

Taller de Lenguajes de  
Programación

# Estructura de un Proyecto en GO, Funciones y Structs

---

Javier Villegas Lainas  
23 de junio 2025

\_\_\_\_\_



## Introducción

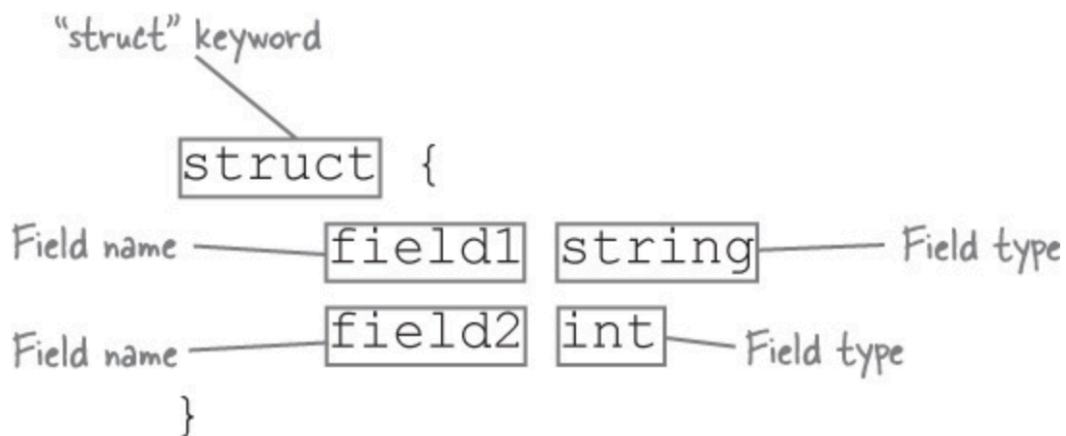
## Structs en Go

Una struct o estructura puede compararse con la clase en el paradigma de la Programación Orientada a Objetos. Si no sabes lo que es la programación orientada a objetos, imagina que una estructura es una receta que declara los ingredientes y el tipo de cada ingrediente.

Una estructura tiene diferentes campos del mismo o diferentes tipos de datos. Si comparas la estructura con una receta, los nombres de los campos de la estructura se convierten en los ingredientes (como la sal) y los tipos de campo se convierten en el tipo de estos ingredientes (como la sal de mesa).

Una estructura se utiliza principalmente cuando es necesario definir un esquema compuesto por diferentes campos individuales (propiedades). Al igual que una clase, podemos crear un objeto a partir de este esquema (la clase es análoga al esquema).

Puesto que podemos instanciar una estructura, debe haber alguna distinción de nomenclatura entre la estructura y la instancia. Por lo tanto, el nombre tipo struct se utiliza para representar el esquema de la estructura y struct o estructura se utiliza para representar la instancia.



Ahora veamos cómo se usan ese tipo de datos en Go

Go

// Definición básica de un struct

```
type Usuario struct {  
    ID      int  
    Nombre  string  
    Email   string  
    Activo  bool  
}
```

// Struct con tags (útil para JSON, validaciones, etc.)

```
type Producto struct {  
    ID      int      `json:"id" db:"product_id"`  
    Nombre  string    `json:"nombre" validate:"required"`  
    Precio  float64 `json:"precio" validate:"min=0"`  
}
```

// Struct anidado

```
type Empresa struct {  
    Nombre      string  
    Direccion   Direccion  
    Empleados []Usuario  
}
```

```
type Direccion struct {  
    Calle  string  
    Ciudad string
```

```
    CP      string
}
```

## Formas de inicializar un Struct

```
Go

// Inicialización con valores cero
var u1 Usuario

// Inicialización con valores específicos
u2 := Usuario{
    ID:      1,
    Nombre:  "Juan",
    Email:   "juan@email.com",
    Activo:  true,
}

// Inicialización parcial (otros campos toman valor cero)
u3 := Usuario{
    Nombre:  "María",
    Email:   "maria@email.com",
}

// Usando punteros (más eficiente para structs grandes)
```

```
u4 := &Usuario{
    ID:      2,
    Nombre: "Pedro",
}
```


## Structs anónimos

También se puede declarar que una variable implementa un tipo struct sin dar primero un nombre al tipo struct. Esto se denomina estructura anónima:

```
Go
var person struct {
    name string
    age  int
    pet  string
}

person.name = "bob"
person.age  = 50
person.pet  = "dog"

pet := struct {
    name string
    kind string
}{
    name: "Fido",
    kind: "dog",
}
```



En este ejemplo, los tipos de las variables persona y mascota son structs anónimos. Los campos de una estructura anónima se asignan (y leen) igual que los de una estructura con nombre. Al igual que puedes inicializar una instancia de una estructura con nombre con un literal de estructura, también puedes hacer lo mismo con una estructura anónima.

## Receptores en GO

En Go, un receptor es un parámetro especial que te permite asociar métodos con un tipo específico (como un struct). Es similar a la palabra clave `this` o `self` en otros lenguajes orientados a objetos. Usando receptores, pueden definir funciones que actúen como métodos en tus tipos personalizados, dándoles comportamiento y haciendo tu código más organizado y legible.

Puntos clave sobre los receptores:

- **Asociación con tipos:** Los receptores asocian métodos a un tipo particular, permitiéndole llamar a esos métodos directamente sobre instancias de ese tipo.
- **Receptores de valor vs. Receptores de puntero:** Go soporta tanto receptores de valor (donde se pasa una copia del tipo) como receptores de puntero (donde se pasa un puntero al tipo).
- **Modificación del objeto original:** Los receptores de puntero permiten a los métodos modificar el objeto original, mientras que los receptores de valor operan sobre una copia.
- **Sintaxis de los métodos:** El receptor va entre paréntesis antes del nombre del método, como (`p *Persona`). El receptor puede ser un valor o un puntero a un tipo.

Veamos de manera preliminar con el siguiente ejemplo:

Go

```
type Rectangle struct {  
    Width, Height int  
}  
  
func (r Rectangle) Area() int { // Value receiver
```



```
        return r.Width * r.Height
    }

    func (r *Rectangle) Scale(factor int) { // Pointer receiver
        r.Width *= factor
        r.Height *= factor
    }

    func main() {
        rect := Rectangle{Width: 10, Height: 5}
        area := rect.Area() // Calling the Area method
        fmt.Println("Area:", area)

        rect.Scale(2) // Calling the Scale method (modifies the original rect)
        fmt.Println("Scaled rect:", rect)
    }
```

En este ejemplo, Area es un método con un receptor de valores, y Scale es un método con un receptor de punteros.

## Anatomía de un receptor

```
Go

// Sintaxis básica

func (receptor TipoReceptor) NombreMetodo(parametros) retorno {

    // implementación

}

// Ejemplos

func (u Usuario) Leer() string           // Receptor de valor

func (u *Usuario) Escribir() error       // Receptor de puntero
```

## Receptor de Valor vs Receptor de Puntero

En Go, un receptor es una variable asociada a un método que permite acceder y operar sobre un tipo de dato específico. Los receptores pueden ser de valor o de puntero, y la elección entre uno u otro afecta la forma en que se modifica el valor subyacente. **Los receptores de puntero permiten modificar el valor original, mientras que los de valor trabajan con una copia.**

### Receptor de valor:

- Un método con un receptor de valor opera sobre una copia del valor original del tipo de dato.
- Cualquier modificación realizada dentro del método no afectará al valor original fuera del método.
- Se declara usando el nombre del tipo directamente, por ejemplo: `func (p Persona) metodo() {}`.
- Útil cuando se desea mantener la inmutabilidad del valor original y evitar efectos secundarios.

### Receptor de puntero:

1. Un método con un receptor de puntero opera directamente sobre la dirección de memoria del valor original.
2. Cualquier modificación realizada dentro del método afectará al valor original.
3. Se declara usando un puntero al tipo, por ejemplo: `func (p *Persona) metodo() {}`.
4. Útil cuando se desea modificar el valor original del tipo de dato dentro del método

Veamos algunas diferencias importantes entre ambos receptores

#### 1. Comportamiento en memoria

```
Go

type DatosGrandes struct {
    buffer [1000000]byte
    nombre string
    id      int
}

// ❌ INEFICIENTE - Copia 1MB en cada llamada
func (d DatosGrandes) ProcesarDatos() {
    // Se crea una copia completa de DatosGrandes
    fmt.Printf("Procesando: %s\n", d.nombre)
}

// ✅ EFICIENTE - Solo pasa referencia (8 bytes en x64)
func (d *DatosGrandes) ProcesarDatosPtr() {
    // Solo se pasa la dirección de memoria
```

```
    fmt.Printf("Procesando: %s\n", d.nombre)
}
```

## 2. Mutabilidad y Side effects

```
Go

type Contador struct {
    valor int
    logs []string
}

// Receptor de VALOR - NO modifica el original
func (c Contador) IncrementarCopia() {
    c.valor++
    c.logs = append(c.logs, "incrementado")
    // Estos cambios solo afectan la COPIA local
}

// Receptor de PUNTERO - SÍ modifica el original
func (c *Contador) IncrementarOriginal() {
    c.valor++
    c.logs = append(c.logs, "incrementado")
    // Estos cambios afectan el struct original
}
```

```

}

// Demostración
func main() {
    contador := Contador{valor: 0, logs: []string{}}

    contador.IncrementarCopia()
    fmt.Println(contador.valor) // 0 - sin cambios

    contador.IncrementarOriginal()
    fmt.Println(contador.valor) // 1 - modificado
}

```

### 3. Conversiones automáticas en GO

```

Go

type Usuario struct {
    nombre string
}

func (u Usuario) MetodoValor() string {
    return u.nombre
}

```

```

func (u *Usuario) MetodoPuntero() {
    u.nombre = "modificado"
}

func main() {
    // Con valor
    usuario := Usuario{nombre: "Juan"}
    usuario.MetodoValor()    // ✅ Directo
    usuario.MetodoPuntero() // ✅ Go convierte automáticamente a &usuario

    // Con puntero
    ptr := &Usuario{nombre: "María"}
    ptr.MetodoValor()    // ✅ Go desreferencia automáticamente (*ptr)
    ptr.MetodoPuntero() // ✅ Directo
}

```

Ahora vamos a desarrollar un caso completo con la finalidad de que podamos tener claridad acerca del uso de structs.

```

Go

package main

import (
    "fmt"

```

```

        "strings"

        "time"
    )

// =====
// PASO 1: STRUCTS BÁSICOS
// =====

// Libro representa un libro en la biblioteca
type Libro struct {
    ID        int
    Titulo    string
    Autor     string
    ISBN      string
    Paginas   int
    Prestado  bool
}

// Usuario representa a un usuario de la biblioteca
type Usuario struct {
    ID        int
    Nombre    string
    Email     string
    Telefono  string
    Activo    bool
}

```

```
}
```

```
// Prestamo representa el préstamo de un libro
```

```
type Prestamo struct {
```

```
    ID          int
```

```
    LibroID     int
```

```
    UsuarioID   int
```

```
    FechaPrestamo time.Time
```

```
    FechaDevolucion time.Time
```

```
    Devuelto     bool
```

```
}
```

```
// =====
```

```
// PASO 2: MÉTODOS CON RECEPTOR DE VALOR
```

```
// (Solo para LEER información, no modifican)
```

```
// =====
```

```
// ObtenerInfo retorna información básica del libro
```

```
// Usa receptor de VALOR porque solo LEE, no modifica
```

```
func (l Libro) ObtenerInfo() string {
```

```
    estado := "Disponible"
```

```
    if l.Prestado {
```

```
        estado = "Prestado"
```

```
    }
```

```
    return fmt.Sprintf("[%d] %s por %s - %s", l.ID, l.Titulo, l.Autor, estado)
```



```

}

// EsPrestable verifica si el libro se puede prestar
// Usa receptor de VALOR porque solo LEE
func (l Libro) EsPrestable() bool {
    return !l.Prestado && l.Paginas > 0
}

// EsLibroGrande determina si es un libro extenso
// Usa receptor de VALOR porque solo LEE
func (l Libro) EsLibroGrande() bool {
    return l.Paginas > 300
}

// ObtenerResumen retorna un resumen del usuario
// Usa receptor de VALOR porque solo LEE
func (u Usuario) ObtenerResumen() string {
    estado := "Inactivo"
    if u.Activo {
        estado = "Activo"
    }
    return fmt.Sprintf("%s (%s) - %s", u.Nombre, u.Email, estado)
}

// PuedePrestar verifica si el usuario puede pedir préstamos

```

```

// Usa receptor de VALOR porque solo LEE
func (u Usuario) PuedePrestar() bool {
    return u.Activo && u.Email != "" && u.Nombre != ""
}

// =====

// PASO 3: MÉTODOS CON RECEPTOR DE PUNTERO
// (Para MODIFICAR el estado del struct)
// =====

// Prestar marca el libro como prestado
// Usa receptor de PUNTERO porque MODIFICA el estado
func (l *Libro) Prestar() error {
    if l.Prestado {
        return fmt.Errorf("el libro '%s' ya está prestado", l.Titulo)
    }

    if l.Paginas <= 0 {
        return fmt.Errorf("el libro '%s' no es válido", l.Titulo)
    }

    l.Prestado = true
    return nil
}

// Devolver marca el libro como devuelto
// Usa receptor de PUNTERO porque MODIFICA el estado

```

```
func (l *Libro) Devolver() error {
    if !l.Prestado {
        return fmt.Errorf("el libro '%s' no está prestado", l.Titulo)
    }

    l.Prestado = false
    return nil
}

// ActualizarInfo permite actualizar información del libro
// Usa receptor de PUNTERO porque MODIFICA el estado
func (l *Libro) ActualizarInfo(titulo, autor string, paginas int) error {
    if titulo == "" || autor == "" {
        return fmt.Errorf("título y autor no pueden estar vacíos")
    }

    if paginas <= 0 {
        return fmt.Errorf("número de páginas debe ser positivo")
    }

    l.Titulo = titulo
    l.Autor = autor
    l.Paginas = paginas
    return nil
}
```

```

// Activar activa la cuenta del usuario
// Usa receptor de PUNTERO porque MODIFICA el estado
func (u *Usuario) Activar() {
    u.Activo = true
}

// Desactivar desactiva la cuenta del usuario
// Usa receptor de PUNTERO porque MODIFICA el estado
func (u *Usuario) Desactivar() {
    u.Activo = false
}

// ActualizarContacto actualiza información de contacto
// Usa receptor de PUNTERO porque MODIFICA el estado
func (u *Usuario) ActualizarContacto(email, telefono string) error {
    if !strings.Contains(email, "@") {
        return fmt.Errorf("email inválido: %s", email)
    }

    u.Email = email
    u.Telefono = telefono
    return nil
}

// =====

```

```

// PASO 4: STRUCT PRINCIPAL CON COMPOSICIÓN

// =====

// Biblioteca es el struct principal que maneja todo el sistema
type Biblioteca struct {
    Nombre    string
    Direccion string
    Libros    []Libro
    Usuarios  []Usuario
    Prestamos []Prestamo
    proximoID int
}

// =====

// PASO 5: MÉTODOS AVANZADOS CON LÓGICA DE NEGOCIO

// =====

// NuevaBiblioteca es un constructor (patrón común en Go)
func NuevaBiblioteca(nombre, direccion string) *Biblioteca {
    return &Biblioteca{
        Nombre:    nombre,
        Direccion: direccion,
        Libros:     make([]Libro, 0),
        Usuarios:   make([]Usuario, 0),
        Prestamos: make([]Prestamo, 0),
    }
}

```

```

        proximoID: 1,
    }
}

// AgregarLibro añade un nuevo libro a la biblioteca
// Usa receptor de PUNTERO porque modifica el slice de libros
func (b *Biblioteca) AgregarLibro(titulo, autor, isbn string, paginas int)
(*Libro, error) {
    if titulo == "" || autor == "" {
        return nil, fmt.Errorf("título y autor son obligatorios")
    }

    // Verificar que no exista un libro con el mismo ISBN
    for _, libro := range b.Libros {
        if libro.ISBN == isbn && isbn != "" {
            return nil, fmt.Errorf("ya existe un libro con ISBN: %s",
isbn)
        }
    }

    libro := Libro{
        ID:      b.proximoID,
        Titulo:  titulo,
        Autor:   autor,
        ISBN:    isbn,
        Paginas: paginas,
    }
}

```

```

        Prestado: false,
    }

    b.Libros = append(b.Libros, libro)
    b.proximoID++

    return &libro, nil
}

// RegistrarUsuario registra un nuevo usuario
// Usa receptor de PUNTERO porque modifica el slice de usuarios
func (b *Biblioteca) RegistrarUsuario(nombre, email, telefono string)
(*Usuario, error) {
    if nombre == "" || email == "" {
        return nil, fmt.Errorf("nombre y email son obligatorios")
    }

    if !strings.Contains(email, "@") {
        return nil, fmt.Errorf("email inválido: %s", email)
    }

    // Verificar que no exista un usuario con el mismo email
    for _, usuario := range b.Usuarios {
        if usuario.Email == email {
            return nil, fmt.Errorf("ya existe un usuario con email:
%s", email)

```

```

        }
    }

    usuario := Usuario{
        ID:      b.proximoID,
        Nombre:  nombre,
        Email:   email,
        Telefono: telefono,
        Activo:  true,
    }

    b.Usuarios = append(b.Usuarios, usuario)
    b.proximoID++

    return &usuario, nil
}

// BuscarLibro busca un libro por ID
// Usa receptor de VALOR porque solo lee y retorna una copia
func (b Biblioteca) BuscarLibro(id int) *Libro {
    for i, libro := range b.Libros {
        if libro.ID == id {
            return &b.Libros[i] // Retorna puntero al libro original
        }
    }
}

```



```

        return nil
    }

    // BuscarUsuario busca un usuario por ID
    // Usa receptor de VALOR porque solo lee
    func (b Biblioteca) BuscarUsuario(id int) *Usuario {
        for i, usuario := range b.Usuarios {
            if usuario.ID == id {
                return &b.Usuarios[i] // Retorna puntero al usuario
            }
        }
        return nil
    }

    // PrestarLibro realiza el préstamo de un libro
    // Usa receptor de PUNTERO porque modifica múltiples estados
    func (b *Biblioteca) PrestarLibro(libroID, usuarioID int) error {
        // Buscar libro
        libro := b.BuscarLibro(libroID)
        if libro == nil {
            return fmt.Errorf("libro con ID %d no encontrado", libroID)
        }

        // Buscar usuario
        usuario := b.BuscarUsuario(usuarioID)

```

```

    if usuario == nil {
        return fmt.Errorf("usuario con ID %d no encontrado", usuarioID)
    }

    // Validar que el usuario pueda prestar

    if !usuario.PuedePrestar() {
        return fmt.Errorf("el usuario %s no puede realizar préstamos",
            usuario.Nombre)
    }

    // Validar que el libro se pueda prestar

    if !libro.EsPrestable() {
        return fmt.Errorf("el libro '%s' no se puede prestar",
            libro.Titulo)
    }

    // Realizar el préstamo

    if err := libro.Prestar(); err != nil {
        return err
    }

    // Registrar el préstamo

    prestamo := Prestamo{
        ID:          b.proximoID,
        LibroID:     libroID,
        UsuarioID:   usuarioID,
    }

```

```

        FechaPrestamo: time.Now(),
        FechaDevolucion: time.Now().AddDate(0, 0, 14), // 2 semanas
        Devuelto:      false,
    }

    b.Prestamos = append(b.Prestamos, prestamo)
    b.proximoID++

    return nil
}

// DevolverLibro procesa la devolución de un libro
// Usa receptor de PUNTERO porque modifica estados
func (b *Biblioteca) DevolverLibro(libroID int) error {
    // Buscar libro
    libro := b.BuscarLibro(libroID)
    if libro == nil {
        return fmt.Errorf("libro con ID %d no encontrado", libroID)
    }

    // Buscar préstamo activo
    var prestamoActivo *Prestamo
    for i := range b.Prestamos {
        if b.Prestamos[i].LibroID == libroID && !b.Prestamos[i].Devuelto {
            prestamoActivo = &b.Prestamos[i]
        }
    }
    if prestamoActivo == nil {
        return fmt.Errorf("no se encontró un préstamo activo para el libro con ID %d", libroID)
    }
    prestamoActivo.Devuelto = true
    return nil
}

```

```

        break
    }
}

if prestamoActivo == nil {
    return fmt.Errorf("no se encontró préstamo activo para el libro '%s'", libro.Titulo)
}

// Realizar la devolución
if err := libro.Devolver(); err != nil {
    return err
}

// Marcar préstamo como devuelto
prestamoActivo.Devuelto = true

return nil
}

// ObtenerEstadisticas retorna estadísticas de la biblioteca
// Usa receptor de VALOR porque solo lee información
func (b Biblioteca) ObtenerEstadisticas() string {
    totalLibros := len(b.Libros)
    librosPrestados := 0
    usuariosActivos := 0

```

```
prestamosActivos := 0

for _, libro := range b.Libros {
    if libro.Prestado {
        librosPrestados++
    }
}

for _, usuario := range b.Usuarios {
    if usuario.Activo {
        usuariosActivos++
    }
}

for _, prestamo := range b.Prestamos {
    if !prestamo.Devuelto {
        prestamosActivos++
    }
}

return fmt.Sprintf(`📊 Estadísticas de %s:
📖 Total de libros: %d
📖 Libros prestados: %d
📖 Libros disponibles: %d
👤 Usuarios activos: %d
```

```

📄 Préstamos activos: %d`,
        b.Nombre, totalLibros, librosPrestados,
        totalLibros-librosPrestados, usuariosActivos, prestamosActivos)
    }

// ListarLibrosDisponibles muestra todos los libros disponibles
// Usa receptor de VALOR porque solo lee
func (b Biblioteca) ListarLibrosDisponibles() {
    fmt.Println("\n📖 Libros Disponibles:")
    fmt.Println("=" + strings.Repeat("=", 50))

    disponibles := 0
    for _, libro := range b.Libros {
        if !libro.Prestado {
            fmt.Printf("  %s\n", libro.ObtenerInfo())
            if libro.EsLibroGrande() {
                fmt.Printf("    📖 Libro extenso (%d páginas)\n",
libro.Paginas)
            }
            disponibles++
        }
    }

    if disponibles == 0 {
        fmt.Println("  No hay libros disponibles")
    }
}

```

```

}

// =====

// FUNCIÓN PRINCIPAL DEMOSTRATIVA

// =====

func main() {
    fmt.Println("🏛️ SISTEMA DE BIBLIOTECA - DEMO PRÁCTICA")
    fmt.Println("=" + strings.Repeat("=", 50))

    // PASO 1: Crear biblioteca

    biblioteca := NuevaBiblioteca("Biblioteca Central", "Av. Principal 123")
    fmt.Printf("\n✅ Biblioteca creada: %s\n", biblioteca.Nombre)

    // PASO 2: Agregar libros

    fmt.Println("\n📖 Agregando libros...")

    libros := []struct {
        titulo, autor, isbn string
        paginas             int
    }{
        {"El Quijote", "Miguel de Cervantes", "978-84-376-0494-7", 863},
        {"Cien Años de Soledad", "Gabriel García Márquez",
"978-84-376-0495-4", 471},
        {"Go Programming", "Alan Donovan", "978-0-13-419044-0", 380},
        {"Clean Code", "Robert Martin", "978-0-13-235088-4", 464},
    }

```

```

    }

    for _, l := range libros {
        libro, err := biblioteca.AgregarLibro(l.titulo, l.autor, l.isbn,
l.paginas)

        if err != nil {
            fmt.Printf("❌ Error: %v\n", err)
        } else {
            fmt.Printf("✅ Agregado: %s\n", libro.ObtenerInfo())
        }
    }
}

// PASO 3: Registrar usuarios
fmt.Println("\n👤 Registrando usuarios...")

usuarios := []struct {
    nombre, email, telefono string
}{
    {"Ana García", "ana.garcia@email.com", "555-0101"},
    {"Carlos López", "carlos.lopez@email.com", "555-0102"},
    {"María Rodríguez", "maria.rodriguez@email.com", "555-0103"},
}

for _, u := range usuarios {
    usuario, err := biblioteca.RegistrarUsuario(u.nombre, u.email,
u.telefono)

```



```

        if err != nil {
            fmt.Printf("❌ Error: %v\n", err)
        } else {
            fmt.Printf("✅ Registrado: %s\n",
usuario.ObtenerResumen())
        }
    }

// PASO 4: Realizar préstamos
fmt.Println("\n📋 Realizando préstamos...")

prestamos := []struct {
    libroID, usuarioID int
}{}

{1, 1}, // Ana toma El Quijote
{3, 2}, // Carlos toma Go Programming
{2, 3}, // María toma Cien Años de Soledad
}

for _, p := range prestamos {
    err := biblioteca.PrestarLibro(p.libroID, p.usuarioID)
    if err != nil {
        fmt.Printf("❌ Error en préstamo: %v\n", err)
    } else {
        libro := biblioteca.BuscarLibro(p.libroID)
        usuario := biblioteca.BuscarUsuario(p.usuarioID)
    }
}

```

```

        fmt.Printf("  ✅ %s prestó '%s'\n", usuario.Nombre,
libro.Titulo)
    }
}

// PASO 5: Mostrar estado actual
biblioteca.ListarLibrosDisponibles()

// PASO 6: Devolver un libro
fmt.Println("\n🔄 Devolviendo libro...")
err := biblioteca.DevolverLibro(1) // Ana devuelve El Quijote
if err != nil {
    fmt.Printf("❌ Error en devolución: %v\n", err)
} else {
    fmt.Println("  ✅ El Quijote devuelto correctamente")
}

// PASO 7: Mostrar estadísticas finales
fmt.Println("\n" + biblioteca.ObtenerEstadisticas())

// PASO 8: Demostrar diferencia entre receptor de valor y puntero
fmt.Println("\n🔍 DEMO: Diferencia entre receptores")
fmt.Println("=" + strings.Repeat("=", 50))

libro := biblioteca.BuscarLibro(4) // Clean Code
fmt.Printf("Estado inicial: %s\n", libro.ObtenerInfo())

```

```

// Intentar prestar (modifica el struct)

err = libro.Prestar()

if err != nil {
    fmt.Printf("❌ Error: %v\n", err)
} else {
    fmt.Printf("Después del préstamo: %s\n", libro.ObtenerInfo())
}

// Verificar info (no modifica)

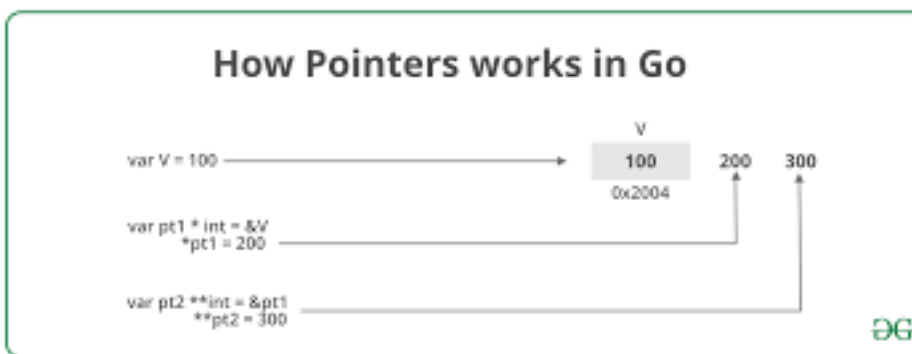
fmt.Printf("¿Es prestable?: %v\n", libro.EsPrestable())
fmt.Printf("¿Es libro grande?: %v\n", libro.EsLibroGrande())

fmt.Println("\n🎯 ¡Demo completada! Los estudiantes pueden ver:")
fmt.Println("    • Structs básicos y composición")
fmt.Println("    • Métodos con receptor de valor (lectura)")
fmt.Println("    • Métodos con receptor de puntero (modificación)")
fmt.Println("    • Validaciones y manejo de errores")
fmt.Println("    • Lógica de negocio completa")
}

```

## Punteros en GO

En Go, un puntero es una variable que almacena la dirección de memoria de otra variable. En lugar de contener un valor directo, «apunta» a la ubicación en memoria donde se almacena ese valor. Cuando definimos un puntero a puntero entonces el primer puntero se utiliza para almacenar la dirección del segundo puntero. Este concepto se denomina a veces Punteros Dobles



Conceptos clave de los punteros en Go:

- **Declaración:** Los punteros se declaran utilizando el símbolo asterisco (\*) seguido del tipo del valor al que apuntan. Por ejemplo, `var ptr *int` declara un puntero `ptr` que puede contener la dirección de memoria de una variable entera.
- **Tomar la dirección:** El símbolo ampersand (&) se utiliza para obtener la dirección de memoria de una variable. Por ejemplo, `ptr = &miVariable` asigna la dirección de memoria de `miVariable` al puntero `ptr`.
- **Desreferenciación:** El símbolo asterisco (\*) también se utiliza para «desreferenciar» un puntero, es decir, para acceder al valor almacenado en la dirección de memoria que contiene el puntero. Por ejemplo, `value := *ptr` recupera el valor entero de la dirección almacenada en `ptr` y lo asigna a `value`.
- **Sin aritmética de punteros:** A diferencia de otros lenguajes (como C/C++), Go no permite la aritmética de punteros. No puedes realizar operaciones como `ptr++` para moverte a la siguiente dirección de memoria.
- **La nueva función:** Go proporciona la función incorporada `new` para asignar memoria a una nueva variable de un tipo especificado y devolver un puntero a la misma. Por

ejemplo, `ptr := new(int)` crea una variable `int` (inicializada a su valor cero, 0 para `int`) y devuelve un puntero a la misma.

- **Casos de Uso:** Los punteros se utilizan comúnmente en Go para:
- **Eficiencia:** Pasar punteros a funciones en lugar de copiar grandes estructuras de datos (como structs) puede mejorar la per

Veamos la definición básica de un puntero con un ejemplo:

```
Go

var x int = 10

var ptr *int = &x

fmt.Println(x) // output: 10

fmt.Println(ptr) // output: 0xc0000140a8

fmt.Println(*ptr) // output: 10
```

En este ejemplo, declaramos una variable entera `x` y la inicializamos con el valor 10. También declaramos una variable puntero `ptr` de tipo `*int` y la inicializamos con la dirección de memoria de `x` utilizando el operador `&`. También declaramos una variable puntero `ptr` de tipo `*int` y la inicializamos con la dirección de memoria de `x` utilizando el operador `&`. A continuación, imprimimos el valor de `x`, la dirección de memoria de `x` almacenada en `ptr`, y el valor de `x` utilizando el operador `*`, que de referencia el puntero y nos da el valor almacenado en la dirección de memoria.

## Gestión de memoria con punteros

En Go, la gestión de memoria es automática y manejada por el recolector de basura. Sin embargo, Go también proporciona la capacidad de asignar memoria dinámicamente usando punteros. La asignación dinámica de memoria te permite crear estructuras de datos que pueden crecer y decrecer según sea necesario. He aquí un ejemplo:

Go

```
fmt.Println(ptr) // output: 0xc0000160c0  
fmt.Println(*ptr) // output: 0  
  
*ptr = 10  
fmt.Println(*ptr) // output: 10  
  
ptr = nil
```

En este ejemplo, declaramos una variable puntero `ptr` de tipo `*int` y utilizamos la función `new` para asignar memoria a un valor entero. A continuación, imprimimos la dirección de memoria almacenada en `ptr`, que es la dirección del bloque de memoria recién asignado. También imprimimos el valor de `*ptr`, que es el valor almacenado en la dirección de memoria, cuyo valor inicial es 0. A continuación, asignamos el valor 10 a la posición de memoria apuntada por `ptr` utilizando el operador `*`. Finalmente, establecemos la variable puntero a `nil`, lo que libera el bloque de memoria asignado por `new`.

## Punteros en argumentos de funciones

Uno de los usos principales de los punteros en Go es pasar datos por referencia a funciones. Cuando se pasa una variable a una función, se crea una copia de la variable. Sin embargo, para estructuras de datos grandes, esto puede suponer una sobrecarga de rendimiento. Los punteros permiten a las funciones trabajar directamente con los datos originales, evitando la duplicación innecesaria.

Go

```
package main
```

```
import "fmt"

func modifyValue(ptr *int) {
    *ptr = 100
}

func main() {
    var x int = 42

    fmt.Println("Before modification:", x)
    modifyValue(&x)
    fmt.Println("After modification:", x)
}
```

En el código anterior, la función `modifyValue` toma como argumento un puntero a un entero. Al pasar la dirección de la variable `x` mediante `&x`, la función puede actualizar directamente el valor original. Este enfoque es particularmente beneficioso cuando se trabaja con funciones que necesitan modificar el estado de variables fuera de su ámbito.

## Punteros y estructuras de datos

El verdadero poder de los punteros en Go se hace evidente cuando se trata con estructuras de datos complejas, como los structs. exploremos cómo se pueden emplear los punteros con una estructura `Persona`:

```
Go

package main
```

```
import "fmt"

type Person struct {
    Name string
    Age  int
}

func main() {
    p := Person{Name: "John", Age: 30}
    fmt.Println("Before modification:", p)
    modifyPerson(&p)
    fmt.Println("After modification:", p)
}

func modifyPerson(ptr *Person) {
    ptr.Age = 31
    ptr.Name = "John Doe"
}
```

En el código anterior, la función `modifyPerson` toma un puntero a una estructura `Persona`. Al utilizar un puntero, la función puede actualizar directamente los campos de la variable `Persona` original. Esto es particularmente útil cuando se trabaja con estructuras de datos grandes y complejas, ya que elimina la necesidad de pasar copias de toda la estructura.



—

## Interfaces en GO

En Go, una interfaz es un tipo que define un conjunto de firmas de métodos. Cualquier tipo que implemente todos los métodos definidos en la interfaz se dice que satisface la interfaz. Esto le permite escribir código que es genérico y se puede utilizar con cualquier tipo que satisfaga la interfaz. Una **interfaz** en Go define un conjunto de métodos que un tipo debe implementar. A diferencia de otros lenguajes, Go usa **duck typing**: "Si camina como un pato y hace cuac como un pato, es un pato".

Las interfaces en Golang son bastante diferentes de las de otros lenguajes. En Go, la interfaz es un tipo personalizado que se utiliza para especificar un conjunto de una o más firmas de métodos.

Un valor de tipo interfaz puede contener cualquier valor que implemente esos métodos. La interfaz es abstracta, lo que significa que no podemos crear instancias de una interfaz. Lo que hacemos es crear instancias de valores concretos que implementan esa interfaz.

```
type myInterface interface {  
    methodWithoutParameters()  
    methodWithParameter(float64)  
    methodWithReturnValue() string  
}
```

The diagram shows a Go interface definition with several annotations:

- "interface" keyword**: Points to the `interface` keyword in `type myInterface interface {`.
- Method name**: Three labels pointing to the method names `methodWithoutParameters`, `methodWithParameter`, and `methodWithReturnValue`.
- Type of parameter**: Points to the `float64` parameter in `methodWithParameter(float64)`.
- Type of return value**: Points to the `string` return type in `methodWithReturnValue() string`.

Veámoslo con un ejemplo

```
Go  
  
package main
```

```
import "fmt"

type Shape interface {
    Area() float64
    Perimeter() float64
}

type Rect struct {
    width  float64
    height float64
}

func (r Rect) Area() float64 {
    return r.width * r.height
}

func (r Rect) Perimeter() float64 {
    return 2 * (r.width + r.height)
}

func main() {
    var s Shape
    s = Rect{5.0, 4.0}
    r := Rect{5.0, 4.0}
```

```
fmt.Printf("type of s is %T\n", s)

fmt.Printf("value of s is %v\n", s)

fmt.Println("area of rectangle s", s.Area())

fmt.Println("s == r is", s == r)

}
```

Deberíamos tener este resultado

```
Go
type of s is main.Rect
value of s is {5 4}
area of rectangle s 20
s == r is true
```

Veamos una breve explicación del código:

- Hemos creado la interfaz Shape y el tipo struct Rect en el programa anterior. Después hemos definido métodos para el tipo Rect, cómo Area y Perimeter, por lo que Rect implementa estos métodos.
- El tipo struct Rect implementa la interfaz Shape porque la interfaz Shape define estos métodos. La interfaz Shape está siendo implementada por Rect automáticamente porque no la hemos forzado. Como resultado, las interfaces en Go se dicen implementadas implícitamente.
- Una variable de un tipo que implementa una interfaz también puede ser referida como el tipo de una interfaz. Definiendo una interfaz nil de tipo Shape y asignando una struct de tipo Rect, podemos confirmarlo.

Vamos a desarrollar un caso completo para lograr entender el uso de interfaces

```
Go

package main

import (
    "fmt"
    "errors"
    "strings"
    "time"
    "encoding/json"
)

// =====
// PASO 1: INTERFACES BÁSICAS
// =====

// Notificador define la funcionalidad básica de envío
type Notificador interface {
    EnviarNotificacion(destinatario, mensaje string) error
}

// ValidadorMensaje valida contenido antes del envío
type ValidadorMensaje interface {
    ValidarMensaje(mensaje string) error
    ValidarDestinatario(destinatario string) error
}
```

```
// Rastreador permite hacer seguimiento de notificaciones
```

```
type Rastreador interface {  
    ObtenerEstado(id string) (string, error)  
    ObtenerEstadisticas() map[string]int  
}
```

```
// Logger registra eventos del sistema
```

```
type Logger interface {  
    Log(nivel, mensaje string)  
    LogError(error)  
    LogInfo(string)  
}
```

```
// =====
```

```
// PASO 2: INTERFACES COMPUESTAS
```

```
// =====
```

```
// NotificadorCompleto combina funcionalidades básicas
```

```
type NotificadorCompleto interface {  
    Notificador  
    ValidadorMensaje  
}
```

```
// NotificadorAvanzado incluye todas las funcionalidades
```

```

type NotificadorAvanzado interface {
    Notificador
    ValidadorMensaje
    Rastreador
    Logger
}

// =====
// PASO 3: STRUCTS Y TIPOS DE DATOS
// =====

type TipoNotificacion string

const (
    Email TipoNotificacion = "email"
    SMS   TipoNotificacion = "sms"
    Push  TipoNotificacion = "push"
    Slack TipoNotificacion = "slack"
)

type EstadoNotificacion string

const (
    Pendiente EstadoNotificacion = "pendiente"
    Enviada   EstadoNotificacion = "enviada"
)

```

```
        Fallida      EstadoNotificacion = "fallida"
        Entregada EstadoNotificacion = "entregada"
    )
```

```
type RegistroNotificacion struct {
    ID          string
    Tipo        TipoNotificacion
    Destinatario string
    Mensaje     string
    Estado      EstadoNotificacion
    Timestamp   time.Time
    Intentos    int
    Error       string
}
```

```
type ConfiguracionNotificacion struct {
    MaxIntentos    int
    TimeoutSegundos int
    ReintentoAuto  bool
}
```

```
// =====
// PASO 4: IMPLEMENTACIONES CONCRETAS
// =====
```



```

// EmailNotificador - Implementa múltiples interfaces
type EmailNotificador struct {
    servidor      string
    puerto        int
    usuario       string
    password      string
    configuracion ConfiguracionNotificacion
    registros     map[string]*RegistroNotificacion
}

// Constructor para EmailNotificador
func NuevoEmailNotificador(servidor string, puerto int, usuario, password
string) *EmailNotificador {
    return &EmailNotificador{
        servidor:  servidor,
        puerto:    puerto,
        usuario:   usuario,
        password:  password,
        configuracion: ConfiguracionNotificacion{
            MaxIntentos:    3,
            TimeoutSegundos: 30,
            ReintentoAuto:   true,
        },
        registros: make(map[string]*RegistroNotificacion),
    }
}

```

```

// Implementa Notificador

func (e *EmailNotificador) EnviarNotificacion(destinatario, mensaje string)
error {

    // Validar antes de enviar

    if err := e.ValidarDestinatario(destinatario); err != nil {

        return err

    }

    if err := e.ValidarMensaje(mensaje); err != nil {

        return err

    }

    // Crear registro

    id := fmt.Sprintf("email_%d", time.Now().UnixNano())

    registro := &RegistroNotificacion{

        ID:          id,

        Tipo:         Email,

        Destinatario: destinatario,

        Mensaje:      mensaje,

        Estado:       Pendiente,

        Timestamp:    time.Now(),

        Intentos:     1,

    }

    e.registros[id] = registro

```

```

// Simular envío de email

e.LogInfo(fmt.Sprintf("Enviando email a %s", destinatario))

time.Sleep(100 * time.Millisecond) // Simular latencia

// Simular éxito/fallo (90% éxito)
if time.Now().UnixNano()%10 == 0 {

    registro.Estado = Fallida

    registro.Error = "Servidor SMTP no disponible"

    e.LogError(errors.New(registro.Error))

    return errors.New("fallo al enviar email")

}

registro.Estado = Enviada

e.LogInfo(fmt.Sprintf("Email enviado exitosamente: %s", id))

return nil

}

// Implementa ValidadorMensaje

func (e *EmailNotificador) ValidarMensaje(mensaje string) error {

    if len(mensaje) == 0 {

        return errors.New("mensaje no puede estar vacío")

    }

    if len(mensaje) > 1000 {

        return errors.New("mensaje muy largo (máximo 1000 caracteres)")

    }

}

```

```

        return nil
    }

func (e *EmailNotificador) ValidarDestinatario(destinatario string) error {
    if !strings.Contains(destinatario, "@") {
        return errors.New("email inválido: debe contener @")
    }
    if !strings.Contains(destinatario, ".") {
        return errors.New("email inválido: debe contener dominio")
    }
    return nil
}

// Implementa Rastreador
func (e *EmailNotificador) ObtenerEstado(id string) (string, error) {
    if registro, existe := e.registros[id]; existe {
        return string(registro.Estado), nil
    }
    return "", errors.New("notificación no encontrada")
}

func (e *EmailNotificador) ObtenerEstadisticas() map[string]int {
    stats := map[string]int{
        "total": 0,
        "enviadas": 0,
    }

```

```

        "fallidas": 0,

        "pendientes": 0,
    }

    for _, registro := range e.registros {
        stats["total"]++

        switch registro.Estado {
        case Enviada:
            stats["enviadas"]++
        case Fallida:
            stats["fallidas"]++
        case Pendiente:
            stats["pendientes"]++
        }
    }

    return stats
}

// Implementa Logger
func (e *EmailNotificador) Log(nivel, mensaje string) {
    timestamp := time.Now().Format("2006-01-02 15:04:05")

    fmt.Printf("[%s] EMAIL [%s]: %s\n", timestamp, nivel, mensaje)
}

```

```

func (e *EmailNotificador) LogError(err error) {
    e.Log("ERROR", err.Error())
}

func (e *EmailNotificador) LogInfo(mensaje string) {
    e.Log("INFO", mensaje)
}

// =====

// SMSNotificador - Otra implementación
type SMSNotificador struct {
    apiKey    string
    proveedor string
    registros map[string]*RegistroNotificacion
}

func NuevoSMSNotificador(apiKey, proveedor string) *SMSNotificador {
    return &SMSNotificador{
        apiKey:    apiKey,
        proveedor: proveedor,
        registros: make(map[string]*RegistroNotificacion),
    }
}

```

```

// Implementa Notificador

func (s *SMSNotificador) EnviarNotificacion(destinatario, mensaje string) error
{
    if err := s.ValidarDestinatario(destinatario); err != nil {
        return err
    }

    if err := s.ValidarMensaje(mensaje); err != nil {
        return err
    }

    id := fmt.Sprintf("sms_%d", time.Now().UnixNano())
    registro := &RegistroNotificacion{
        ID:          id,
        Tipo:         SMS,
        Destinatario: destinatario,
        Mensaje:      mensaje,
        Estado:       Pendiente,
        Timestamp:    time.Now(),
        Intentos:     1,
    }

    s.registros[id] = registro

    s.LogInfo(fmt.Sprintf("Enviando SMS a %s via %s", destinatario,
s.proveedor))

    time.Sleep(50 * time.Millisecond) // SMS más rápido que email

```

```

// SMS más confiable (95% éxito)
if time.Now().UnixNano()%20 == 0 {
    registro.Estado = Fallida
    registro.Error = "Número no válido"
    s.LogError(errors.New(registro.Error))
    return errors.New("fallo al enviar SMS")
}

registro.Estado = Enviada
s.LogInfo(fmt.Sprintf("SMS enviado exitosamente: %s", id))
return nil
}

// Implementa ValidadorMensaje
func (s *SMSNotificador) ValidarMensaje(mensaje string) error {
    if len(mensaje) == 0 {
        return errors.New("mensaje SMS no puede estar vacío")
    }
    if len(mensaje) > 160 {
        return errors.New("mensaje SMS muy largo (máximo 160 caracteres)")
    }
    return nil
}

```



```

func (s *SMSNotificador) ValidarDestinatario(destinatario string) error {
    if len(destinatario) < 10 {
        return errors.New("número de teléfono muy corto")
    }

    if !strings.HasPrefix(destinatario, "+") &&
!strings.HasPrefix(destinatario, "0") {
        return errors.New("número debe empezar con + o 0")
    }

    return nil
}

// Implementa Rastreador
func (s *SMSNotificador) ObtenerEstado(id string) (string, error) {
    if registro, existe := s.registros[id]; existe {
        return string(registro.Estado), nil
    }

    return "", errors.New("SMS no encontrado")
}

func (s *SMSNotificador) ObtenerEstadisticas() map[string]int {
    stats := map[string]int{
        "total":    0,
        "enviados": 0,
        "fallidos": 0,
        "pendientes": 0,
    }
}

```

```

    for _, registro := range s.registros {
        stats["total"]++

        switch registro.Estado {
            case Enviada:
                stats["enviados"]++

            case Fallida:
                stats["fallidos"]++

            case Pendiente:
                stats["pendientes"]++
        }
    }

    return stats
}

// Implementa Logger
func (s *SMSNotificador) Log(nivel, mensaje string) {
    timestamp := time.Now().Format("2006-01-02 15:04:05")
    fmt.Printf("[%s] SMS [%s]: %s\n", timestamp, nivel, mensaje)
}

func (s *SMSNotificador) LogError(err error) {
    s.Log("ERROR", err.Error())
}

```

```

func (s *SMSNotificador) LogInfo(mensaje string) {
    s.Log("INFO", mensaje)
}

// =====

// SlackNotificador - Implementación más simple
type SlackNotificador struct {
    webhook string
    canal   string
}

func NuevoSlackNotificador(webhook, canal string) *SlackNotificador {
    return &SlackNotificador{
        webhook: webhook,
        canal:   canal,
    }
}

// Solo implementa Notificador (implementación mínima)
func (sl *SlackNotificador) EnviarNotificacion(destinatario, mensaje string)
error {
    fmt.Printf("🔔 Slack -> Canal: %s | Usuario: %s | Mensaje: %s\n",
        sl.canal, destinatario, mensaje)
}

```

```

        // Simular envío instantáneo
        time.Sleep(10 * time.Millisecond)

        return nil
    }

    // =====
    // PASO 5: SERVICIO PRINCIPAL
    // =====

type ServicioNotificaciones struct {
    notificadores []Notificador
    logger         Logger
}

func NuevoServicioNotificaciones() *ServicioNotificaciones {
    return &ServicioNotificaciones{
        notificadores: make([]Notificador, 0),
    }
}

func (sn *ServicioNotificaciones) AgregarNotificador(notificador Notificador) {
    sn.notificadores = append(sn.notificadores, notificador)

    if sn.logger != nil {
        sn.logger.LogInfo(fmt.Sprintf("Notificador agregado: %T",
notificador))
    }
}

```

```

    }
}

func (sn *ServicioNotificaciones) EstablecerLogger(logger Logger) {
    sn.logger = logger
}

// Enviar a todos los notificadores

func (sn *ServicioNotificaciones) EnviarATodos(destinatario, mensaje string)
map[string]error {
    resultados := make(map[string]error)

    if sn.logger != nil {
        sn.logger.LogInfo(fmt.Sprintf("Enviando a %d notificadores",
len(sn.notificadores)))
    }

    for _, notificador := range sn.notificadores {
        tipoNotificador := fmt.Sprintf("%T", notificador)
        err := notificador.EnviarNotificacion(destinatario, mensaje)
        resultados[tipoNotificador] = err

        if sn.logger != nil {
            if err != nil {
                sn.logger.LogError(fmt.Errorf("%s falló: %v",
tipoNotificador, err))
            }
        }
    }
}

```

```

        } else {
            sn.logger.LogInfo(fmt.Sprintf("%s éxito",
tipoNotificador))
        }
    }
}

return resultados
}

// Enviar solo a notificadores que implementen ValidadorMensaje
func (sn *ServicioNotificaciones) EnviarConValidacion(destinatario, mensaje
string) map[string]error {
    resultados := make(map[string]error)

    for _, notificador := range sn.notificadores {
        tipoNotificador := fmt.Sprintf("%T", notificador)

        // Type assertion para verificar si implementa ValidadorMensaje
        if validador, implementa := notificador.(ValidadorMensaje);
implementa {

            // Validar antes de enviar

            if err := validador.ValidarMensaje(mensaje); err != nil {
                resultados[tipoNotificador] = fmt.Errorf("validación
falló: %v", err)

                continue
            }

```

```

        if err := validador.ValidarDestinatario(destinatario); err
!= nil {

            resultados[tipoNotificador] =
fmt.Errorf("destinatario inválido: %v", err)

            continue

        }

    }

    // Enviar notificación

    err := notificador.EnviarNotificacion(destinatario, mensaje)

    resultados[tipoNotificador] = err

}

return resultados
}

// =====
// PASO 6: FUNCIONES DE UTILIDAD
// =====

// Función que acepta cualquier Notificador
func ProbarNotificador(n Notificador, destinatario, mensaje string) {

    fmt.Printf("\n🔧 Probando %T:\n", n)

    fmt.Println("    Enviando:", mensaje)

    err := n.EnviarNotificacion(destinatario, mensaje)

```

```

    if err != nil {
        fmt.Printf("    ✖ Error: %v\n", err)
    } else {
        fmt.Printf("    ✔ Enviado correctamente\n")
    }
}

// Función que verifica capacidades usando type assertions
func Analizar CapacidadesNotificador(n Notificador) {
    fmt.Printf("\n🔍 Analizando capacidades de %T:\n", n)

    capacidades := []string{}

    // Verificar cada interface
    if _, implementa := n.(Notificador); implementa {
        capacidades = append(capacidades, "✔ Notificador (envío básico)")
    }

    if _, implementa := n.(ValidadorMensaje); implementa {
        capacidades = append(capacidades, "✔ ValidadorMensaje (validación)")
    }

    if _, implementa := n.(Rastreador); implementa {
        capacidades = append(capacidades, "✔ Rastreador (seguimiento)")
    }
}

```



```

        if _, implementa := n.(Logger); implementa {
            capacidades = append(capacidades, "✅ Logger (registro de
eventos)")
        }

        if _, implementa := n.(NotificadorCompleto); implementa {
            capacidades = append(capacidades, "📧 NotificadorCompleto")
        }

        if _, implementa := n.(NotificadorAvanzado); implementa {
            capacidades = append(capacidades, "🚀 NotificadorAvanzado")
        }

        for _, capacidad := range capacidades {
            fmt.Printf("    %s\n", capacidad)
        }
    }

    // Type switch para manejar diferentes tipos
    func ProcesarNotificadorPorTipo(n Notificador, destinatario, mensaje string) {
        switch notificador := n.(type) {
        case *EmailNotificador:
            fmt.Println("📧 Procesando como EmailNotificador...")
            fmt.Printf("    Servidor: %s:%d\n", notificador.servidor,
notificador.puerto)

```

```

        notificador.EnviaNotificacion(destinatario, mensaje)

    case *SMSNotificador:

        fmt.Println("📱 Procesando como SMSNotificador...")

        fmt.Printf("    Proveedor: %s\n", notificador.proveedor)

        notificador.EnviaNotificacion(destinatario, mensaje)

    case *SlackNotificador:

        fmt.Println("💬 Procesando como SlackNotificador...")

        fmt.Printf("    Canal: %s\n", notificador.canal)

        notificador.EnviaNotificacion(destinatario, mensaje)

    default:

        fmt.Printf("❓ Tipo desconocido: %T\n", notificador)

        notificador.EnviaNotificacion(destinatario, mensaje)
    }
}

// =====
// PASO 7: FUNCIÓN PRINCIPAL DEMOSTRATIVA
// =====

func main() {

    fmt.Println("🔔 SISTEMA DE NOTIFICACIONES - INTERFACES EN ACCIÓN")

    fmt.Println("=" + strings.Repeat("=", 60))

```

```

// Crear diferentes notificadoros

email := NuevoEmailNotificador("smtp.gmail.com", 587, "app@empresa.com",
"password")

sms := NuevoSMSNotificador("api-key-123", "Twilio")

slack := NuevoSlackNotificador("https://hooks.slack.com/...", "#general")


// Crear servicio principal

servicio := NuevoServicioNotificaciones()

servicio.EstablecerLogger(email) // Email también funciona como logger


// Agregar notificadoros

servicio.AgregarNotificador(email)

servicio.AgregarNotificador(sms)

servicio.AgregarNotificador(slack)


fmt.Println("\n📄 1. POLIMORFISMO BÁSICO:")

fmt.Println(strings.Repeat("-", 40))


// Todos son tratados como Notificador

notificadores := []Notificador{email, sms, slack}


for _, n := range notificadores {
    ProbarNotificador(n, "usuario@ejemplo.com", "¡Hola desde Go!")
}

```

```

fmt.Println("\n📄 2. TYPE ASSERTIONS Y CAPACIDADES:")
fmt.Println(strings.Repeat("-", 40))

// Analizar capacidades de cada notificador
for _, n := range notificadores {
    Analizar CapacidadesNotificador(n)
}

fmt.Println("\n📄 3. TYPE SWITCH EN ACCIÓN:")
fmt.Println(strings.Repeat("-", 40))

// Usar type switch para lógica específica por tipo
for _, n := range notificadores {
    ProcesarNotificadorPorTipo(n, "+54911234567", "Mensaje tipo
específico")
    fmt.Println()
}

fmt.Println("\n📄 4. INTERFACES COMPUESTAS:")
fmt.Println(strings.Repeat("-", 40))

// Verificar interfaces compuestas
verificarInterfaceCompuesta := func(n Notificador) {
    nombre := fmt.Sprintf("%T", n)

    if completo, esCompleto := n.(NotificadorCompleto); esCompleto {

```

```

nombre)                fmt.Printf("✅ %s implementa NotificadorCompleto\n",

                        // Puede usar todas las funciones de NotificadorCompleto
                        completo.ValidarMensaje("test")
                        completo.EnviarNotificacion("test@test.com", "test")
                    } else {
nombre)                fmt.Printf("❌ %s NO implementa NotificadorCompleto\n",
                    }

                        if avanzado, esAvanzado := n.(NotificadorAvanzado); esAvanzado {
nombre)                fmt.Printf("🚀 %s implementa NotificadorAvanzado\n",
                        stats := avanzado.ObtenerEstadisticas()
                        fmt.Printf("    Estadísticas: %v\n", stats)
                    } else {
nombre)                fmt.Printf("⚠️ %s NO implementa NotificadorAvanzado\n",
                    }
                        fmt.Println()
                    }

for _, n := range notificadores {
    verificarInterfaceCompuesta(n)
}

fmt.Println("\n📋 5. SERVICIO CON MÚLTIPLES NOTIFICADORES:")

```

```

fmt.Println(strings.Repeat("-", 40))

// Enviar a todos
fmt.Println("🔴 Enviando a TODOS los notificadoros:")

resultados := servicio.EnviarATodos("admin@empresa.com", "Sistema
iniciado correctamente")

for tipo, err := range resultados {
    if err != nil {
        fmt.Printf("  ❌ %s: %v\n", tipo, err)
    } else {
        fmt.Printf("  ✅ %s: Éxito\n", tipo)
    }
}

fmt.Println("\n🔴 Enviando CON validación:")

resultados = servicio.EnviarConValidacion("usuario@empresa.com", "Mensaje
validado")

for tipo, err := range resultados {
    if err != nil {
        fmt.Printf("  ❌ %s: %v\n", tipo, err)
    } else {
        fmt.Printf("  ✅ %s: Éxito\n", tipo)
    }
}

```

```

fmt.Println("\n📄 6. ESTADÍSTICAS Y RASTREABILIDAD:")

fmt.Println(strings.Repeat("-", 40))

// Mostrar estadísticas solo de notificadores que implementan Rastreador
for _, n := range notificadores {
    if rastreador, implementa := n.(Rastreador); implementa {
        nombre := fmt.Sprintf("%T", n)
        stats := rastreador.ObtenerEstadisticas()
        fmt.Printf("📊 Estadísticas de %s:\n", nombre)

        statsJSON, _ := json.MarshalIndent(stats, "  ", "  ")
        fmt.Printf("    %s\n\n", string(statsJSON))
    }
}

fmt.Println("🎯 CONCEPTOS DEMOSTRADOS:")

fmt.Println(strings.Repeat("-", 40))

conceptos := []string{
    "✅ Definición de interfaces simples y compuestas",
    "✅ Implementación implícita de interfaces",
    "✅ Polimorfismo con múltiples implementaciones",
    "✅ Type assertions para verificar capacidades",
    "✅ Type switches para lógica específica por tipo",
    "✅ Composición de interfaces",
}

```

```
        "✅ Interfaces como contratos flexibles",
        "✅ Uso práctico en arquitectura de servicios",
    }

    for _, concepto := range conceptos {
        fmt.Printf("    %s\n", concepto)
    }

    fmt.Println("\n🎉 ¡Ejemplo completado!")
}
```



## Estructura de un proyecto en GO

La organización del código de un **proyecto es un problema en constante evolución**. Como dicen todos los desarrolladores sabios, «siempre depende de las necesidades». Pero seguir una estructura estándar ayudará a mantener la base de código más limpia y mejorará la productividad del equipo. Go es un lenguaje **minimalista y pragmático**, lo que se refleja en su estructura de proyecto. No hay una única forma de organizar un proyecto, pero sí hay una **convención bien aceptada** basada en el repositorio [golang-standards/project-layout](https://golang-standards/project-layout) que muchos equipos siguen.

```
mi_proyecto/
├── go.mod           # Declaración del módulo (import path raíz del proyecto)
├── go.sum           # Checksum de dependencias (autogenerado)
├── main.go          # Punto de entrada principal de la aplicación
├── cmd/             # Comandos principales de la app (CLI/API/etc)
│   └── app/         # cmd/app/main.go -> punto de entrada real
├── internal/        # Lógica interna que no debe ser usada fuera del proyecto
│   ├── usuario/     # Lógica de dominio relacionada a "usuarios"
│   │   ├── service.go
│   │   └── repository.go
│   └── auth/
│       └── token.go
├── pkg/             # Paquetes reutilizables por otros proyectos
│   └── logger/
│       └── logger.go
├── api/             # Definiciones de la API REST, JSON-RPC, GraphQL, etc
│   └── usuario_handler.go
├── models/          # Definición de los modelos (structs de dominio)
│   └── usuario.go
├── configs/         # Archivos de configuración YAML, JSON, TOML, etc.
│   └── config.yaml
├── migrations/      # Archivos SQL para migración de base de datos
│   └── 001_init.sql
├── scripts/         # Scripts utilitarios (Bash, Python, Go tools, etc.)
├── test/            # Archivos de test separados (con datos de test)
│   └── usuario_test.go
└── README.md        # Documentación inicial del proyecto
```

Ahora vamos a revisar a detalle los componentes clave de la estructura planteada

## 1. go.mod y go.sum

**go.mod** define el nombre del módulo y gestiona las dependencias. **El archivo go.mod es la raíz de la gestión de dependencias en GoLang.** Todos los módulos que se necesitan o se van a utilizar en el proyecto se mantienen en el archivo go.mod. Para todos los paquetes que vayamos a importar/utilizar en nuestro proyecto, creará una entrada de esos módulos en go.mod. Tener un archivo go mod ahorra el esfuerzo de ejecutar el comando go get para cada módulo dependiente para ejecutar el proyecto con éxito.

Se puede generar con la siguiente instrucción:

go mod init - crea un nuevo módulo, inicializando el fichero go.mod que describe el módulo. Al principio, sólo añadirá la ruta del módulo y la versión de go en el archivo go mod.

Shell

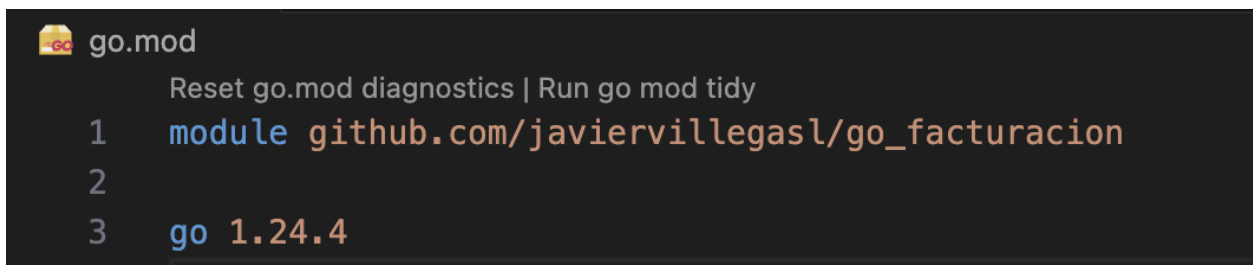
```
go mod init github.com/<usuario>/<nombre_proyecto>

##

go mod init github.com/javiervillegasl/go_facturacion


go: creating new go.mod: module github.com/javiervillegasl/go_facturacion
```

Ahora si abrimos nuestra solución en VSCode deberíamos observar lo siguiente:



```
go.mod
Reset go.mod diagnostics | Run go mod tidy
1 module github.com/javiervillegasl/go_facturacion
2
3 go 1.24.4
```

Después de ejecutar cualquier comando de construcción de paquetes como go build, go test por primera vez, se instalarán todos los paquetes con versiones específicas, es decir, cuáles son las últimas en ese momento.



También creará un archivo `go.sum` que mantendrá la suma de comprobación para que cuando ejecute el proyecto de nuevo no instale todos los paquetes de nuevo. Pero utiliza la caché que se almacena dentro del directorio `$GOPATH/pkg/mod` (directorio de caché de módulos).

**`go.sum`** es generado automáticamente y guarda los hashes de verificación. Además, no tiene que editar ni modificar.

## 2. `main.go` o `cmd/`

En el lenguaje Go, el paquete `main` es un paquete especial que se utiliza con los programas que son ejecutables y este paquete contiene la función `main()`. La función `main()` es un tipo especial de función y es el punto de entrada de los programas ejecutables. No toma ningún argumento ni devuelve nada. Go llama automáticamente a la función `main()`, por lo que no hay necesidad de llamar a la función `main()` explícitamente y cada programa ejecutable debe contener un único paquete `main` y la función `main()`.

- Para proyectos sencillos, `main.go` puede ser el punto de entrada.
- Para proyectos grandes, se recomienda mover el código de entrada a `cmd/app/main.go`.

## 3. `internal`

Contiene lógica **interna y privada** del proyecto. Es una convención oficial en Go: **cualquier paquete dentro de `internal` no puede ser importado fuera del módulo**.

**Ejemplo:**

`internal/usuario/service.go`

```
Go

package usuario

import "fmt"
```

```
func Ejecutar() {  
    fmt.Println("Servicio de usuarios ejecutándose...")  
}
```

#### 4. pkg/

Contiene paquetes que **pueden ser reutilizados por otros proyectos**, como logger, utils, validator.

**Ejemplo: pkg/logger/logger.go**

```
Go  
  
package logger  
  
import "log"  
  
func Info(msg string) {  
    log.Println("[INFO]", msg)  
}
```

## 5. models/

Aquí colocamos los **structs que representan entidades** del dominio del negocio.

**Ejemplo: models/usuario.go**

```
Go

package models

type Usuario struct {
    ID        int
    Nombre    string
    Correo    string
    Activo    bool
}
```

## 6. api/ o handlers/

Contiene los controladores que **exponen las funcionalidades**, por ejemplo, handlers HTTP, controladores CLI, etc.

**Ejemplo de handler básico:**

```
Go

package api
```

```
import (  
    "encoding/json"  
    "net/http"  
    "mi_proyecto/internal/usuario"  
)  
  
func UsuarioHandler(w http.ResponseWriter, r *http.Request) {  
    u := usuario.Obtener()  
    json.NewEncoder(w).Encode(u)  
}
```

## Flujo General de Ejecución

1. main.go o cmd/app/main.go arranca la aplicación.
2. Se inicializan servicios desde internal/ o pkg/.
3. Se usan modelos desde models/.
4. Se exponen endpoints desde api/ o handlers/.

## Funciones en Go

.En Go, una función es un bloque de código que realiza una tarea específica y puede ser llamada desde otras partes del programa. Las funciones se definen usando la palabra clave `func`, seguida del nombre de la función, una lista de parámetros (opcional), el tipo de retorno (opcional) y el cuerpo de la función encerrado entre llaves `{}`

### Definir funciones en Golang

Para comenzar lo primero que tendremos que hacer es crear un archivo `main.go` con su correspondiente package y la función `main` o función principal.

```
package main
import "fmt"
func main() {
}
```

### Declarando Funciones en Golang

En go las funciones se declaran anteponiendo la palabra `func` al nombre de la función. De la siguiente manera:

Go

```
func borrarRoot() { }
```

Recuerda que al ser un lenguaje compilado, go requiere que especifiques el tipo de dato en los argumentos.

Go

```
func borrarRoot(argumento int, otroArgumento int) { }
```

Aquí, una particularidad, si todos los argumentos son del mismo tipo podemos ahorrarnos una palabra omitiendo el primer tipo, en este caso int.

Go

```
func borrarRoot(argumento, otroArgumento int) { }
```

## Return en GO

Como en casi todos los lenguajes usamos la palabra return en una función para retornar un valor. Una función no requiere que retorne nada forzosamente, y no necesitas especificar un retorno, como si harías en C++ y otros lenguajes similares.

Por otro lado, si tu función sí cuenta con un return, requieren especificar el tipo de dato a retornar, colocándolo después de los argumentos.

Go

```
func RetornaUno(argumento, otroArgumento int) int{  
    return 1  
}
```

Así mismo, podemos retornar dos valores, como si de una tupla se tratara.

Go

```
package services
```

```
import (
```



```

    "errors"

    "fmt"

    "strconv"

    "strings"

    "time"
)

// Ejemplo 1: Función que puede fallar
func dividir(a, b float64) (float64, error) {
    if b == 0 {
        return 0, errors.New("división por cero no permitida")
    }

    return a / b, nil
}

```

En Go, las funciones pueden retornar múltiples valores, y estos valores pueden tener nombres (named return values). Cuando se usan nombres, los valores de retorno se tratan como variables declaradas al principio de la función y se inicializan con el valor cero de su tipo. Esto facilita la documentación de los valores de retorno y permite una sintaxis de retorno más limpia, utilizando `return` sin argumentos para devolver los valores con nombre

```

Go

func analizarTexto(texto string) (palabras int, caracteres int, lineas int) {
    caracteres = len(texto)

    lineas = strings.Count(texto, "\n") + 1
}

```

```

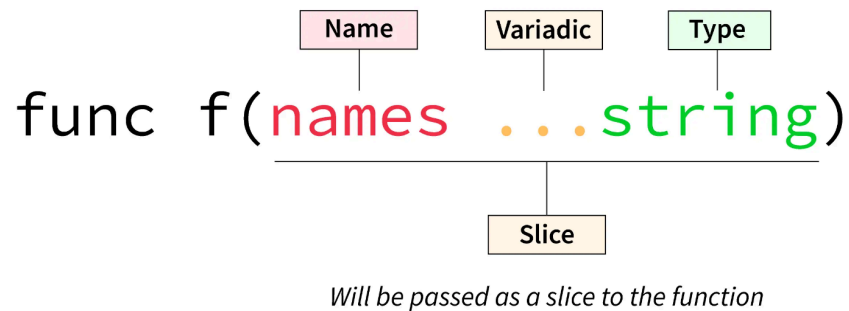
palabras = len(strings.Fields(texto))

return // Return implícito cuando los valores están nombrados
}

```

## Funciones Variádicas (Varargs)

Has estado utilizando `fmt.Println` para imprimir resultados en la pantalla y probablemente te hayas dado cuenta de que permite cualquier número de parámetros de entrada. ¿Cómo lo hace? Como muchos lenguajes, Go soporta parámetros variádicos. **El parámetro variádico debe ser el último (o único) parámetro de la lista de parámetros de entrada. Se indica con tres puntos (...) antes del tipo.** La variable que se crea dentro de la función es una porción del tipo especificado. Se utiliza como cualquier otra porción. Veamos cómo funcionan escribiendo un programa que suma un número base a un número variable de parámetros y devuelve el resultado como una porción de `int`.



```

Go

package utils

import (
    "fmt"
    "strings"
)

```

```

        "time"
    )

// Función variádica básica
func calcularPromedio(numeros ...float64) float64 {
    if len(numeros) == 0 {
        return 0
    }

    var suma float64

    for _, num := range numeros {
        suma += num
    }

    return suma / float64(len(numeros))
}

// Función variádica con parámetros fijos
func LogActividad(nivel string, usuario string, acciones ...string) {
    timestamp := time.Now().Format("2006-01-02 15:04:05")

    fmt.Printf("[%s] %s - Usuario: %s\n", timestamp, nivel, usuario)

    if len(acciones) == 0 {
        fmt.Println("    (Sin acciones registradas)")
    }
}

```

```

        return
    }

    for i, accion := range acciones {
        fmt.Printf("  %d. %s\n", i+1, accion)
    }
    fmt.Println("---")
}

// Función variádica avanzada para el sistema de biblioteca
func GenerarReporte(titulo string, secciones ...func() string) string {
    var builder strings.Builder

    builder.WriteString(fmt.Sprintf("=== %s ===\n", strings.ToUpper(titulo)))

    builder.WriteString(fmt.Sprintf("Generado: %s\n\n",
time.Now().Format("02/01/2006 15:04:05")))

    for i, seccion := range secciones {
        builder.WriteString(fmt.Sprintf("--- Sección %d ---\n", i+1))
        builder.WriteString(seccion())
        builder.WriteString("\n\n")
    }

    return builder.String()
}

// Ejemplos de uso de funciones variádicas

```

```

func ejemplosVariadicas() {

    // Promedio

    prom1 := calcularPromedio(8.5, 9.0, 7.5, 8.0)
    prom2 := calcularPromedio(5.0)
    promVacio := calcularPromedio()

    fmt.Printf("Promedios: %.2f, %.2f, %.2f\n", prom1, prom2, promVacio)

    // Log de actividades
    LogActividad("INFO", "juan.perez",
        "Inicio de sesión exitoso",
        "Búsqueda de libros de ciencia ficción",
        "Préstamo del libro: Dune")

    LogActividad("ERROR", "sistema")

    // Usar slice existente con operador ...
    acciones := []string{
        "Backup de base de datos",
        "Limpieza de archivos temporales",
        "Actualización de índices",
    }

    LogActividad("SYSTEM", "admin", acciones...) // Desempaquetar slice
}

```

## Funciones como Tipos de Primera Clase

En Go, las funciones son tratadas como tipos de primera clase, lo que significa que se pueden usar y manipular de la misma manera que otros tipos de datos como enteros o cadenas. Esto permite a los programadores pasar funciones como argumentos a otras funciones, devolverlas como valores de otras funciones y asignarlas a variables.

### ¿Qué significa "tipos de primera clase"?

Un lenguaje de programación con funciones de primera clase trata a las funciones como cualquier otro valor en el lenguaje. Esto implica que las funciones pueden:

- Ser asignadas a variables: Se pueden guardar en variables, como cualquier otro dato.
- Ser pasadas como argumentos: Se pueden pasar como parámetros a otras funciones.
- Ser devueltas como valores: Se pueden retornar desde otras funciones.

Vamos a verlo con un ejemplo

```
Go

package utils

import (
    "fmt"
    "sort"
    "time"
)

// Definir tipos de función
type FiltroLibro func(Libro) bool
type ComparadorLibro func(Libro, Libro) bool
type TransformadorLibro func(Libro) Libro
```

```
// Función que acepta otra función como parámetro

func FiltrarLibros(libros []Libro, filtro FiltroLibro) []Libro {

    var resultado []Libro

    for _, libro := range libros {
        if filtro(libro) {
            resultado = append(resultado, libro)
        }
    }

    return resultado
}

// Función que retorna una función

func CrearFiltroAutor(autorBuscado string) FiltroLibro {

    return func(libro Libro) bool {

        return strings.Contains(strings.ToLower(libro.Autor),
strings.ToLower(autorBuscado))

    }
}

// Función de orden superior para transformar libros

func MapearLibros(libros []Libro, transformador TransformadorLibro) []Libro {

    resultado := make([]Libro, len(libros))
```

```

    for i, libro := range libros {
        resultado[i] = transformador(libro)
    }

    return resultado
}

// Función para ordenar con comparador personalizado
func OrdenarLibros(libros []Libro, comparador ComparadorLibro) {
    sort.Slice(libros, func(i, j int) bool {
        return comparador(libros[i], libros[j])
    })
}

// Función que combina múltiples filtros
func CombinarFiltros(filtros ...FiltroLibro) FiltroLibro {
    return func(libro Libro) bool {
        for _, filtro := range filtros {
            if !filtro(libro){
                return false
            }
        }
        return true
    }
}

```



```

// Ejemplos prácticos de uso

func ejemplosFuncionesComoTipos() {
    libros := obtenerLibros() // Función que retorna lista de libros

    // 1. Filtros predefinidos

    filtroDisponibles := func(l Libro) bool { return l.Disponible }
    filtroRecientes := func(l Libro) bool {
        return time.Since(l.FechaPublicacion).Hours() < 24*365*2 // Últimos 2
años
    }
    filtroBaratos := func(l Libro) bool { return l.Precio < 25.0 }

    // 2. Crear filtro dinámico

    filtroTolkien := CrearFiltroAutor("Tolkien")

    // 3. Aplicar filtros individuales

    disponibles := FiltrarLibros(libros, filtroDisponibles)
    librosTolkien := FiltrarLibros(libros, filtroTolkien)

    // 4. Combinar múltiples filtros

    filtroComplejo := CombinarFiltros(filtroDisponibles, filtroRecientes,
filtroBaratos)

    librosEspeciales := FiltrarLibros(libros, filtroComplejo)

    // 5. Transformar libros (aplicar descuento)

```

```

aplicarDescuento := func(l Libro) Libro {
    l.Precio = l.Precio * 0.9 // 10% de descuento
    return l
}

librosConDescuento := MapearLibros(libros, aplicarDescuento)

// 6. Ordenar por diferentes criterios
OrdenarLibros(libros, func(a, b Libro) bool {
    return a.FechaPublicacion.After(b.FechaPublicacion) // Más recientes primero
})

fmt.Printf("Encontrados: %d disponibles, %d de Tolkien, %d especiales\n",
    len(disponibles), len(librosTolkien), len(librosEspeciales))
}


```

## Closures en Go

Los closures son una potente característica de la programación Go que permite encapsular el estado y el comportamiento dentro de las funciones. Proporcionan una forma de crear unidades de código autocontenidas que pueden acceder y manipular variables desde su ámbito circundante.

En esencia, un Closure es una función unida a su entorno de referencia. Captura variables de su ámbito externo, permitiendo que esas variables sean accedidas y utilizadas dentro del cierre. Esto permite al cierre mantener su propio estado y conservar el acceso a las variables capturadas incluso después de que la función externa haya finalizado su ejecución.

Los closures en Go pueden ser creados usando tanto funciones anónimas como funciones con nombre. Las funciones anónimas se utilizan a menudo para crear cierres debido a su naturaleza



en línea. Sin embargo, las funciones con nombre también se pueden utilizar cuando una función específica necesita ser reutilizada o cuando se desea una claridad adicional.

Las funciones anónimas se definen directamente en el lugar de su uso, sin un nombre específico. Suelen asignarse a variables y pueden invocarse inmediatamente o pasarse como argumentos a otras funciones.

Las funciones con nombre, en cambio, pueden definirse por separado y asignarse a variables o devolverse desde otras funciones. También pueden capturar variables de su ámbito externo, lo que las hace adecuadas para crear cierres.

## Beneficios y Casos de Uso

Los closures proporcionan varios beneficios en la programación en Go:

- Encapsulan el estado privado: Los cierres permiten la creación de funciones con variables privadas que son inaccesibles desde fuera del cierre. Esto ayuda a construir código modular y seguro.
- Fábricas de funciones: Los cierres pueden actuar como fábricas para generar funciones especializadas basadas en configuraciones o parámetros específicos. Permiten crear funciones personalizadas con comportamientos preestablecidos.
- Mantener el estado a través de múltiples llamadas: Los cierres permiten que las funciones conserven el estado a través de invocaciones sucesivas. Las variables capturadas dentro de los cierres almacenan sus valores, lo que permite a las funciones recordar y actualizar su estado según sea necesario.
- Devoluciones de llamada y controladores de eventos: Los cierres se utilizan habitualmente para implementar retrollamadas y controladores de eventos. Capturan variables y proporcionan un mecanismo para ejecutar acciones específicas cuando se producen eventos.
- Operaciones asíncronas: Los cierres son útiles cuando se trata de operaciones asíncronas o goroutines. Ayudan a pasar datos y comportamientos a las goroutines, asegurando que cada goroutine opera con su propio conjunto de variables capturadas.

Para ilustrar la potencia de los closures, consideremos un ejemplo de implementación de un contador:

Go

```
func createCounter() func() int {  
    count := 0  
    increment := func() int {  
        count++  
        return count  
    }  
    return increment  
}  
  
func main() {  
    counter1 := createCounter()  
    counter2 := createCounter()  
  
    fmt.Println(counter1()) // Output: 1  
    fmt.Println(counter1()) // Output: 2  
    fmt.Println(counter2()) // Output: 1  
    fmt.Println(counter2()) // Output: 2  
}
```