/ELSE ===")

    // IF BÁSICO
    edad := 25

    if edad >= 18 {
        fmt.Println("✅ Mayor de edad")
    }

    // IF-ELSE
    temperatura := 22

    if temperatura > 25 {
        fmt.Println("🔥 Hace calor")
    } else {
        fmt.Println("🌡️ Temperatura agradable")
    }

    // IF-ELSE-IF (cadena)
    puntuacion := 85
```

```

if puntuacion >= 90 {
    fmt.Println("🏆 Excelente")
} else if puntuacion >= 75 {
    fmt.Println("👍 Bueno")
} else if puntuacion >= 60 {
    fmt.Println("😐 Regular")
} else {
    fmt.Println("😞 Necesita mejorar")
}

// IF CON INICIALIZACIÓN (patrón muy común en Go)
if hora := time.Now().Hour(); hora < 12 {
    fmt.Println("🌅 Buenos días")
} else if hora < 18 {
    fmt.Println("☀️ Buenas tardes")
} else {
    fmt.Println("🌙 Buenas noches")
}

// VERIFICACIÓN DE ERRORES (patrón idiomático)
if numero, err := strconv.Atoi("123"); err != nil {
    fmt.Printf("❌ Error de conversión: %v\n", err)
} else {
    fmt.Printf("✅ Número convertido: %d\n", numero)
}

```

```
// MÚLTIPLES CONDICIONES
```

```
usuario := "admin"
```

```
password := "secret123"
```

```
if usuario == "admin" && password == "secret123" {
```

```
    fmt.Println("🔑 Acceso concedido")
```

```
} else {
```

```
    fmt.Println("🚫 Acceso denegado")
```

```
}
```

```
// CONDICIONES COMPLEJAS
```

```
estado := "activo"
```

```
ultimoAcceso := time.Now().Add(-24 * time.Hour)
```

```
if estado == "activo" && time.Since(ultimoAcceso) < 30*24*time.Hour {
```

```
    fmt.Println("👤 Usuario activo y reciente")
```

```
} else if estado == "activo" {
```

```
    fmt.Println("⚠️ Usuario activo pero inactivo por tiempo")
```

```
} else {
```

```
    fmt.Println("❌ Usuario inactivo")
```

```
}
```

```
// CASOS PRÁCTICOS
```

```
demonstrarCasosPracticosIf()
```



```
}
```

```
func demostrarCasosPracticosIf() {  
    fmt.Println("\n--- Casos prácticos con if ---")  
  
    // 1. Validación de entrada  
    email := "usuario@dominio.com"  
  
    if len(email) == 0 {  
        fmt.Println("❌ Email vacío")  
    } else if !strings.Contains(email, "@") {  
        fmt.Println("❌ Email inválido: falta @")  
    } else if !strings.Contains(email, ".") {  
        fmt.Println("❌ Email inválido: falta dominio")  
    } else {  
        fmt.Println("✅ Email válido")  
    }  
  
    // 2. Categorización de rangos  
    velocidad := 75 // km/h  
    limite := 60  
  
    if velocidad <= limite {  
        fmt.Println("🚗 Velocidad normal")  
    } else if velocidad <= limite+10 {
```

```

        fmt.Println("⚠️ Ligero exceso de velocidad")
    } else if velocidad <= limite+20 {
        fmt.Println("🚔 Exceso moderado - multa")
    } else {
        fmt.Println("🚚 Exceso grave - suspensión")
    }
}

// 3. Lógica de negocio con múltiples factores
edad := 25
experiencia := 3 // años
certificaciones := 2

if edad >= 21 && experiencia >= 2 && certificaciones >= 1 {
    fmt.Println("✅ Candidato calificado para posición senior")
} else if edad >= 18 && (experiencia >= 1 || certificaciones >= 1) {
    fmt.Println("✅ Candidato calificado para posición junior")
} else if edad >= 18 {
    fmt.Println("⚠️ Candidato para posición de entrenamiento")
} else {
    fmt.Println("❌ No cumple requisitos mínimos")
}

// 4. Manejo de casos especiales
valor := 0.0

```

```
if valor > 0 {
    fmt.Printf("Valor positivo: %.2f\n", valor)
} else if valor < 0 {
    fmt.Printf("Valor negativo: %.2f\n", valor)
} else {
    // Caso especial: exactamente cero
    fmt.Println("Valor es exactamente cero")
}

// 5. Verificación de recursos
memoryUsage := 85.5 // porcentaje
cpuUsage := 70.2
diskUsage := 45.0

alertLevel := "normal"

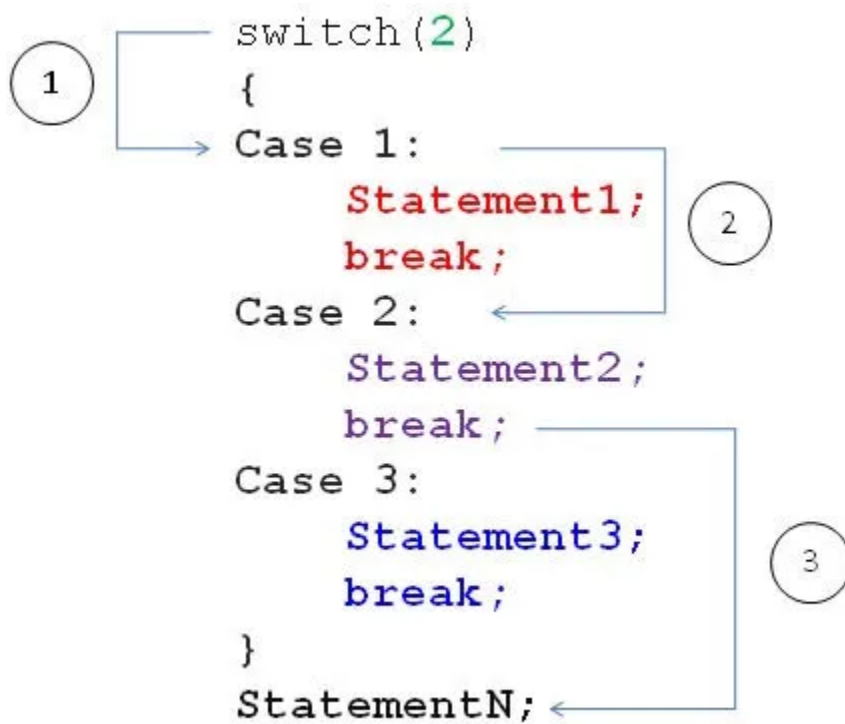
if memoryUsage > 90 || cpuUsage > 90 || diskUsage > 95 {
    alertLevel = "crítico"
} else if memoryUsage > 80 || cpuUsage > 80 || diskUsage > 85 {
    alertLevel = "warning"
}

switch alertLevel {
case "crítico":
    fmt.Println("🚨 ALERTA CRÍTICA: Recursos del sistema agotados")
}
```

```
case "warning":
    fmt.Println("⚠️ ADVERTENCIA: Alto uso de recursos")
default:
    fmt.Println("✅ Recursos del sistema normales")
}
```

Switch Case Sentence

Switch case es una sentencia de flujo de control que permite a los desarrolladores evaluar una expresión y ejecutar diferentes bloques de código basados en el valor de esa expresión. Es una construcción común en los lenguajes de programación, incluido Golang.



La sintaxis para esta instrucción es la siguiente

Unset

```
switch expression {  
  case value1:  
    // code to execute if expression is equal to value1  
  case value2:  
    // code to execute if expression is equal to value2  
  case value3:  
    // code to execute if expression is equal to value3  
  default:  
    // code to execute if expression is not equal to any of the  
    values above  
}
```

En la sentencia switch case, se evalúa la expresión y se ejecuta el código dentro del bloque correspondiente al caso coincidente. Si ninguno de los casos coincide, se ejecuta el bloque por defecto (si está presente).

Vamos a desarrollar un caso práctico para que nos quede más claro.

Go

```
package main
```

```
import (  
    "fmt"  
    "runtime"  
    "time"  
)  
  
func main() {  
    fmt.Println("=== ESTRUCTURAS SWITCH ===")  
  
    // SWITCH BÁSICO  
    dia := time.Now().Weekday()  
  
    switch dia {  
    case time.Monday:  
        fmt.Println("😓 Lunes - Inicio de semana")  
    case time.Tuesday:  
        fmt.Println("💪 Martes - A trabajar")  
    case time.Wednesday:  
        fmt.Println("🐪 Miércoles - Mitad de semana")  
    case time.Thursday:  
        fmt.Println("🚀 Jueves - Casi llegamos")  
    case time.Friday:  
        fmt.Println("🎉 Viernes - ¡Fin de semana próximo!")  
    case time.Saturday, time.Sunday:  
        fmt.Println("☂️ Fin de semana")  
    }
```

```

default:
    fmt.Println("😬 Día desconocido")
}

// SWITCH CON INICIALIZACIÓN

switch mes := time.Now().Month(); mes {
case time.December, time.January, time.February:
    fmt.Println("❄️ Época de verano (Hemisferio Sur)")
case time.March, time.April, time.May:
    fmt.Println("🍂 Otoño")
case time.June, time.July, time.August:
    fmt.Println("🌨️ Invierno")
case time.September, time.October, time.November:
    fmt.Println("🌸 Primavera")
}

// SWITCH SIN EXPRESIÓN (actúa como if-else-if)

hora := time.Now().Hour()

temperatura := 22.0

switch {
case hora < 6:
    fmt.Println("🌃 Madrugada")
case hora < 12 && temperatura > 20:
    fmt.Println("☀️ Mañana agradable")
}

```

```
case hora < 12:
    fmt.Println("☀️ Mañana fresca")
case hora < 18 && temperatura > 25:
    fmt.Println("☀️ Tarde calurosa")
case hora < 18:
    fmt.Println("☀️ Tarde normal")
default:
    fmt.Println("🌙 Noche")
}
```

// SWITCH CON FALLTHROUGH (poco común)

```
numero := 3

switch numero {
case 1:
    fmt.Print("uno")
    fallthrough
case 2:
    fmt.Print("dos")
    fallthrough
case 3:
    fmt.Print("tres")
    fallthrough
case 4:
    fmt.Print("cuatro")
}
```



```

    }

    fmt.Println() // Nueva línea

    // SWITCH CON TYPE ASSERTION

    var interfaz interface{} = "texto"

    switch valor := interfaz.(type) {
    case string:
        fmt.Printf("Es string: '%s' (longitud: %d)\n", valor, len(valor))
    case int:
        fmt.Printf("Es entero: %d\n", valor)
    case float64:
        fmt.Printf("Es float: %.2f\n", valor)
    case bool:
        fmt.Printf("Es booleano: %t\n", valor)
    case nil:
        fmt.Println("Es nil")
    default:
        fmt.Printf("Tipo desconocido: %T\n", valor)
    }

    // CASOS PRÁCTICOS CON SWITCH

    demostrarCasosPracticosSwitch()
}

```

```

func demostrarCasosPracticosSwitch() {
    fmt.Println("\n--- Casos prácticos con switch ---")

    // 1. Procesamiento de códigos de estado HTTP
    statusCode := 404

    switch statusCode {
    case 200:
        fmt.Println("✅ OK")
    case 201:
        fmt.Println("✅ Creado")
    case 400:
        fmt.Println("❌ Petición incorrecta")
    case 401:
        fmt.Println("🔒 No autorizado")
    case 403:
        fmt.Println("🚫 Prohibido")
    case 404:
        fmt.Println("🔍 No encontrado")
    case 500:
        fmt.Println("💣 Error interno del servidor")
    default:
        if statusCode >= 200 && statusCode < 300 {
            fmt.Println("✅ Éxito")
        } else if statusCode >= 400 && statusCode < 500 {

```

```

        fmt.Println("❌ Error del cliente")
    } else if statusCode >= 500 {
        fmt.Println("💥 Error del servidor")
    } else {
        fmt.Printf("😞 Código desconocido: %d\n", statusCode)
    }
}

```

// 2. Categorización de archivos por extensión

```

filename := "documento.pdf"
extension := filename[len(filename)-3:]

switch extension {
case "pdf":
    fmt.Println("📄 Documento PDF")
case "doc", "docx":
    fmt.Println("📝 Documento de Word")
case "xls", "xlsx":
    fmt.Println("📊 Hoja de cálculo")
case "jpg", "png", "gif":
    fmt.Println("🖼️ Imagen")
case "mp4", "avi", "mov":
    fmt.Println("🎬 Video")
case "mp3", "wav", "flac":
    fmt.Println("🎵 Audio")
}

```

```

default:
    fmt.Printf("📁 Archivo de tipo: %s\n", extension)
}

// 3. Lógica de permisos por rol
rol := "admin"
accion := "delete_user"

switch rol {
case "super_admin":
    fmt.Println("🔑 Acceso total - Todas las acciones permitidas")
case "admin":
    switch accion {
    case "create_user", "edit_user", "view_user":
        fmt.Println("✅ Acción permitida para admin")
    case "delete_user":
        fmt.Println("⚠️ Acción sensible - Requiere confirmación")
    default:
        fmt.Println("❌ Acción no permitida para admin")
    }
case "moderator":
    switch accion {
    case "view_user", "edit_user":
        fmt.Println("✅ Acción permitida para moderador")
    default:

```

```

        fmt.Println("❌ Acción no permitida para moderador")
    }
case "user":
    switch accion {
    case "view_user":
        fmt.Println("✅ Solo visualización permitida")
    default:
        fmt.Println("❌ Acción no permitida para usuario regular")
    }
default:
    fmt.Println("❌ Rol no reconocido")
}

// 4. Procesamiento por sistema operativo
os := runtime.GOOS

switch os {
case "linux":
    fmt.Println("🐧 Configuración para Linux")
    configurarLinux()
case "darwin":
    fmt.Println("🍏 Configuración para macOS")
    configurarMacOS()
case "windows":
    fmt.Println("🪟 Configuración para Windows")

```

```

        configurarWindows()

default:

    fmt.Printf("😞 Sistema operativo no soportado: %s\n", os)

}

// 5. State machine simple

estado := "inicio"

evento := "login_exitoso"

nuevoEstado := procesarEstado(estado, evento)

fmt.Printf("Estado: %s -> Evento: %s -> Nuevo Estado: %s\n", estado,
evento, nuevoEstado)

}

func configurarLinux() {

    fmt.Println(" - Configurando paths de Linux")

    fmt.Println(" - Estableciendo permisos UNIX")

}

func configurarMacOS() {

    fmt.Println(" - Configurando paths de macOS")

    fmt.Println(" - Configurando Keychain")

}

func configurarWindows() {

    fmt.Println(" - Configurando paths de Windows")

```

```
    fmt.Println(" - Configurando Registry")
}

func procesarEstado(estadoActual, evento string) string {
    switch estadoActual {
    case "inicio":
        switch evento {
        case "login_exitoso":
            return "autenticado"
        case "registro":
            return "registrando"
        default:
            return "inicio"
        }
    case "autenticado":
        switch evento {
        case "logout":
            return "inicio"
        case "timeout":
            return "sesion_expirada"
        default:
            return "autenticado"
        }
    case "sesion_expirada":
        switch evento {
```

```

        case "relogin":
            return "autenticado"

        case "timeout_final":
            return "inicio"

        default:
            return "sesion_expirada"
    }

    default:
        return "inicio"
}
}

```

For Sentence

El bucle For en Golang permite repetir una lista de sentencias múltiples veces. Otros lenguajes de programación tienen diferentes tipos de bucles como while, do, y until pero Go sólo tiene un bucle que es el bucle for

```

    for counter ; condition ; modify counter ; {

        // statement
        // statement
        // statement
    }

```


Un bucle for tiene cuatro partes y cada parte se separa con un punto y coma

- **counter**
 - Este es el punto de inicio desde donde el bucle comenzará
 - Puedes asignarle cualquier valor como 0, 1, o -1
- **condition**
 - Cada vez que el bucle comience a ejecutarse entonces evaluará la condición
 - si la condición es verdadera entonces el control irá dentro del bucle
 - si la condición es falsa entonces el bucle no se ejecutará y el control irá a las líneas debajo del bucle
 - Puedes tomar uno de los ejemplos de condición como un contador es menor de 100
- **Statements**
 - Estos son códigos reales que quieres ejecutar cuando el control entre dentro de tu bucle.
 - Puedes escribir cualquier código lógico con múltiples sentencias
- **modificar contador**
 - Una vez que la sentencia dentro del bucle es evaluada entonces puedes modificar el contador
 - Puedes incrementar el contador basado en tu código lógico o
 - Puedes decrementar el contador basado en tu código lógico.

Ahora vamos a implementar un bucle FOR

```
Go

package main

import (
    "fmt"
    "math/rand"
    "time"
```

)

```
func main() {  
    fmt.Println("=== ESTRUCTURAS FOR ===")  
  
    // FOR CLÁSICO (C-style)  
    fmt.Println("--- For clásico ---")  
    for i := 0; i < 5; i++ {  
        fmt.Printf("Iteración %d\n", i)  
    }  
  
    // FOR COMO WHILE  
    fmt.Println("\n--- For como while ---")  
    contador := 0  
    for contador < 3 {  
        fmt.Printf("Contador: %d\n", contador)  
        contador++  
    }  
  
    // FOR INFINITO  
    fmt.Println("\n--- For infinito con break ---")  
    i := 0  
    for {  
        if i >= 3 {  
            break  
        }  
    }  
}
```

```

    }

    fmt.Printf("Bucle infinito - iteración: %d\n", i)

    i++
}

// FOR CON MÚLTIPLES VARIABLES

fmt.Println("\n--- For con múltiples variables ---")

for i, j := 0, 10; i < j; i, j = i+1, j-1 {
    fmt.Printf("i=%d, j=%d, suma=%d\n", i, j, i+j)
}

// FOR CON CONDICIONES COMPLEJAS

fmt.Println("\n--- For con condiciones complejas ---")

x, y := 1, 1

for x < 100 && y < 100 {
    fmt.Printf("Fibonacci: x=%d, y=%d\n", x, y)

    x, y = y, x+y
}

// RANGE CON SLICES

fmt.Println("\n--- Range con slices ---")

frutas := []string{"manzana", "banana", "naranja", "uva"}

// Con índice y valor

for indice, fruta := range frutas {

```

```

        fmt.Printf("%d: %s\n", indice, fruta)
    }

    // Solo valores
    fmt.Println("Solo valores:")
    for _, fruta := range frutas {
        fmt.Printf("- %s\n", fruta)
    }

    // Solo índices
    fmt.Println("Solo índices:")
    for indice := range frutas {
        fmt.Printf("Índice: %d\n", indice)
    }

    // RANGE CON MAPS
    fmt.Println("\n--- Range con maps ---")
    edades := map[string]int{
        "Ana": 25,
        "Luis": 30,
        "María": 28,
    }

    for nombre, edad := range edades {
        fmt.Printf("%s tiene %d años\n", nombre, edad)
    }

```

```

    }

    // RANGE CON STRINGS

    fmt.Println("\n--- Range con strings ---")

    texto := "Hola 世界"

    // Por runes (caracteres Unicode)

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

    // CASOS PRÁCTICOS

    demostrarCasosPracticosFor()
}

func demostrarCasosPracticosFor() {
    fmt.Println("\n--- Casos prácticos con for ---")

    // 1. Procesamiento de lotes de datos

    fmt.Println("1. Procesamiento en lotes:")

    datos := make([]int, 100)

    for i := range datos {
        datos[i] = i + 1
    }
}

```

```

tamañoLote := 10

for i := 0; i < len(datos); i += tamañoLote {
    fin := i + tamañoLote
    if fin > len(datos) {
        fin = len(datos)
    }

    lote := datos[i:fin]

    fmt.Printf("  Procesando lote %d: %d elementos\n", i/tamañoLote+1,
len(lote))

    // Simular procesamiento
    time.Sleep(50 * time.Millisecond)
}

// 2. Búsqueda con múltiples criterios
fmt.Println("\n2. Búsqueda de usuarios:")

usuarios := []struct {
    ID      int
    Nombre  string
    Edad    int
    Activo  bool
    Ciudad  string
}{
    {1, "Ana García", 25, true, "Lima"},
    {2, "Luis Martín", 30, false, "Cusco"},
    {3, "María López", 28, true, "Lima"},
}

```

```

        {4, "Carlos Ruiz", 35, true, "Arequipa"},
        {5, "Elena Torres", 29, true, "Lima"},
    }

    // Buscar usuarios activos de Lima mayores de 25
    fmt.Println("Usuarios activos de Lima > 25 años:")
    for _, usuario := range usuarios {
        if usuario.Activo && usuario.Ciudad == "Lima" && usuario.Edad > 25
{
            fmt.Printf("    - %s (%d años)\n", usuario.Nombre,
usuario.Edad)
        }
    }

    // 3. Validación de datos con acumuladores
    fmt.Println("\n3. Validación de formulario:")
    campos := map[string]string{
        "nombre":  "Juan Pérez",
        "email":   "juan@email.com",
        "telefono": "123456789",
        "edad":    "25",
        "ciudad":  "",
    }

    errores := make([]string, 0)
    camposValidos := 0

```

```

for campo, valor := range campos {
    if valor == "" {
        errores = append(errores, fmt.Sprintf("Campo '%s' es
requerido", campo))
    } else {
        camposValidos++
        fmt.Printf("  ✓ %s: %s\n", campo, valor)
    }
}

if len(errores) > 0 {
    fmt.Println("  Errores encontrados:")
    for _, error := range errores {
        fmt.Printf("    ✗ %s\n", error)
    }
}

fmt.Printf("  Campos válidos: %d/%d\n", camposValidos, len(campos))

// 4. Generación de reportes con agrupación
fmt.Println("\n4. Reporte de ventas por región:")
ventas := []struct {
    Producto string
    Region   string
    Monto    float64

```



```

    }{
        {"Laptop", "Norte", 2500.00},
        {"Mouse", "Norte", 45.50},
        {"Laptop", "Sur", 2500.00},
        {"Teclado", "Centro", 120.00},
        {"Mouse", "Sur", 45.50},
        {"Laptop", "Centro", 2500.00},
    }

    ventasPorRegion := make(map[string]float64)
    contadorPorRegion := make(map[string]int)

    for _, venta := range ventas {
        ventasPorRegion[venta.Region] += venta.Monto
        contadorPorRegion[venta.Region]++
    }

    for region, total := range ventasPorRegion {
        promedio := total / float64(contadorPorRegion[region])
        fmt.Printf("  %s: $%.2f total (%d ventas, promedio: $%.2f)\n",
            region, total, contadorPorRegion[region], promedio)
    }

    // 5. Algoritmo de retry con backoff
    fmt.Println("\n5. Simulación de retry con backoff:")

```

```

maxIntentos := 5

for intento := 1; intento <= maxIntentos; intento++ {
    fmt.Printf("  Intento %d/%d", intento, maxIntentos)

    // Simular operación que puede fallar
    if rand.Float32() < 0.7 { // 70% probabilidad de fallo
        fmt.Println(" - ❌ Falló")

        if intento < maxIntentos {
            // Backoff exponencial
            delay := time.Duration(intento*intento) * 100 *
time.Millisecond

            fmt.Printf("    Esperando %v antes del siguiente
intento...\n", delay)

            time.Sleep(delay)
        }
    } else {
        fmt.Println(" - ✅ Éxito")
        break
    }
}

// 6. Algoritmo de ordenamiento burbuja
fmt.Println("\n6. Ordenamiento burbuja:")
numeros := []int{64, 34, 25, 12, 22, 11, 90}

```

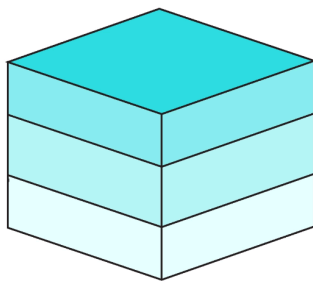
```
fmt.Printf("  Array original: %v\n", numeros)

n := len(numeros)
for i := 0; i < n-1; i++ {
    for j := 0; j < n-i-1; j++ {
        if numeros[j] > numeros[j+1] {
            numeros[j], numeros[j+1] = numeros[j+1], numeros[j]
        }
    }
}
fmt.Printf("  Array ordenado: %v\n", numeros)
}
```

Range Loop in Golang

La palabra clave `range` en Golang se utiliza con varias estructuras de datos para iterar sobre un elemento. Se utiliza más comúnmente en bucles para iterar sobre elementos de un array, mapa, slice, etc. En este artículo, aprenderemos todo sobre la palabra clave `range` en Golang.

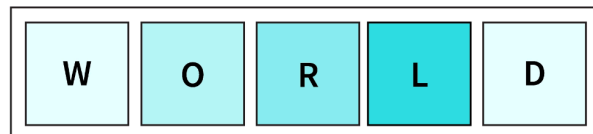
Un bucle `for-range` en Golang se utiliza para iterar sobre elementos en varias estructuras de datos como un array, slice, map, o incluso una cadena, etc. La palabra clave `range` se utiliza para iterar sobre una expresión y para evaluar estructuras de datos como slice, array, map, channel, etc.



Map



Array



String

Sintaxis

```
for index, value := range datastructure {  
    fmt.Println(value)  
}
```

Donde, `index` es el valor al que se accede. `value` es el valor actual que tenemos en cada una de las iteraciones. `datastructure` se utiliza para mantener los valores a los que accede el bucle.

Range Keyword

- La palabra clave `range` se utiliza para iterar sobre elementos de cualquier estructura de datos. Cuando se itera sobre un array o slice, devuelve el índice de un elemento en formato `int`. Del mismo modo, cuando se itera sobre un mapa, devuelve la clave y el valor.

Sin embargo, el rango puede devolver múltiples valores como salida. Para entender esto, veamos cómo el rango funciona de forma diferente para distintos tipos de datos.

- Usando range sobre un array o slice- en este, el primer valor que devuelve a la salida es un índice, y el segundo valor que devuelve es el elemento.
- Usando el rango sobre la cadena- en la cadena, el primer valor que devuelve es el índice y el segundo valor es rune int.
- Usando el rango en el mapa- el primer valor que devuelve el mapa es la clave y el segundo valor devuelve el valor del par clave-valor.
- Usando el rango en el canal- el primer valor que devuelve es un elemento y el segundo valor devuelto es ninguno.

Vamos a implementar este bucle range:

```
Go

package main

import (
    "fmt"
    "strings"
)

func main() {
    opiniones := []string{
        "El servicio fue bueno y rápido",
        "El sistema es muy lento y malo",
        "Buen producto pero entrega lenta",
        "Rápido, eficiente y bueno",
        "Malo servicio, lento y sin soporte",
    }
}
```

```

palabrasClave := []string{"bueno", "malo", "rápido", "lento"}

// Inicializar el mapa de conteo
conteo := make(map[string]int)
for _, clave := range palabrasClave {
    conteo[clave] = 0
}

// Procesar opiniones
for _, opinion := range opiniones {
    // Convertimos a minúsculas para uniformidad
    palabras := strings.Fields(strings.ToLower(opinion))

    for _, palabra := range palabras {
        // Limpiar comas o puntos si los hubiera (básico)
        palabra = strings.Trim(palabra, ".,;")

        // Verificamos si es una palabra clave
        if _, existe := conteo[palabra]; existe {
            conteo[palabra]++
        }
    }
}

// Mostrar resultados

```

```
fmt.Println("📊 Conteo de palabras clave:")

for palabra, cantidad := range conteo {
    fmt.Printf("- %s: %d veces\n", palabra, cantidad)
}
}
```

Control del Flujo

Go proporciona varias sentencias de control para alterar el flujo de los bucles. He aquí un desglose:

break:

- Termina el bucle más interno en el que se encuentra.
- La ejecución se reanuda con la sentencia que sigue inmediatamente al bucle.
- También puede utilizarse para salir de una sentencia switch.
- Puede utilizarse con etiquetas para salir de bucles anidados (véase el ejemplo siguiente).

continue:

- Salta la iteración actual del bucle.
- Pasa directamente a la siguiente iteración del bucle.
- Sólo aplicable dentro de bucles.

goto:

- Transfiere el control a una sentencia etiquetada dentro de la misma función.
- Puede utilizarse para crear bucles, pero suelen preferirse for, while y do while.
- Utilízelo con moderación, ya que puede dificultar la lectura y comprensión del código.

Etiquetas:

- Se utilizan con break, continue y goto para controlar el flujo en bucles anidados.
- Se definen con un nombre seguido de dos puntos (por ejemplo, labelName:).

Vamos a verlo con un ejemplo como trabaja esta funcionalidad de control de flujo:

```
Go

package main

import (
    "fmt"
    "math/rand"
    "strings"
    "time"
)

func main() {
    rand.Seed(time.Now().UnixNano())

    secreto := rand.Intn(10) + 1

    var input string

    jugadores := []string{"Jugador 1", "Jugador 2"}

    intentosMax := 3

    intentos := map[string]int{"Jugador 1": 0, "Jugador 2": 0}

    fmt.Println("🎮 ¡Bienvenidos al juego de adivinanza!")

    fmt.Println("1 2 Adivina el número secreto entre 1 y 10. Escribe 'salir' 3 4 para terminar.")
}
```


JUEGO:

```
for {  
    for _, jugador := range jugadores {  
        if intentos[jugador] >= intentosMax {  
            continue  
        }  
  
        fmt.Printf("👉 %s, intento %d: ", jugador,  
intentos[jugador]+1)  
  
        fmt.Scanln(&input)  
  
        if strings.ToLower(input) == "salir" {  
            fmt.Println("🛑 El juego ha sido cancelado por el  
usuario.")  
  
            goto FIN  
        }  
  
        var guess int  
        _, err := fmt.Sscanf(input, "%d", &guess)  
        if err != nil {  
            fmt.Println("❌ Entrada no válida. Escribe un  
número.")  
  
            continue  
        }  
  
        if guess%2 == 0 {
```

```

        fmt.Println("⚠ Los pares no traen suerte. Intenta
con otro número impar.")

        continue

    }

    intentos[jugador]++

    if guess == secreto {

        fmt.Printf("🎯 ¡%s adivinó el número secreto! Era %d
🎯\n", jugador, secreto)

        break JUEGO

    } else {

        fmt.Println("❌ Incorrecto. Sigue intentando.")

    }

}

// Verifica si ambos jugadores agotaron sus intentos
if intentos["Jugador 1"] >= intentosMax && intentos["Jugador 2"]
>= intentosMax {

    fmt.Println("😞 Ambos jugadores agotaron sus intentos.")

    break

}

}

FIN:

    fmt.Println("🎯 Fin del juego. El número secreto era:", secreto)

}

```

Defer, panic y recover

Defer

La sentencia defer es una forma conveniente de ejecutar un trozo de código antes de que retorne una función, como se explica en la especificación de Golang:

En su lugar, las funciones aplazadas se invocan inmediatamente antes de que retorne la función que las rodea, en el orden inverso al que fueron aplazadas.

Casos comunes de uso de Defer

Defer asegura que ciertas operaciones ocurran de forma fiable, independientemente de cómo salga una función. Es particularmente útil para:

- Cerrar archivos
- Desbloquear mutexes
- Limpiar conexiones a bases de datos

Vamos a ver un caso práctico para el uso de Defer en Go

```
Go

package main

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

```
func main() {  
    fmt.Println("=== DEFER ===")  
  
    // DEFER BÁSICO  
    demostrarDeferBasico()  
  
    // MÚLTIPLES DEFERS  
    demostrarMultiplesDefers()  
  
    // DEFER CON VALORES  
    demostrarDeferConValores()  
  
    // CASOS PRÁCTICOS  
    demostrarCasosPracticosDefer()  
}  
  
func demostrarDeferBasico() {  
    fmt.Println("--- Defer básico ---")  
  
    fmt.Println("1. Inicio de función")  
  
    defer fmt.Println("4. Este mensaje se ejecuta al final (defer)")  
  
    fmt.Println("2. En medio de función")  
    fmt.Println("3. Antes del return")  
}
```

```

    // El defer se ejecuta aquí automáticamente
}

func demostrarMultiplesDefers() {
    fmt.Println("\n--- Múltiples defers (LIFO - Last In, First Out) ---")

    defer fmt.Println("🏆 Tercer defer (se ejecuta primero)")
    defer fmt.Println("🥈 Segundo defer (se ejecuta segundo)")
    defer fmt.Println("🥇 Primer defer (se ejecuta último)")

    fmt.Println("Código normal ejecutándose...")
}

func demostrarDeferConValores() {
    fmt.Println("\n--- Defer con valores capturados ---")

    x := 10

    defer fmt.Printf("Valor de x en defer: %d (capturado al definir defer)\n",
x)

    x = 20

    fmt.Printf("Valor actual de x: %d\n", x)

    // El defer usará x=10, no x=20
}

```

```
// Para usar el valor actual, usar función anónima
defer func() {
    fmt.Printf("Valor actual de x en defer con closure: %d\n", x)
}()

x = 30

fmt.Printf("Valor final de x: %d\n", x)
}

func demostrarCasosPracticosDefer() {
    fmt.Println("\n--- Casos prácticos con defer ---")

    // 1. Manejo de archivos
    fmt.Println("1. Manejo de archivos:")
    manejarArchivo()

    // 2. Medición de tiempo
    fmt.Println("\n2. Medición de tiempo:")
    medirTiempoEjecucion()

    // 3. Cleanup de recursos
    fmt.Println("\n3. Cleanup de recursos:")
    simularConexionDB()

    // 4. Logging de entrada y salida
```

```

fmt.Println("\n4. Logging:")
funcionConLogging("parametro_importante")

// 5. Mutex unlocking
fmt.Println("\n5. Manejo de mutex:")
simularMutex()
}

func manejarArchivo() {
    // Simular apertura de archivo
    fmt.Println(" 📁 Abriendo archivo...")

    // defer se ejecuta incluso si hay error
    defer fmt.Println(" 🔒 Cerrando archivo (defer)")

    // Simular trabajo con archivo
    fmt.Println(" 📝 Escribiendo datos...")
    fmt.Println(" 📖 Leyendo datos...")

    // Si hubiera un error aquí, defer aún se ejecutaría
    // return // El defer se ejecuta antes del return
}

func medirTiempoEjecucion() {
    inicio := time.Now()

```

```

defer func() {
    duracion := time.Since(inicio)
    fmt.Printf(" 🕒 Función tardó: %v\n", duracion)
}()

fmt.Println(" 🔄 Iniciando operación costosa...")
time.Sleep(100 * time.Millisecond) // Simular trabajo
fmt.Println(" ✅ Operación completada")
}

func simularConexionDB() {
    fmt.Println(" 🐼 Conectando a base de datos...")

    defer fmt.Println(" 🐼 Desconectando de base de datos (defer)")

    // Simular múltiples operaciones
    fmt.Println(" 📊 Ejecutando query 1...")
    fmt.Println(" 📊 Ejecutando query 2...")
    fmt.Println(" 📊 Ejecutando query 3...")
}

func funcionConLogging(parametro string) {
    fmt.Printf(" 📦 ENTRADA: funcionConLogging(%s)\n", parametro)
}

```



```

defer fmt.Println(" 🍷 SALIDA: funcionConLogging")

// Lógica de la función
fmt.Println(" ⚙️ Procesando lógica de negocio...")

if parametro == "error" {
    fmt.Println(" ❌ Error simulado")
    return // defer aún se ejecuta
}

fmt.Println(" ✅ Procesamiento exitoso")
}

func simularMutex() {
    fmt.Println(" 🔒 Adquiriendo lock...")

    defer fmt.Println(" 🔓 Liberando lock (defer)")

    // Simular trabajo en sección crítica
    fmt.Println(" ⚙️ Trabajando en sección crítica...")
    time.Sleep(50 * time.Millisecond)
}

```

Panic

En Go, un Panic es una función incorporada que detiene la ejecución normal de un programa. Se utiliza para señalar un error irrecuperable que el programa no puede manejar. Cuando se produce un pánico, el programa detiene inmediatamente la ejecución de la función actual, y comienza a desenrollar la pila de llamadas, ejecutando cualquier función diferida en el camino.

He aquí algunos puntos clave sobre el pánico:

- **Propósito:** Indicar un error crítico que impide que el programa continúe de forma segura.
- **Activación:** Se desencadena llamando a la función de pánico incorporada, que acepta un argumento de cualquier tipo.
- **Desenrollado:** Cuando se produce un pánico, el programa desenrolla la pila, ejecutando funciones diferidas.
- **Terminación:** Si no se recupera el pánico, el programa termina, imprimiendo una traza de pila.

```
Go

package main

import "fmt"

func causaPanico() {
    fmt.Println("Antes del pánico 🤯")
    panic("¡Algo salió mal!")
    fmt.Println("Después del pánico ❌") // no se ejecuta
}

func main() {
    causaPanico()
    fmt.Println("Esto nunca se ejecuta ❌")
}
```

Recover

En Go, la función `recover` se utiliza para recuperar el control de una goroutine en pánico. Es una función incorporada que se utiliza principalmente dentro de funciones diferidas para manejar los pánicos con gracia y evitar que el programa se bloquee.

Puntos clave sobre `recover`:

Requisito de la función diferida: `recover` sólo tiene efecto cuando se llama dentro de una función diferida. Si se llama fuera de una función diferida, devuelve `nil` y no tiene efecto.

- **Intercepción de pánico:** Cuando se produce un pánico, las funciones diferidas se ejecutan en orden inverso. Si una función diferida llama a recuperar, intercepta el pánico, detiene su propagación y devuelve el valor pasado a la función de pánico.
- **Ejecución normal:** Si no se produce el pánico, `recover` devuelve `nil` y no afecta a la ejecución normal.
- **Manejo de errores:** `recover` le permite manejar errores con elegancia, registrarlos o tomar otras acciones sin bloquear el programa.
- **Flujo de Control:** Después de llamar a `recover`, la ejecución continúa normalmente después de la función diferida.

Vamos a implementar el uso de `Recover` con un ejemplo

```
Go

package main

import "fmt"

func protegido() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("🚨 Se recuperó del pánico:", r)
        }
    }
}
```

```

    }()

    fmt.Println("Ejecutando función protegida")
    panic("🔥 Error inesperado")
}

func main() {
    protegido()
    fmt.Println("✅ El programa continúa después del recover")
}

```

Vamos a implementar otro ejercicio para lograr entender mejor el uso de panic y recover. Para ello creamos un nuevo proyecto en Go, y usamos el siguiente código

```

Go

package main

import (
    "fmt"
)

func main() {
    fmt.Println("=== PANIC Y RECOVER ===")
}

```

```
// PANIC BÁSICO

demonstrarPanicBasico()

// RECOVER PARA MANEJAR PANIC

demonstrarRecover()

// CASOS PRÁCTICOS

demonstrarCasosPracticosPanicRecover()
}

func demostrarPanicBasico() {
    fmt.Println("--- Panic básico ---")

    // defer se ejecuta incluso con panic
    defer fmt.Println("3. Defer ejecutándose durante panic")

    fmt.Println("1. Antes del panic")
    fmt.Println("2. Justo antes del panic")

    // Este panic terminará el programa si no se recupera
    // panic("¡Algo salió terriblemente mal!")

    fmt.Println("Esta línea nunca se ejecutaría")
}
```

```

func demostrarRecover() {
    fmt.Println("\n--- Recover para manejar panic ---")

    // Función que puede hacer panic
    funcionPeligrosa := func() {
        defer func() {
            if r := recover(); r != nil {
                fmt.Printf(" 🚨 Panic recuperado: %v\n", r)
                fmt.Println(" 🔄 Continuando ejecución normal...")
            }
        }()

        fmt.Println(" ⚙️ Iniciando operación peligrosa...")
        panic("¡Error crítico simulado!")
        fmt.Println(" Esta línea nunca se ejecuta")
    }

    fmt.Println("1. Antes de función peligrosa")
    funcionPeligrosa()
    fmt.Println("2. Después de función peligrosa (recuperada)")
    fmt.Println("3. El programa continúa normalmente")
}

func demostrarCasosPracticosPanicRecover() {
    fmt.Println("\n--- Casos prácticos ---")

```

```

// 1. Servidor web que no debe caerse

fmt.Println("1. Simulación de servidor web:")

simularServidorWeb()

// 2. Validación estricta

fmt.Println("\n2. Validación con panic/recover:")

testValidacion()

// 3. Procesamiento de datos con recovery

fmt.Println("\n3. Procesamiento de lote con recovery:")

procesarLoteDatos()

// 4. División segura

fmt.Println("\n4. División segura:")

testDivisionSegura()

}

func simularServidorWeb() {

    // Simular múltiples requests

    requests := []string{"GET /users", "POST /users", "GET /invalid", "DELETE /users/1"}

    for i, request := range requests {

        func() {

            defer func() {

```

```

        if r := recover(); r != nil {
            fmt.Printf(" 🚨 Request %d falló: %v\n", i+1,
r)
            fmt.Println(" 📝 Logging error y
continuando...")
        }
    }()

    fmt.Printf(" 🍰 Procesando request %d: %s\n", i+1,
request)

    // Simular error en request específico
    if request == "GET /invalid" {
        panic("endpoint no válido")
    }

    fmt.Printf(" ✅ Request %d completado exitosamente\n",
i+1)
    }()
}

fmt.Println(" 🌐 Servidor continúa funcionando")
}

func testValidacion() {
    usuarios := []struct {
        Nombre string

```



```

        Edad    int
        Email   string
    }{
        {"Ana", 25, "ana@email.com"},
        {"", 30, "luis@email.com"},          // Error: nombre vacío
        {"María", -5, "maria@email.com"},    // Error: edad negativa
        {"Carlos", 35, "carlos@email.com"},
    }

    for i, usuario := range usuarios {
        func() {
            defer func() {
                if r := recover(); r != nil {
                    fmt.Printf(" ❌ Usuario %d inválido: %v\n",
i+1, r)
                }
            }()

            validarUsuario(usuario.Nombre, usuario.Edad, usuario.Email)

            fmt.Printf(" ✅ Usuario %d válido: %s\n", i+1,
usuario.Nombre)
        }()
    }
}

func validarUsuario(nombre string, edad int, email string) {

```

```

if nombre == "" {
    panic("nombre no puede estar vacío")
}

if edad < 0 {
    panic("edad no puede ser negativa")
}

if email == "" {
    panic("email no puede estar vacío")
}
}

func procesarLoteDatos() {
    datos := []interface{}{1, "texto", 3.14, []int{1, 2, 3}, nil, 42}
    resultados := make([]string, 0)

    for i, dato := range datos {
        func() {
            defer func() {
                if r := recover(); r != nil {
                    fmt.Printf(" ⚠ Error procesando elemento %d:
%v\n", i, r)

                    resultados = append(resultados,
fmt.Sprintf("ERROR_%d", i))
                }
            }()
        }()
    }
}

```

```

        resultado := procesarDato(dato)

        resultados = append(resultados, resultado)

        fmt.Printf("  ✅ Elemento %d procesado: %s\n", i,
resultado)

    }()

}

fmt.Printf(" 📊 Resultados finales: %v\n", resultados)
}

func procesarDato(dato interface{}) string {
    switch v := dato.(type) {
    case int:
        return fmt.Sprintf("INT_%d", v*2)

    case string:
        return fmt.Sprintf("STR_%s", v)

    case float64:
        return fmt.Sprintf("FLOAT_%.2f", v)

    case nil:
        panic("no se puede procesar nil")

    default:
        panic(fmt.Sprintf("tipo no soportado: %T", v))
    }
}

func testDivisionSegura() {

```

```

operaciones := []struct {
    a, b float64
}{
    {10, 2},
    {15, 3},
    {20, 0}, // División por cero
    {25, 5},
}

for i, op := range operaciones {
    resultado := divisionSegura(op.a, op.b)
    fmt.Printf(" %.1f ÷ %.1f = %s\n", op.a, op.b, resultado)
}

func divisionSegura(a, b float64) string {
    defer func() {
        if r := recover(); r != nil {
            fmt.Printf("Error en división: %v", r)
        }
    }()

    if b == 0 {
        panic("división por cero")
    }
}

```

```

    resultado := a / b

    return fmt.Sprintf("%.2f", resultado)
}

// Función utilitaria para demostrar panic con stack trace
func demostrarStackTrace() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Printf("Panic recuperado: %v\n", r)
            // En aplicaciones reales, aquí podrías imprimir el stack
            // trace completo
        }
    }()

    funcionNivel1()
}

func funcionNivel1() {
    funcionNivel2()
}

func funcionNivel2() {
    funcionNivel3()
}

```

```
func funcionNivel3() {  
    panic("Error en función nivel 3")  
}
```