

LUCAS EGA ACADEMY

GO

CLOUD-NATIVE PROGRAMMING & SYSTEMS



Curso Completo de Go

© 2026 Esteban Gutiérrez

Colibrí Editorial EGA

Hecho en México

ÍNDICE

PRÓLOGO	6
Por qué este libro	6
Cómo está organizado	6
Cómo estudiar este libro	7
Una nota sobre el estilo de Go	8
Agradecimiento	8
UNIDAD 1: Introducción a Go y tu primer programa	9
Explicación detallada	10
Ejemplos paso a paso	24
Práctica	34
Resumen de la unidad	41
UNIDAD 2: Variables, constantes y tipos de datos	42
Explicación detallada	43
Ejemplos paso a paso	59
Práctica	72
Resumen de la unidad	79
UNIDAD 3: Operadores y control de flujo	81
Explicación detallada	82
Ejemplos paso a paso	103
Práctica	116
Resumen de la unidad	124
UNIDAD 4: Funciones y paquetes	126
Explicación detallada	127
Ejemplos paso a paso	149
Práctica	161
Resumen de la unidad	170
UNIDAD 5: Arreglos, slices y mapas	171
Explicación detallada	172
Ejemplos paso a paso	188
Práctica	200

Resumen de la unidad	211
UNIDAD 6: Structs y métodos	214
Explicación detallada	215
Ejemplos paso a paso	225
Práctica	241
Resumen de la unidad	254
UNIDAD 7: Interfaces y composición	260
Explicación detallada	264
Ejemplos paso a paso	275
Práctica	290
Resumen de la unidad	307
UNIDAD 8: Goroutines y canales: concurrencia	309
Explicación detallada	312
Ejemplos paso a paso	326
Práctica	339
Resumen de la unidad	355
UNIDAD 9: Manejo de errores y testing	356
Explicación detallada	356
Ejemplos paso a paso	382
Práctica	393
Resumen de la unidad	406
UNIDAD 10: Go en la práctica: servidores web y despliegue	409
Explicación detallada	409
Ejemplos paso a paso	430
Práctica	444
Resumen de la unidad	457
Proyecto final integrador: API de biblioteca	458
Consejos para conseguir trabajo con Go	459
EXAMEN FINAL	461
Parte 1: Opción múltiple	461
Parte 2: Verdadero o falso	464
Parte 3: Preguntas abiertas con código	465

Respuestas y soluciones	468
Cómo calificarte	482

PRÓLOGO

Bienvenido a "Curso Completo de Go", el libro oficial de Lucas EGA Academy (lucasegaacademy.com) para dominar el lenguaje de programación Go, también conocido como Golang.

Por qué este libro

Go nació en Google en 2007, creado por Robert Griesemer, Rob Pike y Ken Thompson, y se liberó al público en 2009. Nació con una misión muy clara: ser un lenguaje simple, rápido de compilar y diseñado desde cero para la era de los servidores multicore y la nube. Hoy, casi dos décadas después, Go mueve una parte enorme de la infraestructura de internet: Docker, Kubernetes, Terraform, Prometheus y miles de APIs y microservicios están escritos en Go. Las empresas lo buscan porque con Go se escribe software rápido, confiable y fácil de mantener, y porque un programador de Go puede ser productivo en semanas, no en meses.

Este libro no es un resumen ni una chuleta de sintaxis. Es un curso completo, con explicaciones extensas, analogías, ejemplos paso a paso, prácticas con soluciones y un examen final. Está diseñado para llevarte desde cero, sin asumir que sabes programar en otro lenguaje (aunque si ya programas, avanzarás más rápido), hasta construir servidores web reales y desplegarlos.

Cómo está organizado

El libro tiene 10 unidades, cada una agrupa 3 clases del curso en video de la academia:

- Unidad 1: Introducción a Go y tu primer programa.
- Unidad 2: Variables, constantes y tipos de datos.
- Unidad 3: Operadores y control de flujo.
- Unidad 4: Funciones y paquetes.
- Unidad 5: Arreglos, slices y mapas.
- Unidad 6: Structs y métodos.
- Unidad 7: Interfaces y composición.
- Unidad 8: Goroutines y canales: concurrencia.
- Unidad 9: Manejo de errores y testing.

- Unidad 10: Go en la práctica: servidores web y despliegue.

Cada unidad sigue la misma estructura para que siempre sepas qué esperar:

1. Introducción: qué vas a aprender y por qué importa.
2. Explicación detallada: la teoría con analogías, errores comunes que cometen los principiantes y buenas prácticas de programadores profesionales.
3. Ejemplos paso a paso: código real, explicado línea por línea, que puedes copiar, ejecutar y modificar.
4. Práctica: ejercicios para resolver por tu cuenta, con soluciones explicadas al final de la unidad. Hazlos de verdad: programar se aprende programando, no leyendo.
5. Resumen de la unidad: lo esencial en pocas líneas para repasar.

Al final del libro encontrarás el examen final: 50 preguntas de opción múltiple, verdadero o falso y preguntas abiertas con código, con todas las respuestas explicadas.

Cómo estudiar este libro

Te recomiendo este método, que es el que usamos en la academia:

1. Lee la explicación detallada de la unidad completa antes de tocar el teclado. Entiende primero el "por qué".
2. Escribe tú mismo cada ejemplo paso a paso en tu computadora. No copies y pegues: escribir el código con tus manos es lo que hace que se quede en tu cabeza.
3. Rompe los ejemplos a propósito. Cambia valores, quita líneas, provoca errores y lee los mensajes del compilador. Go tiene uno de los mejores mensajes de error entre los lenguajes compilados: aprende a leerlos, son tus maestros gratis.
4. Resuelve los ejercicios de la práctica sin mirar las soluciones. Si te atorras más de 30 minutos, mira la solución, entiéndela, ciérrala y vuelve a escribirla tú solo.
5. Al terminar cada unidad, lee el resumen en voz alta. Si hay algo que no puedes explicar con tus palabras, repasa esa parte.

Necesitas una computadora con Go instalado (la Unidad 1 te guía paso a paso) y ganas de practicar. No necesitas pagar nada: Go es gratis y de código abierto, y todas sus herramientas también.

Una nota sobre el estilo de Go

Go tiene una filosofía que vas a escuchar muchas veces en este libro: la simplicidad es una característica, no una limitación. Go deliberadamente tiene pocas palabras clave (solo 25) y evita la "magia". Al principio eso puede sentirse limitado si vienes de otros lenguajes, pero con el tiempo descubrirás que esa simplicidad es lo que hace que el código Go sea tan legible y que los equipos puedan mantener proyectos enormes sin perderse.

Otra idea central: en Go los errores se manejan de forma explícita, la concurrencia es barata gracias a las goroutines y la herramienta `gofmt` se encarga del formato por ti. Abraza estas ideas desde el día uno y programarás como un gopher profesional mucho antes de lo que crees.

Agradecimiento

Gracias por confiar en Lucas EGA Academy para tu formación. Este libro se escribió con un objetivo: que al terminarlo puedas decir con confianza "sé programar en Go" y demostrarlo con proyectos reales. Si este libro te sirve, compártelo y recomiédanos en lucasegaacademy.com. Y recuerda: el mejor programador no es el que más memoriza, sino el que más practica.

¡A programar!

Esteban Lucas EGA Academy - lucasegaacademy.com

UNIDAD 1: Introducción a Go y tu primer programa

Bienvenido a la primera unidad de este curso de Go en Lucas EGA Academy. Si nunca has escuchado hablar de Go, esta es la mejor noticia que te podían dar hoy: estás a punto de aprender un lenguaje de programación que hoy mueve una parte enorme del internet, desde la infraestructura que mantiene vivos millones de servidores hasta herramientas de inteligencia artificial, pasando por las plataformas que usan empresas gigantes todos los días. Y lo mejor de todo es que Go fue diseñado precisamente para personas que están cansadas de los lenguajes complicados, con sintaxis enredada y configuraciones eternas. Go es directo, rápido y honesto: lo que escribes es lo que entiendes, y lo que entiendes es lo que funciona.

En esta unidad vamos a responder las preguntas más importantes antes de escribir código de verdad: qué es Go y por qué existe, quién lo creó y con qué problema quería acabar, dónde se usa en el mundo real para que veas que no es un lenguaje de juguete, cómo instalarlo en tu computadora sin importar si usas Windows, Mac o Linux, y cómo funciona el ecosistema de herramientas que trae consigo. Después vamos a desmenuzar el programa más famoso del mundo de la programación, el clásico "Hola, mundo", para que entiendas pieza por pieza qué hace cada línea, qué significa `package main`, qué significa `func main`, y cómo Go convierte tu texto en un programa ejecutable de verdad. Al final de la unidad vas a tener Go instalado, vas a haber escrito y ejecutado tu primer programa, y vas a entender los comandos básicos del toolchain de Go que usarás durante todo el resto del curso: `go run`, `go build`, `go mod init`, `go mod tidy`, y la herramienta mágica llamada `gofmt` que mantendrá tu código limpio sin que tengas que pelear con espacios y tabuladores.

Por qué importa tanto empezar bien con estos fundamentos. Muchas personas que aprenden programación cometen el mismo error: saltan directo al código sin entender el entorno donde ese código vive. Es como querer cocinar sin conocer tu cocina: puedes seguir una receta, pero en cuanto algo salga mal no vas a saber si el problema fue la estufa, la olla o el ingrediente. En esta unidad nos tomamos el tiempo de conocer la cocina. Vas a entender qué es un compilador, por qué Go compila tan rápido que casi parece un lenguaje interpretado, qué significa que un lenguaje sea de tipado estático pero se sienta liviano, y qué es un módulo, que en Go es la forma moderna de organizar proyectos. Estas ideas van a regresar una y otra vez en las siguientes unidades, así que cada minuto que inviertas aquí te ahorra horas de confusión después.

Go también tiene una filosofía que vale la pena entender desde el primer día. La mayoría de los lenguajes de programación nacen con la idea de "dar más poder": más formas de hacer lo

mismo, más atajos, más sintaxis inteligente. Go nació con la idea contraria: "quitar lo que sobra". Sus creadores decidieron que un lenguaje con menos formas de hacer las cosas produce programas más fáciles de leer, y los programas más fáciles de leer son más fáciles de mantener cuando trabajan cientos de programadores en el mismo código. Esa decisión suena simple, pero es la razón por la que empresas enormes lo adoptaron: cuando tienes un equipo grande, lo más caro no es escribir el código la primera vez, es entenderlo seis meses después. Go está diseñado para que entenderlo seis meses después sea lo más fácil posible. Mientras avanzas en este curso, cada vez que te preguntes "por qué Go hace esto de esta forma tan estricta", la respuesta casi siempre va a ser la misma: porque así el código queda más claro para la próxima persona que lo lea, y esa próxima persona bien puedes ser tú.

Explicación detallada

Qué es Go: el lenguaje que nació de la frustración

Go, también llamado Golang por el dominio de su sitio web (golang.org), es un lenguaje de programación de código abierto creado dentro de Google. La historia empieza en 2007, cuando tres ingenieros de Google, Robert Griesemer, Rob Pike y Ken Thompson, se hartaron de los lenguajes que usaban todos los días. Google tenía un problema de escala que casi nadie más tenía en ese entonces: millones de líneas de código en C++, compilaciones que tardaban horas, y equipos enormes de programadores que tenían que coordinarse entre ellos. Ken Thompson es una leyenda viva de la computación: fue uno de los creadores del sistema operativo Unix y del lenguaje C. Rob Pike también trabajó en Unix y en el sistema operativo Plan 9. Robert Griesemer había trabajado en el motor de JavaScript de Google. No eran aficionados: eran de las personas que mejor entendían los lenguajes de programación en el mundo, y aun así estaban frustrados.

La frustración era muy concreta. C++ era potentísimo, pero compilar un proyecto grande podía tomar una eternidad, y el lenguaje era tan complejo que dos programadores expertos podían escribir código completamente distinto para resolver el mismo problema. Java resolvía parte del desorden, pero traía su propia burocracia: jerarquías de clases enormes, archivos de configuración interminables y una máquina virtual pesada. Python era rápido de escribir, pero lento de ejecutar y sin la seguridad de tipos que Google necesitaba para sistemas críticos. Los tres se preguntaron: ¿y si diseñamos un lenguaje que tenga la velocidad de ejecución de C, la facilidad de escritura de Python, y que compile tan rápido que ni lo notes? Esa pregunta se convirtió en Go, anunciado públicamente en 2009 y con su versión 1.0 en 2012.

Una analogía que ayuda mucho: piensa en los lenguajes de programación como herramientas de carpintería. C es como una sierra manual: potentísima en manos expertas, pero te puedes cortar un dedo si te descuidas. Java es como un taller industrial completo: puedes construir lo que sea, pero necesitas un manual de 500 páginas para prender la máquina. Python es como un taladro inalámbrico: lo agarras y empiezas a trabajar en segundos, pero se queda corto cuando el trabajo es muy pesado. Go es como un taladro profesional bien diseñado: lo agarras y funciona de inmediato, tiene la potencia para el trabajo pesado, y no trae 40 accesorios que nunca vas a usar. Esa es la filosofía: pocas piezas, bien hechas, que encajan perfecto.

Por qué existe Go: los tres problemas que quería resolver

Go no nació porque a alguien se le antojó inventar un lenguaje más. Nació para resolver tres problemas reales que Google sufría todos los días. Entender estos tres problemas te ayuda a entender por qué Go es como es.

El primer problema era la lentitud de compilación. En Google, compilar un programa grande en C++ podía tardar 20, 30 o hasta 60 minutos. Imagina trabajar así: escribes un cambio pequeño, esperas media hora para probarlo, encuentras un error, lo corriges, y esperas otra media hora. Los programadores perdían horas del día simplemente esperando. Go se diseñó para compilar rapidísimo. ¿Cómo? Con decisiones de diseño inteligentes: el compilador de Go no necesita leer archivos de encabezados gigantes como en C, las dependencias se declaran de forma explícita y simple, y el lenguaje evita las construcciones que hacen lento el análisis. El resultado es que programas que en otros lenguajes tardan minutos en compilar, en Go tardan segundos. Esta velocidad cambia la forma de trabajar: puedes probar ideas rápido, equivocarte rápido y corregir rápido, que es exactamente como se aprende a programar.

El segundo problema era la complejidad innecesaria. Los lenguajes populares de la época ofrecían muchísimas formas de hacer lo mismo: herencia múltiple, sobrecarga de operadores, genéricos complicados, excepciones con jerarquías enteras. Cada característica nueva suena bien en un folleto, pero en la práctica cada una es una forma nueva de confundirse. Go tomó la decisión radical de incluir solo lo esencial. No hay herencia de clases como en Java o C++. No hay excepciones como en Python. No hay sobrecarga de operadores. Lo que hay es un conjunto pequeño de herramientas que combinadas resuelven casi todo. La apuesta fue: un lenguaje simple produce programadores productivos más rápido, y el código simple es el código que sobrevive años sin convertirse en una pesadilla de mantenimiento.

El tercer problema era la concurrencia, es decir, hacer muchas cosas al mismo tiempo. Para 2007 ya era evidente que los procesadores ya no se hacían más rápidos aumentando la velocidad de un solo núcleo, sino agregando más núcleos. Los servidores de Google tenían que atender miles de peticiones simultáneas. Hacer eso en C++ o Java era posible, pero

manejar hilos del sistema operativo es complicado y propenso a errores difíciles de encontrar: condiciones de carrera, bloqueos mutuos, fugas de memoria. Go incluyó la concurrencia en el corazón del lenguaje con dos ideas geniales: las goroutines, que son como hilos pero ultraligeros (puedes tener miles sin que tu computadora sude), y los channels, que son tuberías seguras para que esas goroutines se comuniquen. Con unas pocas líneas de Go puedes hacer lo que en otros lenguajes requiere bibliotecas enteras y mucho cuidado. Este tema lo veremos a fondo en una unidad avanzada, pero desde ahora quédate con la idea: la concurrencia en Go es de primera clase, no un parche.

Dónde se usa Go en el mundo real

Una pregunta válida antes de invertir tiempo en un lenguaje es: ¿esto se usa de verdad o es un juguete académico? Con Go la respuesta es contundente: se usa de verdad, y en los lugares más exigentes de la industria.

Docker, la plataforma de contenedores que revolucionó cómo se despliegan aplicaciones en todo el mundo, está escrita en Go. Kubernetes, el sistema que orquesta esos contenedores y que hoy es el estándar para correr aplicaciones en la nube, también está escrito en Go. Terraform, la herramienta más popular para crear infraestructura como código (es decir, definir servidores y redes con archivos de texto en vez de configurarlos a mano), está escrita en Go. Estos tres proyectos son la columna vertebral de la computación en la nube moderna, y los tres eligieron Go por la misma razón: necesitaban programas rápidos, que compilaran rápido, que manejaran miles de conexiones simultáneas y que produjeran un solo archivo ejecutable fácil de distribuir.

Pero no se queda ahí. La base de datos de series de tiempo Prometheus, el sistema de mensajería NATS, la base de datos distribuida CockroachDB, el proxy inverso Traefik y el sistema de control de versiones de GitLab en varias de sus partes usan Go. Empresas como Google, Uber, Dropbox, Twitch, Cloudflare y Mercado Libre usan Go en producción para servicios críticos. Cuando Twitch transmite video a millones de personas al mismo tiempo, hay Go en el camino. Cuando Cloudflare filtra ataques contra sitios web en milisegundos, hay Go en el camino.

¿Y en qué tipo de programas brilla Go? Principalmente en cuatro áreas. Primero, servidores web y APIs: Go trae un servidor HTTP en su biblioteca estándar que es rápido y simple de usar, sin necesidad de instalar frameworks pesados. Segundo, herramientas de línea de comandos: como Go compila a un solo archivo ejecutable sin dependencias externas, es perfecto para crear herramientas que instalas y funcionan de inmediato. Tercero, sistemas distribuidos y microservicios: la concurrencia nativa lo hace ideal para programas que hablan con muchos otros programas al mismo tiempo. Cuarto, DevOps e infraestructura: por eso

Docker, Kubernetes y Terraform lo eligieron. Si te interesa cualquiera de estas áreas, Go es una de las mejores inversiones que puedes hacer con tu tiempo.

Instalación de Go en Windows, Mac y Linux

Instalar Go es sorprendentemente sencillo, y esa sencillez es intencional: el equipo de Go quiere que pases de "cero" a "programa funcionando" en minutos. Vamos paso a paso por cada sistema operativo. En todos los casos, lo primero es descargar el instalador oficial desde el sitio web de Go, que es go.dev. Siempre descarga desde el sitio oficial; es gratis y es la única fuente confiable.

En Windows, descarga el instalador con extensión `.msi` desde go.dev. Haz doble clic en el archivo descargado y sigue el asistente: acepta la licencia, deja la ruta de instalación por defecto (normalmente `C:\Program Files\Go`) y termina la instalación. El instalador configura automáticamente la variable de entorno `PATH`, que es lo que permite que el comando `go` funcione desde cualquier terminal. Para verificar, abre una terminal nueva (puedes buscar "cmd" o "PowerShell" en el menú inicio) y escribe `go version`. Si ves algo como `"go version go1.21.0 windows/amd64"`, la instalación funcionó. Un detalle importante en Windows: después de instalar, cierra y vuelve a abrir la terminal, porque las terminales que ya estaban abiertas no se enteran del cambio en el `PATH`.

En Mac, tienes dos caminos. El más directo es descargar el instalador `.pkg` desde go.dev, abrirlo y seguir los pasos; instala Go en `/usr/local/go` y también actualiza el `PATH`. El otro camino, muy popular entre programadores de Mac, es usar Homebrew, el gestor de paquetes: abres la terminal y escribes `brew install go`. Homebrew descarga y configura todo por ti. Para verificar, abre una terminal y escribe `go version`; deberías ver algo como `"go version go1.21.0 darwin/amd64"` o `"darwin/arm64"` si tu Mac tiene chip Apple Silicon. Si el comando no se reconoce, revisa que `/usr/local/go/bin` esté en tu `PATH`.

En Linux, descarga el archivo `.tar.gz` desde go.dev. Abre una terminal y extrae el archivo en `/usr/local` con un comando como `sudo tar -C /usr/local -xzf go1.21.0.linux-amd64.tar.gz` (ajusta el nombre del archivo a la versión que descargaste). Después agrega `/usr/local/go/bin` a tu `PATH` editando tu archivo de configuración del shell (`~/.bashrc`, `~/.zshrc` o el que uses) con la línea `export PATH=$PATH:/usr/local/go/bin`, y recarga con `source ~/.bashrc`. Verifica con `go version`. Muchas distribuciones de Linux también ofrecen Go en sus repositorios (sudo `apt install golang` en Ubuntu, por ejemplo), pero ojo: esas versiones suelen estar desactualizadas. Para este curso, que usa características de Go 1.21 en adelante, es mejor instalar desde el sitio oficial.

Un error común durante la instalación es descargar el instalador pero no verificar con `go version`, y luego descubrir a mitad del curso que el comando `go` no existe. Haz la verificación ahora mismo: abre tu terminal y escribe `go version`. Si responde con la versión, estás listo. Si dice "comando no encontrado", el problema casi siempre es el PATH: la terminal no sabe dónde está Go. Revisa los pasos de tu sistema operativo y asegúrate de haber abierto una terminal nueva después de instalar.

GOPATH vs módulos: cómo organiza Go tu código

Aquí hay un tema que confunde a muchos principiantes, especialmente si siguen tutoriales viejos de internet, así que vamos a dejarlo clarísimo desde el inicio. Go ha tenido dos formas de organizar el código a lo largo de su historia: el modo GOPATH, que es el antiguo, y los módulos, que son la forma moderna y la que usaremos en todo este curso.

El GOPATH era una carpeta especial en tu computadora (normalmente `~/go` en Mac y Linux, o `%USERPROFILE%\go` en Windows) donde TODO tu código Go tenía que vivir, en una estructura rígida de carpetas: `src` para el código, `pkg` para paquetes compilados y `bin` para ejecutables. Tu proyecto tenía que estar en `$GOPATH/src/github.com/tu-usuario/tu-proyecto`, sin excepciones. Esto funcionaba, pero era incómodo: no podías poner tu proyecto donde quisieras, y manejar dependencias de otros proyectos era un dolor de cabeza porque no había un sistema oficial de versiones.

Desde Go 1.11, y de forma definitiva desde Go 1.16, la forma correcta de trabajar es con módulos. Un módulo es simplemente tu proyecto más un archivo llamado `go.mod` que dice: "este proyecto se llama así y usa esta versión de Go". Con módulos puedes crear tu proyecto en cualquier carpeta de tu computadora, sin la estructura rígida del GOPATH. Las dependencias se declaran en el `go.mod` y Go las descarga y versiona automáticamente. Es más simple, más flexible y es el estándar actual.

La regla de oro para ti: olvida el GOPATH como lugar obligatorio para tu código. Usa módulos siempre. Cuando veas un tutorial viejo que te diga que pongas tu código dentro de `$GOPATH/src`, ignóralo: es de la época anterior a los módulos. En este curso, cada proyecto empezará con `go mod init`, que crea el archivo `go.mod`, y a partir de ahí todo fluye. El GOPATH todavía existe en tu sistema y Go lo usa internamente para guardar dependencias descargadas y herramientas instaladas, pero tú ya no tienes que organizar tu código alrededor de él. Piensa en el GOPATH como la bodega de Go y en los módulos como tu casa: tú vives en tu casa, la bodega solo guarda cosas.

El comando go: tu navaja suiza

El comando go no es solo el compilador: es todo un conjunto de herramientas en uno solo. En otros lenguajes necesitas instalar por separado el compilador, el gestor de dependencias, el formateador de código y el corredor de pruebas. En Go, todo viene incluido. Estos son los subcomandos que usarás constantemente.

`go version` te dice qué versión de Go tienes instalada. Es el primer comando que debes probar después de instalar, y es útil cuando algo no funciona: muchos problemas se resuelven actualizando Go.

`go run` es el comando para ejecutar un programa directamente desde el código fuente, sin generar un archivo ejecutable permanente. Escribes `go run main.go` y Go compila tu programa en memoria y lo ejecuta en un solo paso. Es perfecto mientras estás aprendiendo y probando: escribes, ejecutas, ves el resultado, corriges. Más adelante veremos que también puedes pasar varios archivos: `go run .` ejecuta todos los archivos `.go` de la carpeta actual.

`go build` compila tu código y genera un archivo ejecutable real, un programa que puedes correr sin tener Go instalado. En Windows genera un `.exe`; en Mac y Linux genera un archivo sin extensión que ejecutas con `./nombre`. La magia de `go build` es que el ejecutable incluye todo lo necesario: no necesitas instalar nada más en la computadora donde lo vayas a correr. Por eso las herramientas escritas en Go se distribuyen tan fácil: es un solo archivo.

`go mod init nombre-del-modulo` crea el archivo `go.mod` e inicializa tu proyecto como un módulo. El nombre del módulo suele ser una ruta, como `github.com/tu-usuario/mi-proyecto`, aunque para practicar puedes usar algo simple como `ejemplo.com/hola`. Este comando lo ejecutarás una sola vez por proyecto, al inicio.

`go mod tidy` es el comando que mantiene ordenadas tus dependencias. Revisa tu código, detecta qué paquetes externos usas, los agrega al `go.mod` si faltan y quita los que ya no usas. Es como el asistente de limpieza de tu proyecto: después de agregar o quitar código que usa bibliotecas externas, corre `go mod tidy` y todo queda consistente.

Hay otros comandos útiles que conocerás pronto: `go fmt` formatea tu código automáticamente, `go vet` revisa errores sospechosos, y `go test` corre tus pruebas. Pero con los cinco anteriores (`version`, `run`, `build`, `mod init`, `mod tidy`) tienes todo lo necesario para empezar a trabajar como un programador de Go de verdad.

Estructura de un programa en Go: package main y func main

Todo programa ejecutable en Go tiene la misma estructura mínima, y entenderla te quita el miedo a la "magia" del código. Vamos a ver el programa más famoso del mundo y a desarmarlo pieza por pieza.

```
package main

import "fmt"

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

Guarda ese código en un archivo llamado main.go. Para ejecutarlo, abre la terminal en la carpeta del archivo y escribe go run main.go. Verás en pantalla: Hola, mundo. Felicidades: acabas de ejecutar tu primer programa en Go. Ahora desarmemos cada línea.

La primera línea, package main, declara a qué paquete pertenece este archivo. En Go, todo el código vive dentro de paquetes, que son como cajas que agrupan archivos relacionados. El paquete main es especial: le dice a Go "este paquete se compila como un programa ejecutable, no como una biblioteca". Todo programa que quieras ejecutar directamente necesita un paquete main. Es una regla sin excepciones: sin package main, no hay ejecutable.

La línea import "fmt" trae el paquete fmt a tu programa. fmt es parte de la biblioteca estándar de Go y su nombre viene de "format": sirve para dar formato a texto y mostrarlo en pantalla, entre otras cosas. La palabra import es como decirle a Go: "voy a usar herramientas de esta caja, tráemela". Go es estricto con los imports: si importas un paquete y no lo usas, el programa no compila. Al principio esto molesta, pero es una de esas decisiones de diseño que mantienen el código limpio: nada de imports olvidados acumulando polvo.

Luego viene func main() { ... }. La palabra func declara una función, que es un bloque de código con nombre que hace algo. La función main es tan especial como el paquete main: es el punto de entrada del programa, es decir, la primera función que Go ejecuta cuando corres tu programa. Dentro del paquete main tiene que haber exactamente una función main, ni cero ni dos. Todo lo que pongas dentro de sus llaves se ejecuta de arriba hacia abajo cuando el programa arranca.

Dentro de main está la línea fmt.Println("Hola, mundo"). Aquí pasan tres cosas: fmt.Println es la función "imprimir línea" del paquete fmt (la notación paquete.Funcion es como decir "la función Println que vive en la caja fmt"); los paréntesis contienen los argumentos, lo que le pasas a la

función; y "Hola, mundo" entre comillas dobles es un string, una cadena de texto. `Println` muestra el texto en pantalla y agrega un salto de línea al final. Las llaves `{ }` delimitan el cuerpo de la función: todo lo que está entre ellas pertenece a `main`.

Un error común de principiante es olvidar alguna de estas piezas y no entender el mensaje de error. Si olvidas `package main`, Go te dirá algo sobre que no encuentra el paquete. Si escribes `Func Main` con mayúsculas, no funcionará: Go distingue mayúsculas de minúsculas y `main` tiene que ir en minúsculas. Si olvidas importar `fmt` pero usas `fmt.Println`, el error dirá "undefined: `fmt`". Lee los mensajes de error con calma: en Go suelen ser cortos y directos, y casi siempre apuntan exactamente al problema.

Comentarios y `gofmt`: el código que se explica solo

Un programa no solo tiene que funcionar: tiene que entenderse. Por eso existen los comentarios, que son notas que escribes para humanos y que Go ignora por completo al compilar. En Go hay dos formas de escribir comentarios.

El comentario de una línea empieza con `//` y todo lo que sigue en esa línea es ignorado por el compilador. Se usa para explicar una línea o un bloque corto de código.

```
package main

import "fmt"

func main() {
    // Esta línea muestra un saludo en la pantalla
    fmt.Println("Hola, mundo") // los comentarios también pueden ir al final de una
    línea
}
```

El comentario de bloque empieza con `/*` y termina con `*/`, y puede abarcar varias líneas. Se usa para explicaciones largas o para desactivar temporalmente un pedazo de código mientras pruebas algo.

```

/*
Este programa es mi primer programa en Go.
Lo escribí siguiendo la unidad 1 del curso
de Lucas EGA Academy.
*/
package main

import "fmt"

func main() {
    fmt.Println("Hola, mundo")
}

```

Buenas prácticas con comentarios: explica el POR QUÉ, no el QUÉ. Un comentario que dice // imprime hola mundo encima de una línea que claramente imprime hola mundo no aporta nada; es ruido. En cambio, un comentario que explica por qué elegiste cierto valor o por qué el orden de las operaciones importa, ese sí vale oro. Y mantén los comentarios actualizados: un comentario que describe código que ya cambió es peor que no tener comentario, porque miente.

La otra herramienta de legibilidad es gofmt, el formateador oficial de Go. gofmt toma tu código y lo reescribe con el formato estándar: indentación con tabuladores, espacios consistentes, llaves en su lugar. Lo importante es entender la filosofía: en Go no se discute el estilo. No hay debates de equipo sobre si la llave va en la misma línea o en la siguiente, porque gofmt ya lo decidió por todos. Esto elimina una fuente infinita de discusiones y hace que todo el código Go del mundo se vea igual, lo cual facilita leer código ajeno.

Para usarlo, escribe gofmt -w main.go y tu archivo quedará formateado en su lugar. La bandera -w significa "write": escribe el resultado en el archivo. Sin -w, gofmt solo muestra cómo quedaría, sin cambiar nada. Muchos editores de código ejecutan gofmt automáticamente cada vez que guardas el archivo, y te recomiendo activar esa opción en cuanto puedas: así nunca tendrás que pensar en el formato. Un detalle curioso: gofmt no solo arregla espacios; también puede alinear imports y detectar errores de sintaxis básicos. Si gofmt se queja de tu archivo, casi seguro hay un error de sintaxis que debes corregir primero.

Errores comunes al empezar y cómo evitarlos

Antes de pasar a los ejemplos, vale la pena conocer los tropiezos clásicos de quien empieza con Go, para que cuando te pasen (porque te van a pasar) no entres en pánico.

El primero es el error de "package main" faltante o mal escrito. Recuerda: el archivo que quieres ejecutar debe empezar con package main, en minúsculas, y debe contener func main(). Si Go te dice "function main is undeclared in the main package" o algo parecido, revisa que tu función se llame exactamente main.

El segundo es importar paquetes que no usas, o usar paquetes que no importaste. Go no compila si hay un import sin usar: te dirá "imported and not used". La solución es simple: quita el import o úsalo. Y si usas fmt.Println sin haber escrito import "fmt", el error será "undefined: fmt": agrega el import.

El tercero es la llave de apertura en la línea equivocada. En Go, la llave { que abre una función o un bloque DEBE ir en la misma línea que la declaración, no en la línea siguiente como en otros lenguajes. Esto no es una sugerencia de estilo: es una regla del lenguaje, porque Go inserta puntos y coma automáticamente al final de las líneas. Si escribes func main() y la llave en la línea siguiente, el compilador se confunde. gofmt te ayuda a no cometer este error, pero conviene saber por qué existe la regla.

El cuarto es no guardar el archivo antes de ejecutar go run. Parece obvio, pero le pasa a todo el mundo: editas el código, corres go run main.go, y ves el resultado viejo porque olvidaste guardar. Acostúmbrate a guardar antes de ejecutar, o activa el guardado automático en tu editor.

El quinto es ejecutar go run desde la carpeta equivocada. Si tu archivo está en Documentos/go-proyectos/hola/main.go y tu terminal está en otra carpeta, go run main.go no va a encontrar el archivo. Usa cd para moverte a la carpeta correcta, o pasa la ruta completa. Aprender a moverte por la terminal con cd y a listar archivos es una habilidad que este curso asume poco a poco, pero para empezar basta con saber llegar a tu carpeta de trabajo.

go env y el PATH: conociendo tu instalación

Cuando instalas Go, en realidad instalas mucho más que un compilador: instalas un ecosistema con carpetas, variables de entorno y herramientas auxiliares. El comando go env te muestra toda esa configuración de un vistazo. Ejecútalo en tu terminal y verás una lista larga de variables como GOROOT, GOPATH, GOOS y GOARCH. No necesitas memorizarlas todas, pero hay cuatro que vale la pena conocer.

GOROOT es la carpeta donde vive la instalación de Go: ahí están el compilador, la biblioteca estándar y las herramientas. Normalmente no debes modificar nada dentro de GOROOT; es territorio del instalador. GOPATH, como ya vimos, es tu espacio de trabajo personal: ahí Go guarda las dependencias que descarga y las herramientas que instalas con go install. GOOS indica el sistema operativo para el que se compila (windows, darwin para Mac, linux) y

GOARCH la arquitectura del procesador (amd64 para procesadores de 64 bits tradicionales, arm64 para chips Apple Silicon y procesadores ARM modernos). Estas dos últimas son la base de una de las características más queridas de Go: la compilación cruzada, que te permite generar un ejecutable para Windows estando en Mac, o para Linux estando en Windows, con solo cambiar dos variables. Lo veremos en una unidad avanzada, pero es bueno saber desde ahora que esa capacidad existe y es trivial de usar.

El PATH es una variable de tu sistema operativo que lista las carpetas donde la terminal busca programas cuando escribes un comando. Cuando escribes `go version`, tu terminal recorre el PATH carpeta por carpeta hasta encontrar el programa `go`. Si Go está instalado pero el PATH no incluye su carpeta `bin`, la terminal dirá "comando no encontrado" aunque todo esté perfecto. Por eso el instalador de Go actualiza el PATH automáticamente, y por eso debes abrir una terminal nueva después de instalar: las terminales viejas cargaron el PATH antiguo y no se enteran del cambio.

Si alguna vez necesitas verificar manualmente, puedes ver tu PATH con `echo $PATH` en Mac y Linux, o `echo %PATH%` en Windows. Busca en la lista la ruta de Go: `/usr/local/go/bin` en Mac y Linux, o `C:\Program Files\Go\bin` en Windows. Si no aparece, agrégala siguiendo los pasos de instalación de tu sistema. Este diagnóstico de dos minutos resuelve el 90 por ciento de los problemas de "instalé Go pero no funciona".

Biblioteca estándar: la caja de herramientas que ya tienes

Una de las grandes ventajas de Go es su biblioteca estándar: un conjunto enorme de paquetes oficiales que vienen incluidos con la instalación y cubren casi todo lo que un programa necesita. Cuando en otros lenguajes tienes que buscar e instalar una biblioteca externa para tareas comunes, en Go muchas veces ya viene incluida, probada y documentada.

El paquete `fmt`, que ya usamos, sirve para entrada y salida de texto con formato. El paquete `strings` trae funciones para manipular texto: convertir a mayúsculas, dividir, unir, buscar. El paquete `strconv` convierte entre strings y números, y lo usaremos mucho en esta unidad. El paquete `math` ofrece funciones matemáticas como raíz cuadrada y potencias. El paquete `time` maneja fechas, horas y duraciones. El paquete `os` permite interactuar con el sistema operativo: leer variables de entorno, manejar archivos. El paquete `net/http` incluye un servidor web completo: con unas 20 líneas de Go puedes levantar un servidor que responda peticiones en tu navegador, sin instalar nada más. Y el paquete `encoding/json` convierte datos entre Go y JSON, el formato más usado para que programas se comuniquen por internet.

La filosofía de la biblioteca estándar es la misma que la del lenguaje: simple, bien documentada y estable. Google mantiene un compromiso de compatibilidad muy serio: el

código que escribas hoy con la biblioteca estándar seguirá compilando en versiones futuras de Go. Esa estabilidad es oro para empresas y también para ti como estudiante: lo que aprendas no se vuelve obsoleto en un año.

Para explorar la biblioteca estándar, el mejor recurso es la documentación oficial en pkg.go.dev, donde cada paquete tiene su lista de funciones con ejemplos. Acostúmbrate a consultarla: saber buscar en la documentación es una habilidad más valiosa que memorizar funciones, porque ningún programador memoriza toda la biblioteca estándar.

Cómo piensa Go: tipado estático con alma liviana

Go es un lenguaje de tipado estático, lo que significa que cada variable tiene un tipo definido (número entero, texto, booleano) y el compilador verifica que no mezcles tipos de forma incorrecta. Esto contrasta con lenguajes de tipado dinámico como Python o JavaScript, donde una variable puede guardar un número en una línea y un texto en la siguiente sin que nadie se queje hasta que el programa falla en plena ejecución.

El tipado estático tiene una gran ventaja: muchos errores se detectan al compilar, antes de que el programa corra. Si intentas sumar un texto con un número, Go te lo dice de inmediato y no te deja continuar hasta corregirlo. En un lenguaje dinámico, ese error podría esconderse hasta que un usuario real ejecute justo esa parte del código. Para programas grandes y equipos grandes, atrapar errores temprano vale muchísimo.

Pero el tipado estático tiene fama de ser verboso y burocrático: en Java tienes que declarar el tipo de todo, todo el tiempo, con sintaxis pesada. Go rompe ese estereotipo con la inferencia de tipos: en la mayoría de los casos Go deduce el tipo automáticamente a partir del valor que asignas. Cuando escribes `edad := 30`, Go entiende que `edad` es un entero sin que tengas que decirlo. Obtienes la seguridad del tipado estático con la comodidad de escritura de un lenguaje dinámico. Es lo mejor de los dos mundos, y es una de las razones por las que programar en Go se siente ágil.

Hay una regla de oro que resume la actitud de Go hacia los tipos: sé estricto donde importa y flexible donde no. Go es estricto al no permitir mezclar tipos incompatibles ni dejar variables sin usar, porque esos son errores reales que causan bugs reales. Pero es flexible al inferir tipos y al no exigirte ceremonias innecesarias, porque la burocracia no previene bugs, solo cansa. Mientras avances en el curso, notarás este equilibrio una y otra vez: Go te cuida de los errores tontos sin tratarte como tonto.

El flujo de trabajo del programador de Go: editar, compilar, probar

Ya conoces las herramientas por separado; ahora veamos cómo se combinan en el día a día de un programador de Go. El ciclo básico tiene cuatro pasos y lo repetirás cientos de veces: editar el código en tu editor, guardar el archivo, ejecutar con `go run` para probar, y leer el resultado o el error. Cuando el programa hace lo que quieres, lo formateas con `gofmt` y, si es una versión para compartir, lo compilas con `go build`. Ese es todo el ritual. No hay servidores que reiniciar, no hay configuraciones que regenerar, no hay cachés que limpiar (casi nunca).

Este ciclo es rápido en Go por dos razones que ya conoces: la compilación veloz y el comando `go run` que une compilación y ejecución en un paso. Aprovechalo para trabajar en incrementos pequeños: escribe un pedacito de funcionalidad, pruébalo, y solo cuando funcione pasa al siguiente pedacito. Los principiantes suelen cometer el error de escribir 100 líneas de código nuevo y luego intentar que todo funcione de golpe; cuando algo falla, no saben cuál de las 100 líneas es la culpable. El programador experimentado escribe 10 líneas, las prueba, y repite. Parece más lento, pero es muchísimo más rápido, porque depurar 10 líneas es trivial y depurar 100 es una pesadilla.

Hablemos del editor de código, porque pasarás horas en él. No necesitas nada exótico: Visual Studio Code con la extensión oficial de Go es la combinación más popular y es gratuita. La extensión te da autocompletado, resaltado de sintaxis, formato automático con `gofmt` al guardar y mensajes de error integrados. Otras opciones válidas son GoLand (de paga, muy potente), Vim o Neovim con plugins de Go, y hasta un editor simple si estás empezando. Lo importante no es cuál elijas, sino que configures el formateo automático al guardar: es la forma más barata de mantener tu código limpio sin pensar en ello.

Un consejo sobre los errores de compilación que te ahorrará frustración: léelos de arriba hacia abajo y ataca el primero. Cuando Go encuentra un error, a veces los errores siguientes son consecuencia del primero, como fichas de dominó. Corrige la primera línea marcada, vuelves a compilar, y la mitad de los errores desaparecen solos. Además, los errores de Go suelen incluir el archivo y el número de línea exactos, por ejemplo `main.go:12:34`, donde 12 es la línea y 34 la columna. Ve directo a esa línea: el problema casi siempre está ahí o muy cerca.

Por último, adopta desde hoy el hábito de probar casos extremos con tus programas. Si tu programa calcula días a partir de años, pruébalo con 0, con 1 y con 100, no solo con tu edad. Si lee un nombre, prueba qué pasa con un nombre largo. Este hábito, que en la industria se llama "pensar en casos borde", es lo que separa al programador que "hace que funcione" del programador que "hace que funcione siempre". No necesitas herramientas especiales para esto: solo curiosidad y unos segundos extra por programa.

Dónde seguir aprendiendo: documentación y comunidad Go

Ningún curso, por completo que sea, puede enseñarte todo un lenguaje: la habilidad más importante que puedes llevarte de esta unidad es saber dónde buscar cuando te atores. La buena noticia es que la comunidad de Go tiene fama de ser una de las más amables y de tener la mejor documentación entre los lenguajes populares.

Tu primera parada siempre debe ser la documentación oficial en go.dev. Ahí está el tour interactivo de Go, que te deja probar código en el navegador sin instalar nada; la documentación de cada paquete de la biblioteca estándar con ejemplos ejecutables; y las guías oficiales como "Cómo escribir código Go" y "Effective Go", que explican las convenciones y el estilo idiomático. Cuando tengas una duda sobre una función, busca su página en pkg.go.dev antes de preguntar en ningún lado: nueve de cada diez dudas se resuelven ahí en dos minutos.

Tu segunda parada es el comando `go doc`, que te muestra la documentación sin salir de la terminal. Por ejemplo, `go doc fmt.Println` te muestra qué hace la función `Println`, qué parámetros recibe y qué devuelve. Es rapidísimo y funciona sin internet, porque la documentación viene instalada con Go. Acostúmbrate a usarlo cada vez que encuentres una función desconocida en un ejemplo: en diez segundos sabes qué hace.

Para preguntas y discusión, la comunidad de Go se reúne en varios lugares. El foro oficial en forum.golangbridge.org es amable con principiantes. En Reddit, [r/golang](https://www.reddit.com/r/golang) tiene discusiones de todos los niveles. En Stack Overflow, la etiqueta "go" tiene cientos de miles de preguntas respondidas: antes de publicar una pregunta nueva, busca si alguien ya la hizo, porque casi seguro que sí. Y si prefieres contenido en español, hay comunidades activas de Go en español en Discord, Telegram y YouTube, además de este curso de Lucas EGA Academy, que está diseñado para acompañarte paso a paso en tu idioma.

Un consejo final sobre cómo pedir ayuda que te servirá toda la vida: cuando preguntes, incluye siempre cuatro cosas: qué querías lograr, qué código escribiste (el mínimo que reproduzca el problema), qué esperabas que pasara y qué pasó en realidad (incluyendo el mensaje de error completo). Las preguntas bien hechas reciben respuestas en minutos; las preguntas vagas como "no me funciona, ayuda" suelen ser ignoradas. Aprender a preguntar bien es aprender a pensar bien, y es una habilidad que ningún compilador te puede dar.

Ejemplos paso a paso

Ahora vamos a poner las manos en el código. Cada ejemplo incluye el código completo, listo para copiar en un archivo y ejecutar con go run. Te recomiendo que no solo copies: escribe cada ejemplo tú mismo, ejecútalo, y luego juega con él cambiando textos y valores. Programar se aprende programando, no leyendo.

Ejemplo 1: Tu primer "Hola, mundo" desde cero

Vamos a crear tu primer proyecto Go de la forma moderna, con módulos. Abre tu terminal y ejecuta estos comandos uno por uno. Primero crea una carpeta para tu proyecto y entra en ella.

```
mkdir hola-go  
cd hola-go
```

Ahora inicializa el módulo. Esto crea el archivo go.mod que convierte tu carpeta en un proyecto Go oficial.

```
go mod init ejemplo.com/hola
```

Verifica que se creó el archivo go.mod con el contenido esperado.

```
cat go.mod
```

Deberías ver algo como:

```
module ejemplo.com/hola  
  
go 1.21
```

Ahora crea el archivo main.go con el siguiente contenido. Puedes usar cualquier editor de texto o de código.

```
package main

import "fmt"

func main() {
    fmt.Println("Hola, mundo")
    fmt.Println("Estoy aprendiendo Go en Lucas EGA Academy")
}
```

Ejecútalo con:

```
go run main.go
```

Y verás en pantalla:

```
Hola, mundo
Estoy aprendiendo Go en Lucas EGA Academy
```

Qué acabas de lograr: creaste un módulo, escribiste un programa con dos instrucciones de impresión y lo ejecutaste. El comando `go run` compiló tu código en segundo plano y lo ejecutó sin dejar archivos extra. Si quieres generar el ejecutable permanente, usa `go build`: se creará un archivo llamado `hola` (o `hola.exe` en Windows) que puedes ejecutar directamente con `./hola`. Prueba cambiar los textos entre comillas y vuelve a ejecutar: verás que el cambio se refleja de inmediato. Así de rápido es el ciclo de trabajo en Go.

Ejemplo 2: Imprimir con formato usando `fmt`

El paquete `fmt` tiene varias funciones para mostrar texto, y vale la pena conocer las tres principales desde el inicio: `Println`, `Print` y `Printf`. `Println` imprime sus argumentos y agrega un salto de línea al final, además de separar los argumentos con espacios. `Print` imprime sin agregar el salto de línea. `Printf` es la más potente: imprime usando una plantilla con "verbos" de formato, como `%s` para texto y `%d` para números enteros.

```

package main

import "fmt"

func main() {
    nombre := "Esteban"
    edad := 30

    fmt.Println("Hola, me llamo", nombre, "y tengo", edad, "años")
    fmt.Print("Esta línea no termina con salto...")
    fmt.Print("por eso sigo en el mismo renglón\n")
    fmt.Printf("Me llamo %s y tengo %d años\n", nombre, edad)
}

```

Al ejecutarlo verás:

```

Hola, me llamo Esteban y tengo 30 años
Esta línea no termina con salto...por eso sigo en el mismo renglón
Me llamo Esteban y tengo 30 años

```

Observa los detalles. En la primera línea, `Println` puso espacios automáticamente entre los argumentos. En las líneas con `Print`, tuvimos que agregar `\n` manualmente al final para el salto de línea: `\n` es un carácter especial que significa "nueva línea". En la línea con `Printf`, `%s` se reemplazó con el valor de nombre y `%d` con el valor de edad, en el orden en que aparecen. `Printf` es la función que más usarás cuando quieras controlar exactamente cómo se ve la salida. Los verbos más comunes son `%s` (string), `%d` (entero decimal), `%f` (número con decimales), `%t` (booleano) y `%v` (valor en formato genérico, sirve para casi todo). No te preocupes si la sintaxis `nombre :=` te parece rara: en la unidad 2 la desmenuzamos a fondo.

Ejemplo 3: Comentarios que sí aportan

Este ejemplo muestra un programa con comentarios bien usados: explican el porqué de las decisiones, no repiten lo obvio. Cópialo, ejecútalo y nota que los comentarios no afectan la salida.

```

package main

import "fmt"

func main() {
    // Usamos Printf en vez de Println para controlar el formato exacto
    // del mensaje, porque el curso pide dos decimales en el promedio.
    nombre := "Ana"
    promedio := 9.56

    fmt.Printf("Alumna: %s\n", nombre)
    fmt.Printf("Promedio: %.2f\n", promedio)

    /*
        NOTA PARA EL FUTURO:
        Si el promedio viene de un cálculo, hay que validar que esté
        entre 0 y 10 antes de mostrarlo. Por ahora usamos un valor fijo
        para practicar el formato.
    */
    fmt.Println("Fin del reporte")
}

```

Salida:

```

Alumna: Ana
Promedio: 9.56
Fin del reporte

```

Fíjate en el verbo `%.2f`: significa "número flotante con exactamente 2 decimales". Es muy útil para precios, promedios y calificaciones. Y observa el comentario de bloque: describe una mejora pendiente. Ese tipo de notas son comunes en código real: marcan trabajo futuro sin romper nada.

Ejemplo 4: gofmt en acción

Escribe el siguiente código a propósito desordenado en un archivo llamado `desorden.go`. Sí, a propósito: quiero que veas lo que hace `gofmt`.

```
package main
import "fmt"
func main() {
fmt.Println("línea sin indentación")
    fmt.Println("línea con demasiados espacios")
fmt.Println("otra línea pegada")
}
```

El programa funciona si lo ejecutas con `go run desorden.go`, porque Go no exige un formato para compilar. Pero se ve terrible. Ahora ejecuta:

```
gofmt -w desorden.go
```

Abre el archivo de nuevo y verás que `gofmt` lo reescribió así:

```
package main

import "fmt"

func main() {
    fmt.Println("línea sin indentación")
    fmt.Println("línea con demasiados espacios")
    fmt.Println("otra línea pegada")
}
```

Magia: indentación con tabuladores, línea vacía entre el `import` y la función, todo consistente. Este es el formato oficial de Go y no se discute. Acostúmbrate a correr `gofmt -w` antes de considerar terminado cualquier archivo, o mejor aún, configura tu editor para que lo haga solo al guardar. En equipos de trabajo, el código con formato consistente se revisa más rápido y genera menos conflictos.

Ejemplo 5: go build y tu primer ejecutable

Hasta ahora hemos usado `go run`, que compila y ejecuta sin dejar rastro. Ahora vamos a crear un programa ejecutable de verdad, de esos que puedes compartir con alguien que no tiene Go instalado. Crea un archivo llamado `saludo.go`:

```

package main

import "fmt"

func main() {
    fmt.Println("=====")
    fmt.Println("  Bienvenido a mi primer programa  ")
    fmt.Println("    compilado con Go                ")
    fmt.Println("=====")
    fmt.Println("Este programa es un archivo ejecutable.")
    fmt.Println("No necesita Go instalado para correr.")
}

```

Compíllalo con:

```
go build -o saludo saludo.go
```

La bandera -o le dice a Go cómo se llamará el archivo ejecutable. En Windows el comando genera saludo.exe automáticamente. Ahora ejecuta el programa directamente, sin go run:

En Mac o Linux:

```
./saludo
```

En Windows:

```
saludo.exe
```

Verás el mismo texto de siempre, pero con una diferencia fundamental: ese archivo funciona solo. Puedes copiarlo a otra computadora sin Go y seguirá funcionando. Esa es una de las superpotencias de Go: el ejecutable lleva todo incluido. Para comparar tamaños y darte una idea, ejecuta `ls -lh saludo` (o dir en Windows): verás que el archivo pesa unos pocos megabytes, porque incluye el runtime de Go. A cambio, no necesitas instalar nada más. Cuando en el futuro crees herramientas para otros, esta característica te va a encantar.

Ejemplo 6: Un programa con varios archivos

Los programas reales no viven en un solo archivo. Vamos a dividir un programa en dos archivos para ver cómo Go los une. Crea una carpeta nueva, inicializa un módulo y crea estos dos archivos.

Archivo main.go:

```
package main

import "fmt"

func main() {
    fmt.Println("=== Mi programa de dos archivos ===")
    mostrarSaludo()
    mostrarDespedida()
}
```

Archivo saludos.go:

```
package main

import "fmt"

// mostrarSaludo imprime un saludo de bienvenida.
func mostrarSaludo() {
    fmt.Println("Hola, qué bueno verte por aquí.")
}

// mostrarDespedida imprime un mensaje de cierre.
func mostrarDespedida() {
    fmt.Println("Nos vemos en la siguiente unidad.")
}
```

Nota que ambos archivos declaran package main: pertenecen al mismo paquete. Para ejecutar el programa con varios archivos, usa go run . (con punto), que le dice a Go "compila y ejecuta todos los archivos .go de esta carpeta".

```
go run .
```

Salida:

```
=== Mi programa de dos archivos ===
Hola, qué bueno verte por aquí.
Nos vemos en la siguiente unidad.
```

Si intentas go run main.go solo, fallará con "undefined: mostrarSaludo", porque le pediste compilar únicamente main.go y las funciones viven en el otro archivo. Este es un error clásico:

recuerda que `go run .` incluye todo. A partir de ahora, cuando tus programas crezcan, organízalos en varios archivos por tema: es la forma profesional de trabajar y tu yo del futuro te lo agradecerá.

Ejemplo 7: Leyendo lo que el usuario escribe

Un programa que solo muestra texto es un monólogo. Vamos a hacerlo conversar: leerá el nombre del usuario desde el teclado y lo saludará. Para eso usamos `fmt.Scan`, que lee lo que el usuario escribe.

```
package main

import "fmt"

func main() {
    var nombre string

    fmt.Print("¿Cómo te llamas? ")
    fmt.Scan(&nombre)

    fmt.Printf("Mucho gusto, %s. Bienvenido al curso de Go.\n", nombre)
}
```

Ejecútalo con `go run main.go`. El programa se detendrá en la pregunta y esperará a que escribas tu nombre y pulses Enter. Si escribes "Luis", verás:

```
¿Cómo te llamas? Luis
Mucho gusto, Luis. Bienvenido al curso de Go.
```

Dos cosas nuevas aquí. Primero, `var nombre string` declara una variable llamada `nombre` de tipo `string` (texto). En la unidad 2 veremos variables a fondo; por ahora basta con saber que es una cajita con nombre donde guardamos el texto. Segundo, el símbolo `&` antes de `nombre` en `fmt.Scan(&nombre)`: significa "la dirección de la variable", es decir, le decimos a `Scan` dónde guardar lo que lea. Si olvidas el `&`, el programa compilará pero no guardará bien el dato, o te dará un error. Es un detalle fácil de olvidar y difícil de detectar a simple vista, así que memorízalo: `fmt.Scan` siempre lleva `&` antes de cada variable.

Una limitación honesta: `fmt.Scan` lee hasta el primer espacio. Si el usuario escribe "María Fernanda", solo guardará "María". Para leer líneas completas con espacios se usan otras técnicas que veremos más adelante. Por ahora, para practicar, pide datos de una sola palabra.

Ejemplo 8: Tu primer programa útil: calculadora de edad en días

Vamos a cerrar los ejemplos con un programa que hace algo útil: calcula cuántos días aproximadamente has vivido a partir de tu edad en años. Usa todo lo aprendido: variables, lectura de datos, impresión con formato y comentarios.

```
package main

import "fmt"

func main() {
    var edad int

    fmt.Println("=== Calculadora de días vividos ===")
    fmt.Print("¿Cuántos años tienes? ")
    fmt.Scan(&edad)

    // Aproximación simple: 365 días por año.
    // Ignoramos los años bisiestos para mantener el ejemplo claro.
    dias := edad * 365

    fmt.Printf("Tienes %d años.\n", edad)
    fmt.Printf("Eso es aproximadamente %d días de vida.\n", dias)
    fmt.Println("¡Aprovéchalos aprendiendo Go!")
}
```

Si el usuario escribe 25, la salida será:

```
=== Calculadora de días vividos ===
¿Cuántos años tienes? 25
Tienes 25 años.
Eso es aproximadamente 9125 días de vida.
¡Aprovéchalos aprendiendo Go!
```

Observa la línea `dias := edad * 365`: crea la variable `dias` y le asigna el resultado de la multiplicación en un solo paso. Esta es la declaración corta que veremos en la unidad 2. Y nota el comentario honesto sobre los años bisiestos: en código real, reconocer las simplificaciones con un comentario es una buena práctica. Nadie espera que tu primer programa sea perfecto; se espera que sea claro y honesto sobre lo que hace.

Ejemplo 9: Explorando tu instalación con go env

Este ejemplo no es un programa Go, sino una exploración de tu entorno con la terminal. Ejecuta cada comando y observa la salida: entender tu instalación te ahorrará horas de confusión en el futuro.

```
go version
go env GOROOT
go env GOPATH
go env GOOS GOARCH
```

En mi computadora con Linux, la salida se ve así (la tuya variará según tu sistema):

```
go version go1.21.0 linux/amd64
/usr/local/go
/home/esteban/go
linux
amd64
```

Qué significa cada línea: la primera confirma tu versión de Go. La segunda muestra dónde está instalado Go (GOROOT). La tercera muestra tu espacio de trabajo personal (GOPATH), donde Go guardará dependencias y herramientas. Las últimas dos muestran tu sistema operativo y arquitectura: linux/amd64, o windows/amd64 en Windows, o darwin/arm64 en una Mac moderna. Si alguna vez pides ayuda en un foro o a un compañero, estos cuatro datos son lo primero que te van a preguntar, así que ya sabes dónde encontrarlos.

Como ejercicio extra, ejecuta go env sin argumentos y recorre la lista completa. No necesitas entender cada variable hoy; solo familiarízate con la idea de que Go guarda su configuración en variables de entorno que puedes consultar en cualquier momento.

Ejemplo 10: Un vistazo a la biblioteca estándar con strings

Para cerrar, un ejemplo que usa un paquete de la biblioteca estándar distinto de fmt: el paquete strings, que sirve para manipular texto. Esto te muestra el patrón que repetirás cientos de veces: importar un paquete y usar sus funciones con la notación paquete.Funcion.

```

package main

import (
    "fmt"
    "strings"
)

func main() {
    frase := "estoy aprendiendo go en lucas ega academy"

    fmt.Println("Original: ", frase)
    fmt.Println("Mayúsculas:", strings.ToUpper(frase))
    fmt.Println("¿Contiene 'go'?", strings.Contains(frase, "go"))
    fmt.Println("Palabras: ", strings.Count(frase, " ")+1)
}

```

Salida:

```

Original:  estoy aprendiendo go en lucas ega academy
Mayúsculas: ESTOY APRENDIENDO GO EN LUCAS EGA ACADEMY
¿Contiene 'go'? true
Palabras:  7

```

Dos novedades aquí. Primero, el import en bloque con paréntesis: cuando importas varios paquetes, puedes agruparlos entre paréntesis en vez de escribir una línea import por paquete. Ambas formas son válidas; gofmt prefiere el bloque cuando hay más de un import. Segundo, las funciones de strings: ToUpper convierte a mayúsculas, Contains revisa si un texto contiene otro (devuelve true o false), y Count cuenta cuántas veces aparece algo: contando espacios + 1 obtenemos el número de palabras. Este ejemplo es una probadita de lo que la biblioteca estándar puede hacer sin instalar nada externo. En la unidad 2 usaremos strconv de forma parecida para convertir texto a números.

Práctica

Llegó la hora de ensuciarse las manos. Los ejercicios están ordenados de menor a mayor dificultad. Intenta resolver cada uno sin mirar la solución: escribe el código, ejecútalo, y solo cuando funcione (o cuando de verdad te atores) revisa la solución. Equivocarse es parte del proceso; cada error que entiendes es una lección que no se olvida.

Ejercicio 1: Escribe un programa que imprima tu nombre completo, tu ciudad y tu lenguaje de programación favorito (que a partir de hoy es Go), cada dato en una línea distinta. Usa tres llamadas a `fmt.Println`.

Ejercicio 2: Escribe un programa que muestre el siguiente dibujo usando `fmt.Println`, línea por línea. Cuida que los espacios queden exactamente igual.

```
  /\_/\
 (  o.o  )
  > ^ <
```

Ejercicio 3: Escribe un programa que pida al usuario su nombre y su color favorito, y luego imprima un mensaje como: "Hola Ana, qué buen gusto tienes: el azul es un gran color." Usa `fmt.Scan` para leer los dos datos y `fmt.Printf` para el mensaje final.

Ejercicio 4: Escribe un programa que pida al usuario un número entero de horas y calcule a cuántos minutos y a cuántos segundos equivalen. Muestra los tres valores con `fmt.Printf`. Por ejemplo, si el usuario escribe 2, el programa debe mostrar 2 horas, 120 minutos y 7200 segundos.

Ejercicio 5: Escribe un programa que use comentarios de línea y de bloque para documentar qué hace cada parte. El programa debe pedir el nombre de una mascota y su edad en años, calcular su edad aproximada en "años humanos" multiplicando por 7, y mostrar el resultado con un mensaje amable. Los comentarios deben explicar el porqué de la multiplicación por 7.

Ejercicio 6 (reto): Escribe un programa dividido en dos archivos. El archivo `main.go` debe tener la función `main` que muestre un título y llame a dos funciones: `mostrarMenu` y `mostrarCreditos`. El archivo `menu.go` debe definir esas dos funciones, cada una imprimiendo varias líneas. Ejecuta el programa con `go run .` y luego genera el ejecutable con `go build`. Verifica que el ejecutable funcione sin usar `go run`.

Solución 1: La clave aquí es simplemente llamar a `fmt.Println` tres veces, una por dato. Recuerda el package `main`, el `import` de `fmt` y la función `main`.

```
package main

import "fmt"

func main() {
    fmt.Println("Esteban López")
    fmt.Println("Ciudad de México")
    fmt.Println("Go")
}
```

Explicación: cada llamada a `fmt.Println` imprime su texto y salta a la siguiente línea automáticamente. Cambia los datos por los tuyos y ejecútalo con `go run main.go`. Si olvidas el `import "fmt"`, el compilador te dirá "undefined: fmt": es el error más común y la solución es agregar la línea del import.

Solución 2: El truco está en que los caracteres especiales como `/` y `\` se escriben tal cual dentro de las comillas, pero la diagonal invertida `\` necesita un cuidado especial: en Go, dentro de un string, `\` inicia secuencias especiales como `\n`. Para mostrar una diagonal invertida literal debes escribirla doble: `\\`. Observa la primera línea del gato.

```
package main

import "fmt"

func main() {
    fmt.Println("  /\_/\")
    fmt.Println(" ( o.o )")
    fmt.Println("  > ^ <")
}
```

Explicación: en `" /\_/\"` cada `\\` produce una sola `\` en pantalla. Si escribieras una sola `\`, Go intentaría interpretarla como secuencia especial y el resultado sería un error o un carácter raro. Este detalle de escapar caracteres aparece en todos los lenguajes, así que vale la pena dominarlo desde ahora. Ejecuta el programa y compara tu salida con el dibujo pedido: los espacios al inicio de cada línea son parte del dibujo.

Solución 3: Necesitamos dos variables de tipo string, dos lecturas con `fmt.Scan` (recuerda el `&` antes de cada variable) y un mensaje final con `fmt.Printf` usando `%s` para insertar los valores.

```

package main

import "fmt"

func main() {
    var nombre string
    var color string

    fmt.Print("¿Cómo te llamas? ")
    fmt.Scan(&nombre)

    fmt.Print("¿Cuál es tu color favorito? ")
    fmt.Scan(&color)

    fmt.Printf("Hola %s, qué buen gusto tienes: el %s es un gran color.\n", nombre,
color)
}

```

Explicación: declaramos las variables con var, leemos cada dato con fmt.Scan pasándole la dirección (&), y al final usamos Printf con dos verbos %s que se reemplazan en orden: el primero con nombre y el segundo con color. Recuerda la limitación de Scan: si el usuario escribe un color de dos palabras como "azul cielo", solo se guardará "azul". Para este ejercicio pide colores de una sola palabra.

Solución 4: La lógica es una multiplicación encadenada: minutos = horas * 60, segundos = minutos * 60 (que es lo mismo que horas * 3600). Leemos las horas como entero con fmt.Scan y mostramos todo con Printf y el verbo %d.


```

package main

import "fmt"

func main() {
    var horas int

    fmt.Print("Escribe un número de horas: ")
    fmt.Scan(&horas)

    minutos := horas * 60
    segundos := minutos * 60

    fmt.Printf("%d horas equivalen a:\n", horas)
    fmt.Printf("- %d minutos\n", minutos)
    fmt.Printf("- %d segundos\n", segundos)
}

```

Explicación: calculamos minutos multiplicando por 60 y luego segundos multiplicando los minutos por 60 otra vez. Podríamos haber calculado segundos directamente como horas * 3600, pero hacerlo en dos pasos muestra mejor la lógica y es más fácil de verificar. La declaración corta minutos := crea la variable y le asigna el valor en un paso. Prueba con 2, con 0 y con 24 para verificar que los resultados tienen sentido: 24 horas deben dar 1440 minutos y 86400 segundos.

Solución 5: Aquí lo importante no es solo que el programa funcione, sino que los comentarios expliquen decisiones. El comentario de bloque al inicio describe el programa completo, y los comentarios de línea explican el porqué de la multiplicación por 7.

```

/*
Programa: edad de mascota en años humanos.
Pide el nombre y la edad de una mascota y estima
su edad equivalente en años humanos.
*/
package main

import "fmt"

func main() {
    var nombre string
    var edad int

    fmt.Print("¿Cómo se llama tu mascota? ")
    fmt.Scan(&nombre)

    fmt.Print("¿Cuántos años tiene? ")
    fmt.Scan(&edad)

    // La regla popular de "un año de mascota = 7 años humanos" es una
    // aproximación muy simplificada, pero sirve para este ejercicio
    // porque lo importante es practicar el cálculo, no la veterinaria.
    edadHumana := edad * 7

    fmt.Printf("%s tiene %d años de mascota.\n", nombre, edad)
    fmt.Printf("Eso equivale a unos %d años humanos. ¡Cuidala mucho!\n",
edadHumana)
}

```

Explicación: el comentario de bloque al inicio funciona como documentación del programa: cualquiera que abra el archivo entiende de qué trata sin leer el código. El comentario de línea antes del cálculo explica por qué multiplicamos por 7 y reconoce honestamente que es una aproximación. Ese tipo de honestidad en los comentarios es una marca de buen programador: prefiero un comentario que diga "esto es aproximado" a un número mágico sin explicación.

Solución 6: Este ejercicio junta todo: varios archivos, funciones separadas y generación del ejecutable. Primero crea una carpeta nueva e inicializa el módulo con `go mod init`. Luego crea los dos archivos.

Archivo `main.go`:

```

package main

import "fmt"

func main() {
    fmt.Println("=====")
    fmt.Println("  MI PRIMER PROGRAMA DOBLE  ")
    fmt.Println("=====")
    mostrarMenu()
    mostrarCreditos()
}

```

Archivo menu.go:

```

package main

import "fmt"

// mostrarMenu imprime las opciones del menú principal.
func mostrarMenu() {
    fmt.Println("--- MENÚ ---")
    fmt.Println("1. Saludar")
    fmt.Println("2. Calcular")
    fmt.Println("3. Salir")
}

// mostrarCreditos imprime los créditos del programa.
func mostrarCreditos() {
    fmt.Println("--- CRÉDITOS ---")
    fmt.Println("Programa escrito en Go.")
    fmt.Println("Curso de Lucas EGA Academy, unidad 1.")
}

```

Explicación: ambos archivos pertenecen a package main, por eso las funciones definidas en menu.go están disponibles en main.go sin necesidad de importar nada. Para ejecutar, usa go run . (el punto incluye todos los archivos). Para generar el ejecutable, usa go build -o miprograma . y luego ejecútalo directamente con ./miprograma (o miprograma.exe en Windows). Si por error ejecutas go run main.go, obtendrás "undefined: mostrarMenu" porque solo compilaste un archivo: ese error es la prueba de que entiendes cómo Go une los archivos de un paquete.

Resumen de la unidad

En esta unidad conociste Go desde sus raíces: un lenguaje creado en Google en 2007 por Robert Griesemer, Rob Pike y Ken Thompson, nacido de la frustración con compilaciones lentas, lenguajes demasiado complejos y la dificultad de programar para múltiples núcleos. Su filosofía es la simplicidad deliberada: pocas herramientas, bien diseñadas, que producen código fácil de leer y mantener. Viste que Go mueve el mundo real: Docker, Kubernetes y Terraform están escritos en Go, y empresas como Google, Uber, Twitch y Cloudflare lo usan en sistemas críticos, especialmente en servidores web, herramientas de línea de comandos, microservicios e infraestructura.

En la parte práctica instalaste Go en tu sistema operativo (Windows, Mac o Linux) y verificaste la instalación con `go version`. Entendiste la diferencia entre el antiguo `GOPATH` y los modernos módulos: hoy los proyectos se organizan con módulos, se inician con `go mod init` y viven en cualquier carpeta que elijas. Dominaste los comandos esenciales del toolchain: `go run` para ejecutar al momento, `go build` para generar ejecutables independientes, `go mod init` para iniciar proyectos y `go mod tidy` para ordenar dependencias.

Desarmaste el programa "Hola, mundo" pieza por pieza: `package main` marca un programa ejecutable, `import "fmt"` trae las herramientas de impresión, `func main` es el punto de entrada que Go ejecuta primero, y `fmt.Println` muestra texto en pantalla. Aprendiste las tres funciones de impresión (`Println`, `Print`, `Printf` con sus verbos `%s`, `%d` y `%f`), a leer datos del usuario con `fmt.Scan` (sin olvidar el `&`), a escribir comentarios útiles con `//` y `/* */`, y a formatear tu código con `gofmt -w` como un profesional.

También conociste los errores clásicos del principiante: olvidar `package main` o escribirlo mal, importar paquetes sin usarlos, poner la llave de apertura en la línea equivocada, no guardar antes de ejecutar y correr `go run` desde la carpeta incorrecta. Cada uno de estos errores te va a pasar al menos una vez; la diferencia es que ahora sabes exactamente qué significan y cómo corregirlos. Y adoptaste el flujo de trabajo profesional: editar en incrementos pequeños, probar con `go run`, leer el primer error de la lista, formatear con `gofmt` y compilar con `go build` cuando el programa esté listo para compartir.

En la siguiente unidad vamos a entrar al corazón de cualquier programa: las variables, las constantes y los tipos de datos. Aprenderás a guardar información en cajitas con nombre, qué valores tienen esas cajitas cuando las creas vacías, cómo Go infiere tipos automáticamente, cómo convertir entre tipos y por qué Go se niega a compilar si declaras una variable y no la usas. Todo lo que construimos en esta unidad (imprimir, leer datos, compilar) se vuelve mucho más poderoso cuando sabes manejar datos de verdad. Nos vemos en la unidad 2.

UNIDAD 2: Variables, constantes y tipos de datos

En la unidad 1 instalaste Go, conociste su historia y escribiste tus primeros programas. Esos programas hacían cosas, pero tenían una limitación enorme: no recordaban nada. Cada vez que se ejecutaban empezaban de cero, sin memoria de lo que el usuario escribió ni de los cálculos que hicieron. Un programa sin memoria es como una persona con amnesia total: puede hablar, pero no puede aprender, no puede acumular información y no puede tomar decisiones basadas en lo que ya pasó. Para que tus programas sean útiles de verdad necesitan memoria, y en programación la memoria tiene un nombre: variables.

En esta unidad vamos a dominar el sistema de datos de Go desde los cimientos. Aprenderás las dos formas de declarar variables (`var` y la declaración corta `:=`), cuándo usar cada una y qué diferencias reales hay entre ellas. Descubrirás el concepto de valor cero: en Go, toda variable nace con un valor predecible y seguro, nunca con basura aleatoria como en otros lenguajes. Conocerás las constantes y la curiosa palabra `iota`, que genera secuencias de valores automáticamente y es una de las características más elegantes del lenguaje. Recorrerás los tipos básicos uno por uno: enteros con y sin signo de distintos tamaños, números flotantes, booleanos, strings, y los tipos especiales `byte` y `rune` que te permitirán entender cómo Go maneja texto por dentro. Verás cómo Go infiere tipos para ahorrarte escritura sin perder seguridad, cómo convertir entre tipos de forma explícita (porque Go jamás convierte tipos a escondidas), y cómo el paquete `strconv` te ayuda a convertir texto en números y viceversa. También hablaremos del ámbito de las variables (dónde existe cada variable y dónde deja de existir), de la famosa regla que impide compilar si declaras una variable y no la usas, y de las buenas prácticas para nombrar tus variables como un profesional: `camelCase`, nombres exportados con mayúscula inicial y todo lo que hace que tu código se lea solo.

Por qué esta unidad es el cimiento de todo lo demás. Cada tema futuro del curso (condicionales, ciclos, funciones, estructuras, concurrencia) manipula variables y tipos de datos. Si entiendes a fondo cómo Go guarda, nombra y convierte datos, cada unidad siguiente será cuesta abajo. Si en cambio pasas de largo con dudas, esas dudas te perseguirán como una sombra: cada error de tipos te parecerá un misterio en vez de un mensaje claro. Así que tómate esta unidad con calma, escribe cada ejemplo, rompe cosas a propósito para ver los mensajes de error, y llega al final con la confianza de quien entiende la materia prima con la que se construye todo programa.

Una idea que te acompañará en toda la unidad: en Go, el compilador es tu aliado más estricto y más honesto. Te va a impedir compilar por cosas que en otros lenguajes pasarían desapercibidas: una variable sin usar, una conversión de tipos dudosa, un tipo mal inferido. Al principio puede sentirse como un maestro regañón, pero cada regla existe porque previene un bug real que alguien sufrió en producción. Cuando el compilador te marque un error en esta unidad, no lo veas como un obstáculo: vélo como un entrenamiento gratuito. Está forjando en ti hábitos que te harán un mejor programador en cualquier lenguaje.

Explicación detallada

Qué es una variable: cajitas con nombre y tipo

Una variable es un espacio de memoria con un nombre, donde guardas un valor para usarlo después. La analogía clásica, y la mejor para empezar, es la de las cajitas etiquetadas: imagina una repisa con cajitas, cada una con una etiqueta que dice qué contiene. La cajita etiquetada "edad" guarda el número 30. La cajita "nombre" guarda el texto "Esteban". Cuando tu programa necesita la edad, busca la cajita con esa etiqueta y usa su contenido. Cuando necesita cambiar la edad, abre la cajita y reemplaza el contenido. Esa es toda la magia: las variables le dan memoria a tu programa.

En Go, cada cajita tiene además un tipo, que dice qué clase de valores puede guardar. La cajita "edad" es de tipo entero: solo guarda números enteros. La cajita "nombre" es de tipo string: solo guarda texto. El tipo es un contrato entre tú y el compilador: tú prometes guardar solo valores de ese tipo, y el compilador promete avisarte si rompes la promesa. Esta verificación ocurre al compilar, antes de que el programa corra, que es exactamente cuando quieres enterarte de los errores: en tu computadora, no en la del usuario.

Hay tres operaciones fundamentales con variables: declarar (crear la cajita y ponerle etiqueta), asignar (meter un valor en la cajita) e inicializar (declarar y asignar en el mismo paso). En Go puedes hacer las tres por separado o combinadas, y cada forma tiene su momento ideal, como veremos en la siguiente sección. Por ahora quédate con el vocabulario: declarar es anunciar la variable, asignar es darle valor, inicializar es hacer ambas cosas a la vez.

Un detalle importante de la analogía de las cajitas: en Go las cajitas siempre tienen algo dentro, nunca están "vacías" en el sentido de contener basura. Cuando declaras una variable sin asignarle valor, Go la llena automáticamente con el valor cero de su tipo: 0 para números, "" (texto vacío) para strings, false para booleanos. Este comportamiento, que veremos a fondo más adelante, elimina toda una categoría de bugs que en otros lenguajes aparecen cuando lees una variable que nunca fue inicializada y obtienes un valor aleatorio de la memoria.

var vs := : las dos formas de declarar y cuándo usar cada una

Go te ofrece dos sintaxis para declarar variables, y elegir bien entre ellas es una de las primeras marcas de un programador que entiende el lenguaje. Vamos a verlas con calma.

La forma larga usa la palabra clave `var`, seguida del nombre, el tipo y opcionalmente el valor inicial.

```
var edad int
var nombre string = "Esteban"
var activo bool = true
```

En la primera línea declaramos `edad` como entero sin valor inicial: Go le asignará automáticamente el valor cero, que para `int` es 0. En las otras dos declaramos e inicializamos en el mismo paso, especificando el tipo de forma explícita. La palabra `var` le dice al compilador: "reserva espacio para una variable con este nombre y este tipo".

La forma corta usa el operador `:=`, que declara la variable e infiere su tipo automáticamente a partir del valor.

```
edad := 30
nombre := "Esteban"
activo := true
```

Aquí no escribimos el tipo: Go lo deduce. Como 30 es un número entero, `edad` será de tipo `int`. Como "Esteban" es texto entre comillas, `nombre` será `string`. Como `true` es un valor booleano, `activo` será `bool`. El operador `:=` hace tres cosas a la vez: declara la variable, infiere su tipo y le asigna el valor. Es la forma más usada en el código Go del día a día.

Entonces, ¿cuándo usar cada una? La regla práctica es simple: usa `:=` dentro de funciones siempre que puedas, porque es más corta y el tipo se deduce con claridad. Usa `var` en tres situaciones: primero, cuando declaras variables a nivel de paquete (fuera de cualquier función), porque `:=` no está permitido ahí; segundo, cuando quieres declarar sin inicializar y aprovechar el valor cero, como `var contador int`; tercero, cuando necesitas un tipo específico distinto al que Go inferiría, como `var temperatura float32 = 36.6` (con `:=`, Go inferiría `float64`, no `float32`).

Hay dos restricciones de `:=` que debes memorizar. Primera: `:=` solo funciona dentro de funciones, nunca a nivel de paquete. Si intentas usar `:=` fuera de una función, el compilador se quejará. Segunda: `:=` exige declarar al menos una variable nueva a la izquierda; no puedes usar `:=` para reasignar variables que ya existen. Para cambiar el valor de una variable existente usas `=` simple. Mira la diferencia:

```
edad := 30    // declara edad y le asigna 30 (bien)
edad := 31    // ERROR: no se puede re-declarar con :=
edad = 31     // reasigna el valor de la variable existente (bien)
```

El error que obtendrás con la segunda línea es "no new variables on left side of `:=`". Suena críptico la primera vez, pero significa exactamente eso: a la izquierda de `:=` debe haber al menos una variable nueva. Cuando veas ese mensaje, cambia `:=` por `=` y listo.

Una trampa sutil relacionada: si usas `:=` dentro de un bloque (por ejemplo, dentro de un `if`) con el mismo nombre de una variable de afuera, no estás modificando la variable de afuera: estás creando una variable nueva que solo existe dentro del bloque, un fenómeno llamado shadowing o sombreado. Es una fuente clásica de bugs y la veremos con ejemplos en la sección de ámbito.

El valor cero: en Go nada nace con basura

En muchos lenguajes, declarar una variable sin inicializarla es jugar a la ruleta rusa: la variable contiene lo que sea que había antes en esa zona de memoria, un valor impredecible que cambia entre ejecuciones. Los bugs causados por esto son de los más difíciles de encontrar, porque el programa a veces funciona y a veces no, dependiendo de la basura que haya en memoria. Go eliminó este problema de raíz con una regla simple y poderosa: toda variable, al declararse, recibe automáticamente el valor cero de su tipo.

Los valores cero son fáciles de memorizar: 0 para todos los tipos numéricos (`int`, `float64`, etcétera), `false` para `bool`, `""` (la cadena vacía) para `string`, y `nil` para tipos como punteros, slices y mapas, que conoceremos en unidades futuras. `nil` es el "nada" de Go: significa ausencia de valor de forma explícita y segura.


```
package main

import "fmt"

func main() {
    var entero int
    var decimal float64
    var texto string
    var logico bool

    fmt.Printf("entero: %d\n", entero)
    fmt.Printf("decimal: %f\n", decimal)
    fmt.Printf("texto: '%s'\n", texto)
    fmt.Printf("logico: %t\n", logico)
}
```

La salida es completamente predecible:

```
entero: 0
decimal: 0.000000
texto: ''
logico: false
```

Por qué importa esto en la práctica. Primero, te permite declarar variables con `var` sin miedo: sabes exactamente qué contienen desde el primer momento. Segundo, hace que muchas estructuras de datos funcionen correctamente "gratis": por ejemplo, un contador declarado con `var contador int` empieza en 0, que es justo lo que quieres antes de empezar a contar. Tercero, elimina una categoría entera de bugs no deterministas: en Go, un programa que lee una variable recién declarada siempre obtiene el valor cero, en cualquier computadora, en cualquier ejecución. Esa predictibilidad es una de las razones por las que los sistemas escritos en Go son tan confiables.

Un matiz importante: el valor cero es un punto de partida seguro, no necesariamente el valor correcto para tu lógica. Si declaras `var promedio float64` y luego divides una suma entre un conteo, el cero inicial está bien porque lo vas a reemplazar. Pero si olvidas asignarle el valor real, tu programa calculará con 0 sin quejarse y el resultado será silenciosamente incorrecto. El valor cero te protege de la basura aleatoria, pero no te protege de olvidar tu propia lógica: esa parte sigue siendo tu responsabilidad.

Constantes: valores que no cambian nunca

No todo en un programa debe poder cambiar. El número de días de la semana, el valor de pi, la tasa de IVA, el nombre de tu academia: son valores fijos que, si alguien los modifica por accidente en medio del programa, generan bugs terribles. Para esos casos existen las constantes: variables con superpoderes de inmutabilidad, valores que se fijan una vez y el compilador garantiza que nadie los cambie jamás.

En Go se declaran con la palabra clave `const`, y la sintaxis se parece a la de `var`.

```
const diasSemana = 7
const pi float64 = 3.14159
const nombreAcademia string = "Lucas EGA Academy"
```

Observa que en la primera no especificamos el tipo: las constantes sin tipo explícito son "constantes no tipadas", y Go les asigna el tipo según cómo se usen. Esto las hace flexibles: `const diasSemana = 7` puede usarse donde se espere un `int`, un `float64` o casi cualquier tipo numérico, sin conversiones. En cambio, `const pi float64 = 3.14159` es una constante tipada: solo puede usarse como `float64`. Para constantes numéricas simples, la forma no tipada suele ser más cómoda.

Si intentas modificar una constante, el compilador te detiene en seco:

```
const iva = 0.16
iva = 0.20 // ERROR: cannot assign to iva (constant)
```

Ese error es exactamente lo que quieres: mejor enterarte al compilar que descubrir en producción que alguien cambió la tasa de IVA a mitad del cálculo. Las constantes también pueden declararse en bloque, igual que los imports y las variables:

```
const (
    lunes      = "Lunes"
    martes     = "Martes"
    miercoles  = "Miércoles"
    servidor   = "api.lucasegaacademy.com"
    puerto     = 8080
    maxIntentos = 3
)
```

El bloque `const` es la forma idiomática de agrupar constantes relacionadas: días de la semana, configuración de un servidor, límites del programa. Cuando veas un bloque `const` al inicio de

un archivo Go, léelo como "estos son los valores fijos que gobiernan este programa". Es documentación viva: cualquiera entiende las reglas del programa sin leer la lógica.

Buenas prácticas con constantes: úsalas para todo valor que tenga significado y no deba cambiar: precios base, URLs, límites, mensajes estándar. Evita los "números mágicos": esos valores sueltos en medio del código como `total * 0.16`, donde nadie sabe qué significa 0.16 sin contexto. Es mucho más claro escribir `total * iva` con `const iva = 0.16` declarada arriba. Tu código se vuelve autodocumentado y, si la tasa cambia algún día, la cambias en un solo lugar en vez de buscarla por todo el programa.

iota: el contador automático de Go

Dentro de los bloques `const` existe una palabra mágica llamada `iota` que merece su propia sección, porque es una de las características más elegantes y más go-idiomáticas del lenguaje. `iota` es un contador que empieza en 0 y aumenta en 1 con cada línea del bloque `const`. Suena simple, y lo es, pero resuelve un problema muy común con una elegancia sorprendente: definir secuencias de valores relacionados, como los días de la semana numerados, los niveles de un juego o los estados de un pedido.

```
const (  
    domingo = iota // 0  
    lunes      // 1  
    martes     // 2  
    miercoles  // 3  
    jueves     // 4  
    viernes    // 5  
    sabado     // 6  
)
```

Solo escribimos `iota` en la primera línea; en las siguientes se repite implícitamente la expresión, así que cada constante recibe el siguiente número automáticamente. El resultado es idéntico a haber escrito los números a mano, pero con una ventaja enorme: si insertas un valor nuevo en medio, no tienes que renumerar nada a mano. `iota` lo hace por ti.

Puedes usar `iota` en expresiones para generar secuencias más interesantes. Un patrón clásico es generar potencias de dos para representar permisos o banderas, usando el desplazamiento de bits `<<`:

```
const (
    permisoLeer      = 1 << iota // 1 (2^0)
    permisoEscribir   // 2 (2^1)
    permisoEjecutar   // 4 (2^2)
    permisoEliminar   // 8 (2^3)
)
```

Aquí `1 << iota` significa "el número 1 desplazado `iota` posiciones a la izquierda en binario", lo que produce 1, 2, 4, 8. Cada permiso es una potencia de dos distinta, lo que permite combinarlos con operaciones de bits: un usuario puede tener `permisoLeer + permisoEscribir = 3`, y el programa puede verificar cada permiso por separado. Este patrón aparece en código real constantemente: sistemas de permisos, opciones de configuración, banderas de estado.

Otro uso elegante es saltar valores con el identificador vacío `_`. Si quieres que tu enumeración empiece en 1 en vez de 0, consumes el primer `iota` con `_`:

```
const (
    _ = iota // descarta el 0
    enero    // 1
    febrero  // 2
    marzo    // 3
)
```

Y puedes multiplicar `iota` para secuencias arbitrarias:

```
const (
    _ = iota // 0 (descartado)
    kb float64 = 1 << (10 * iota) // 1024
    mb           // 1048576
    gb           // 1073741824
)
```

Este ejemplo, tomado de la propia biblioteca estándar de Go, define kilobytes, megabytes y gigabytes usando desplazamientos de bits. No necesitas entender los bits a fondo hoy; lo importante es ver el patrón: `iota` convierte secuencias tediosas de escribir a mano en declaraciones de una línea, legibles y a prueba de errores de numeración.

Errores comunes con `iota`: primero, olvidar que `iota` se reinicia a 0 en cada bloque `const` nuevo. Si tienes dos bloques `const` separados, cada uno empieza su propio conteo desde cero. Segundo, pensar que `iota` "guarda" el valor anterior cuando interrumpes la secuencia con un valor fijo: en realidad `iota` sigue contando líneas aunque una línea tenga un valor explícito.

Tercero, usar `iota` fuera de un bloque `const`: no tiene sentido ahí y el compilador se quejará. Recuerda la regla de oro: `iota` cuenta líneas dentro de su bloque `const`, empezando en cero.

Los tipos enteros: `int`, `int8`, `int16`, `int32`, `int64` y sus primos sin signo

Los números enteros son el tipo de dato más usado en programación: contadores, edades, cantidades, índices. Go ofrece una familia completa de tipos enteros, y entender por qué existen tantos tamaños te hará tomar mejores decisiones.

El tipo `int` es el entero de propósito general. Su tamaño depende de la plataforma: en sistemas de 64 bits (casi todo hoy en día) guarda valores de 64 bits. Para el 95 por ciento de tu código, `int` es la respuesta correcta: es el tipo que Go infiere cuando escribes un número entero, es el que usan los índices y las funciones de la biblioteca estándar, y no tienes que pensar en límites. Regla de oro: si dudas, usa `int`.

Los tipos con tamaño explícito (`int8`, `int16`, `int32`, `int64`) existen para cuando el tamaño sí importa. `int8` guarda de -128 a 127, `int16` de -32768 a 32767, `int32` unos ± 2 mil millones, e `int64` unos ± 9 trillones. ¿Cuándo te importa el tamaño? Cuando trabajas con formatos de archivo binarios, protocolos de red o hardware, donde cada byte cuenta; o cuando guardas millones de valores en memoria y usar `int64` para números que caben en `int8` desperdiciaría memoria. Para programas normales, no te compliques: `int` y punto.

La familia `uint` (`uint`, `uint8`, `uint16`, `uint32`, `uint64`) guarda solo números no negativos, es decir, sin signo. Al no necesitar el bit del signo, un `uint` del mismo tamaño guarda el doble de valores positivos que su primo `int`. `uint8`, por ejemplo, va de 0 a 255. Se usan cuando los valores negativos no tienen sentido: longitudes, conteos, índices de memoria. Pero ojo con un detalle traicionero: restar con `uint` puede dar resultados sorprendentes, porque no hay negativos a donde ir. Si haces `var a uint = 3; var b uint = 5` y calculas `a - b`, no obtienes -2: obtienes un número gigantesco, porque el resultado "da la vuelta" por debajo del cero. Este fenómeno se llama `underflow`, y es una fuente clásica de bugs. Por eso muchos programadores experimentados prefieren `int` incluso para conteos, a menos que haya una razón concreta para usar `uint`.

Dos tipos con nombre especial merecen mención: `byte` es un alias de `uint8`, y se usa para representar datos binarios crudos, `byte` por `byte`. `rune` es un alias de `int32`, y representa un punto de código Unicode, es decir, un carácter. Los veremos a fondo en la sección de strings, pero desde ahora memoriza: `byte` = `uint8` (datos crudos), `rune` = `int32` (caracteres). Cuando veas `[]byte` en código Go, lee "una secuencia de bytes"; cuando veas `rune`, lee "un carácter".

Un punto crucial: en Go, los tipos numéricos distintos son incompatibles entre sí, aunque guarden valores parecidos. No puedes sumar un `int` con un `int64` sin convertir explícitamente

uno de los dos. Esto es intencional: las conversiones silenciosas entre tamaños son una fuente histórica de bugs (pérdida de precisión, desbordamientos invisibles). Go prefiere que seas explícito y que cada conversión sea una decisión consciente. La sintaxis de conversión es simple: `T(v)`, donde `T` es el tipo destino y `v` el valor. Lo veremos con ejemplos en la sección de conversiones.

Números flotantes, booleanos y strings

Los `float32` y `float64` representan números con decimales: precios, promedios, temperaturas, coordenadas. `float64` es el de uso general: tiene el doble de precisión que `float32` y es el tipo que Go infiere cuando escribes un decimal como `3.14`. Usa `float64` siempre, a menos que trabajes con gráficos, hardware o formatos que exijan `float32` explícitamente. Un detalle honesto que debes conocer desde el inicio: los flotantes son aproximaciones, no valores exactos. En binario, `0.1 + 0.2` no da exactamente `0.3`, sino `0.30000000000000004`. Esto no es un bug de Go: es una característica de cómo las computadoras representan decimales, y afecta a todos los lenguajes. Para dinero real en sistemas financieros serios se usan técnicas especiales (como trabajar en centavos con enteros); para el resto de los casos, `float64` es perfecto.

El tipo `bool` guarda valores lógicos: `true` o `false`, verdadero o falso. Parece trivial, pero los booleanos son el corazón de la toma de decisiones en programación: cada `if`, cada `while`, cada validación depende de un booleano. Su valor cero es `false`, lo cual tiene sentido: una condición recién creada empieza siendo falsa hasta que se demuestre lo contrario. Los booleanos se combinan con operadores lógicos: `&&` (Y), `||` (O) y `!` (NO), que veremos en detalle en la unidad de condicionales.

El tipo `string` guarda texto: nombres, mensajes, direcciones. En Go, los strings se escriben entre comillas dobles ("`Hola`") y son inmutables: una vez creado un string, su contenido no puede cambiar. Cuando "modificas" un string, en realidad Go crea uno nuevo. Esto suena ineficiente, pero permite optimizaciones internas y elimina bugs de modificación accidental. Los strings en Go son secuencias de bytes en codificación UTF-8, lo que significa que pueden contener cualquier carácter del mundo: acentos, eñes, emojis, chino, árabe. Para texto de varias líneas existen las comillas invertidas (backticks), que crean "raw strings" donde todo se toma literalmente, incluyendo saltos de línea:

```
poema := `Rosa fresca,  
rosa fragante,  
aprendiendo Go  
muy elegante.`
```

Dentro de un raw string no necesitas escapar nada: lo que ves es lo que hay. Son perfectos para plantillas de texto, consultas SQL o cualquier bloque multilínea.

Ahora, la distinción entre byte y rune que prometimos. Un string en Go es una secuencia de bytes, y para caracteres ASCII (letras sin acento, dígitos, símbolos básicos) cada carácter ocupa exactamente un byte. Pero los caracteres acentuados o especiales en UTF-8 ocupan más de un byte: la "ñ" ocupa 2 bytes, y un emoji puede ocupar 4. Esto causa una confusión clásica: `len("niño")` devuelve 5, no 4, porque cuenta bytes, no caracteres. Si necesitas trabajar carácter por carácter, convierte el string a `[]rune`: cada rune es un carácter completo, sin importar cuántos bytes ocupe. Mira la diferencia:

```
texto := "niño"
fmt.Println(len(texto))           // 5 (bytes)
fmt.Println(len([]rune(texto)))   // 4 (caracteres)
```

Para recorrer texto mostrando cada carácter correctamente, el ciclo for con range sobre un string itera por runes automáticamente. Lo veremos en la unidad de ciclos; por ahora quédate con la idea: byte para datos crudos, rune para caracteres, y len cuenta bytes.

Inferencia de tipos: Go adivina, tú verificas

Ya viste la inferencia en acción con `:=`, pero vale la pena entender sus reglas con precisión, porque "Go deduce el tipo" no significa "Go adivina a ciegas": sigue reglas deterministas que debes conocer.

Cuando escribes `x := 42`, Go asigna tipo int. Cuando escribes `x := 3.14`, asigna float64. Cuando escribes `x := "hola"`, asigna string. Cuando escribes `x := true`, asigna bool. Estas son las reglas por defecto para literales (valores escritos directamente en el código). Son sensatas: el entero más general, el flotante más preciso, el texto estándar, el booleano único.

La inferencia también funciona con var sin tipo explícito: `var x = 42` crea un int, igual que con `:=`. La diferencia entre `var x = 42` y `x := 42` es solo sintáctica dentro de funciones; fuera de funciones solo var está permitido.

Puedes verificar el tipo inferido de cualquier variable con el verbo `%T` de `Printf`, que muestra el tipo en vez del valor. Es una herramienta de aprendizaje excelente: cuando dudes qué tipo infirió Go, pregúntale directamente.

```

a := 42
b := 3.14
c := "hola"
d := true
fmt.Printf("%T %T %T %T\n", a, b, c, d) // int float64 string bool

```

La inferencia brilla cuando el tipo es obvio y verboso de escribir, como en estructuras complejas que veremos después. Pero hay casos donde la inferencia te puede jugar en contra: si necesitas un float32 pero Go infiere float64, o si necesitas un int64 para una función que lo exige. En esos casos, declara el tipo explícitamente con var o convierte el valor. La regla de oro: deja que Go infiera cuando el tipo inferido es el que quieres; especifica el tipo cuando necesitas algo distinto. Nunca pelees contra la inferencia por purismo: úsala a tu favor.

Conversión de tipos: en Go nada se convierte a escondidas

En muchos lenguajes, mezclar tipos "funciona" porque el lenguaje convierte silenciosamente: sumas un entero con un flotante y obtienes un flotante sin enterarte. Go rechaza esa magia. Si intentas operaciones entre tipos distintos, el compilador se niega: "mismatched types". La filosofía es clara: las conversiones entre tipos pueden perder información (un float64 a int pierde los decimales; un int64 a int8 puede desbordarse), así que deben ser decisiones explícitas y visibles en el código.

La sintaxis de conversión es T(v): el tipo destino entre paréntesis alrededor del valor.

```

entero := 42
decimal := float64(entero) // int -> float64: 42.0
otroEntero := int(3.99)    // float64 -> int: 3 (se truncan los decimales, no se redondea)
texto := string(65)        // int -> string: "A" (¡cuidado! convierte el código Unicode, no el dígito)

```

Ese último ejemplo merece una advertencia grande: string(65) no produce "65", produce "A", porque convierte el número como punto de código Unicode. Es una trampa clásica. Para convertir números a su representación en texto (el 65 al texto "65") se usa el paquete strconv, que vemos a continuación.

Las conversiones numéricas entre tipos enteros y flotantes siguen reglas predecibles: al convertir de flotante a entero se truncan los decimales (3.99 se vuelve 3, -3.99 se vuelve -3); al convertir de un entero grande a uno pequeño, si el valor no cabe, se queda con los bits bajos, produciendo un resultado que puede sorprender. Por eso la regla de oro: convierte solo

cuando estés seguro de que el valor cabe en el destino, o cuando hayas validado el rango antes.

strconv: el puente entre texto y números

Los programas reales viven convirtiendo texto a números y números a texto: lees la edad del usuario como texto y necesitas un entero para calcular; calculas un total y necesitas mostrarlo como texto en pantalla. El paquete strconv (string conversion) es la herramienta oficial para estas conversiones, y dominarlo es esencial.

Las funciones más usadas son cuatro. Atoi convierte un string a int ("ASCII to integer"): edad, err := strconv.Atoi("25"). Itoa hace lo inverso, int a string: texto := strconv.Itoa(25). ParseFloat convierte un string a flotante, especificando los bits de precisión: valor, err := strconv.ParseFloat("3.14", 64). FormatFloat convierte un flotante a string controlando el formato: texto := strconv.FormatFloat(3.14159, 'f', 2, 64), que produce "3.14".

Notarás que Atoi y ParseFloat devuelven dos valores: el resultado y un error. Esto es porque la conversión puede fallar: ¿qué pasa si intentas convertir "hola" a número? En Go, las funciones que pueden fallar devuelven un error como segundo valor, y es tu responsabilidad revisarlo. Es el famoso manejo de errores de Go, que veremos a fondo en su unidad; por ahora, la forma mínima correcta es:

```
edadTexto := "25"
edad, err := strconv.Atoi(edadTexto)
if err != nil {
    fmt.Println("Error: eso no es un número válido")
    return
}
fmt.Printf("Edad duplicada: %d\n", edad*2)
```

Si el texto no es un número válido, Atoi devuelve el error y el programa muestra el mensaje en vez de calcular con basura. Este patrón (revisar err != nil) lo verás en todo el código Go del mundo: es la forma idiomática de manejar lo inesperado.

Para conversiones simples de número a texto dentro de fmt.Printf ni siquiera necesitas strconv: los verbos %d, %f y %s ya convierten al mostrar. strconv brilla cuando necesitas el texto como valor para guardarlo, compararlo o enviarlo a otro lado, no solo para mostrarlo.

Ámbito (scope): dónde vive cada variable

El ámbito de una variable es la región del código donde esa variable existe y puede usarse. En Go, el ámbito lo determinan las llaves `{}`: una variable existe desde su declaración hasta el cierre del bloque donde fue declarada. Fuera de ese bloque, la variable no existe: intentar usarla es un error de compilación.

Hay tres niveles principales. Las variables a nivel de paquete se declaran fuera de cualquier función, existen durante todo el programa y son visibles para todas las funciones del paquete. Se declaran con `var` (nunca con `:=`) y se usan para configuración global o estado compartido, aunque los programadores experimentados las usan con moderación porque el estado global complica las pruebas.

```
package main

import "fmt"

var version = "1.0.0" // ámbito de paquete: visible en todo el archivo

func main() {
    nombre := "Esteban" // ámbito de función: visible solo dentro de main
    fmt.Println(version, nombre)
    saludar()
}

func saludar() {
    fmt.Println("Versión:", version) // bien: version es de paquete
    // fmt.Println(nombre) // ERROR: nombre no existe aquí
}
```

Las variables de función viven dentro de su función y mueren cuando la función termina. Cada llamada a la función crea variables frescas: no recuerdan valores de llamadas anteriores. Las variables de bloque viven dentro de un par de llaves anidado (un `if`, un `for`, un bloque suelto) y solo existen ahí dentro.

El sombreado (shadowing) ocurre cuando declaras con `:=` una variable con el mismo nombre que una de un ámbito exterior, dentro de un bloque interno: creas una variable nueva que "oculta" la exterior dentro de ese bloque, y los cambios no afectan a la original. Es legal en Go, pero es una fuente legendaria de bugs sutiles:

```

mensaje := "afuera"
if true {
    mensaje := "adentro" // variable NUEVA, sombrea la exterior
    fmt.Println(mensaje) // "adentro"
}
fmt.Println(mensaje) // "afuera": la exterior no cambió

```

La forma correcta de modificar la variable exterior es usar `=` en vez de `:=` dentro del bloque. Cuando un valor "no se actualiza" misteriosamente dentro de un `if` o un `for`, el sombreado es el primer sospechoso: revisa si usaste `:=` donde debías usar `=`.

Variables no usadas: el error más famoso de Go

Todo programador de Go ha visto este error: "declared and not used". Significa exactamente lo que dice: declaraste una variable y nunca la usaste, y Go se niega a compilar. Al principio parece una tiranía: ¿por qué el compilador se mete en cómo organizo mi código? La respuesta tiene historia: en proyectos grandes, las variables sin usar suelen ser restos de código a medio refactorizar, y esos restos confunden a quien lee el código después ("¿esto se usa en algún lado? ¿lo puedo borrar?"). Al prohibirlas, Go garantiza que todo lo declarado tiene un propósito demostrable.

```

func main() {
    nombre := "Esteban"
    edad := 30
    fmt.Println(nombre)
    // ERROR: edad declared and not used
}

```

Las soluciones son tres, según el caso. Si la variable fue un experimento o un resto, bórrala: era ruido. Si la vas a usar después pero aún no, coméntala o bórrala temporalmente: el código a futuro no se escribe por adelantado. Si necesitas declarar algo y conscientemente ignorarlo (por ejemplo, un valor de retorno que no te interesa), usa el identificador vacío `_`: es la forma oficial de decirle a Go "sé lo que hago, descarta esto".

```

_, err := strconv.Atoi("25") // ignoro el número, solo me importa el error

```

El identificador vacío también aplica a imports: si importas un paquete solo por sus efectos secundarios (raro, pero existe), lo importas como `_ "paquete"`. No lo necesitarás pronto, pero cuando veas `_` en código ajeno ya sabrás qué significa: "esto se declara a propósito y se ignora a propósito".

Un consejo práctico: mientras desarrollas, este error aparecerá cuando comentes código temporalmente o cuando declares variables "para después". No luches contra él: es un recordatorio barato de mantener tu código limpio. Los programadores de otros lenguajes pagan herramientas de análisis para detectar código muerto; en Go viene gratis en el compilador.

Buenas prácticas para nombrar variables

Los nombres son la documentación más leída de tu programa: se leen cientos de veces más de las que se escriben. Go tiene convenciones claras, y seguirlas hace que tu código se sienta nativo.

La convención principal es camelCase: la primera palabra en minúsculas y cada palabra siguiente con mayúscula inicial, sin guiones ni guiones bajos. Ejemplos: nombreUsuario, edadMaxima, totalVentas, esValido. Los guiones bajos en nombres de variables son raros en Go (a diferencia de Python); úsalos solo en casos especiales como constantes de prueba o nombres generados.

La mayúscula inicial tiene un significado técnico crucial en Go: marca lo que está exportado, es decir, visible desde otros paquetes. Una variable o función que empieza con mayúscula (Nombre, CalcularTotal) puede usarse desde cualquier paquete que importe el tuyo. Una que empieza con minúscula (nombre, calcularTotal) es privada: solo existe dentro de su paquete. Esta es la forma en que Go implementa la visibilidad pública/privada, sin palabras clave como public o private. Dentro de tus programas de un solo paquete la distinción no afecta, pero acostúmbrate desde ahora: minúscula para lo interno, mayúscula para lo que otros pueden usar.

Sobre la longitud de los nombres: en Go se prefieren nombres cortos cuando el contexto es claro. En un ciclo corto, `i` para el índice es perfectamente idiomático; en una función de tres líneas, variables como `n` o `s` son aceptables si su significado es obvio. Pero cuando el alcance crece, el nombre debe crecer: una variable que vive durante 200 líneas merece un nombre descriptivo como `intentosFallidos`, no `f`. La regla de oro: mientras más vive una variable y más lejos se usa de donde se declaró, más descriptivo debe ser su nombre. Un nombre como `x` en un programa de 500 líneas es un misterio; `totalAcumulado` se explica solo.

Evita abreviaturas crípticas (`usr`, `calc`, `tmp`) salvo las muy establecidas (`i`, `err`, `ok`). Prefiere nombres que digan qué es o qué hace: `edadUsuario` mejor que `e`, `errorLectura` mejor que `er`. Y usa nombres en el idioma de tu equipo: si tu equipo habla español, nombres en español están bien; si el código es para la comunidad internacional, el inglés es el estándar. Lo importante es la consistencia: no mezcles idiomas en el mismo proyecto.

Finalmente, nombra los booleanos como preguntas: `esValido`, `tienePermiso`, `estaActivo`. Cuando leas `if esValido` en tu código, sonará como español natural: "si es válido". Esa legibilidad es la meta de todo buen nombre: que el código se lea como una explicación de lo que hace.

Constantes no tipadas: la flexibilidad silenciosa

En la sección de constantes mencionamos de pasada las constantes no tipadas, pero merecen un momento extra porque son una idea sutil que afecta cómo escribes código numérico todos los días. Cuando declaras `const pi = 3.14159` sin especificar tipo, Go no le asigna `float64` de inmediato: la guarda como una constante "sin tipo", con precisión arbitraria, y solo le asigna un tipo concreto cuando la usas en un contexto que lo exige.

En la práctica, esto significa que una misma constante puede comportarse como distintos tipos según dónde aparezca:

```
const factor = 3

var a int = factor          // factor actúa como int
var b float64 = factor      // factor actúa como float64
var c int64 = factor * 2    // las operaciones también son flexibles
```

Si `factor` hubiera sido una variable (`var factor = 3`, de tipo `int`), la línea `var b float64 = factor` no compilaría: Go no convierte variables implícitamente. Pero con constantes no tipadas, el compilador permite esta flexibilidad porque conoce el valor exacto en tiempo de compilación y puede verificar que cabe en el tipo destino sin pérdida. Es lo mejor de los dos mundos: comodidad sin sacrificar seguridad.

Esta flexibilidad tiene límites sanos. Si la constante no cabe en el tipo destino, el compilador se queja. Por ejemplo, `const grande = 1000` usada como `var x int8 = grande` falla, porque 1000 no cabe en un `int8` (máximo 127). El compilador hace la verificación por ti en tiempo de compilación, cuando corregir es barato.

Las constantes tipadas (`const pi float64 = 3.14159`) renuncian a esta flexibilidad a cambio de precisión en la intención: le dicen a quien lee el código "este valor es `float64` y nada más". Úsalas cuando el tipo forma parte del significado: una constante de configuración que una función espera como `float32`, por ejemplo. Para constantes numéricas de propósito general (límites, factores, conteos), la forma no tipada es la idiomática.

Un detalle final que confunde a principiantes: las constantes no tipadas también aplican a literales sueltos. Cuando escribes `x := 5`, el literal 5 es una constante no tipada que adopta el

tipo int por defecto. Cuando escribes `var f float32 = 3.14`, el literal 3.14 adopta float32 porque el contexto lo exige. Esta es la razón por la que rara vez necesitas conversiones al trabajar con literales: los literales son flexibles por diseño, y solo las variables exigen conversiones explícitas.

Ejemplos paso a paso

Ejemplo 1: Declarando variables de todas las formas

Este ejemplo muestra las tres formas de declarar variables en un solo programa, para que compares la sintaxis lado a lado. Cópialo, ejecútalo y observa que las tres formas producen variables perfectamente utilizables.

```
package main

import "fmt"

func main() {
    // Forma 1: var con tipo explícito, sin valor inicial (usa el valor cero)
    var contador int
    var mensaje string

    // Forma 2: var con tipo explícito e inicialización
    var edad int = 30
    var nombre string = "Esterban"

    // Forma 3: declaración corta con inferencia de tipos
    ciudad := "Guadalajara"
    temperatura := 24.5
    activo := true

    fmt.Printf("contador=%d mensaje='%s'\n", contador, mensaje)
    fmt.Printf("edad=%d nombre='%s'\n", edad, nombre)
    fmt.Printf("ciudad=%s temperatura=%.1f activo=%t\n", ciudad, temperatura,
activo)
    fmt.Printf("Tipos: %T %T %T\n", ciudad, temperatura, activo)
}
```

Salida:

```
contador=0 mensaje=''
edad=30 nombre=Esteban
ciudad=Guadalajara temperatura=24.5 activo=true
Tipos: string float64 bool
```

Observa los detalles: contador vale 0 y mensaje es texto vacío, los valores cero de sus tipos. La última línea usa %T para mostrar los tipos inferidos: string, float64 y bool, exactamente lo que esperarías. Juega con este ejemplo: cambia valores, agrega variables con var sin inicializar de otros tipos y verifica sus valores cero con %v.

Ejemplo 2: El valor cero en acción

Un programa que demuestra que en Go puedes confiar en los valores iniciales. Declara variables de varios tipos sin asignarles nada y úsalas directamente en operaciones seguras.

```
package main

import "fmt"

func main() {
    var visitas int
    var saldo float64
    var nombre string
    var registrado bool

    // Los valores cero son seguros para operar de inmediato
    visitas = visitas + 1
    saldo = saldo + 100.50

    fmt.Printf("visitas: %d\n", visitas)
    fmt.Printf("saldo: %.2f\n", saldo)
    fmt.Printf("nombre: '%s' (longitud %d)\n", nombre, len(nombre))
    fmt.Printf("registrado: %t\n", registrado)

    if nombre == "" {
        fmt.Println("El nombre está vacío: es el valor cero de string")
    }
}
```

Salida:

```
visitas: 1
saldo: 100.50
nombre: '' (longitud 0)
registrado: false
El nombre está vacío: es el valor cero de string
```

La lección: incrementar un contador que empezó en cero funciona a la primera, sin inicialización manual. Comparar un string recién declarado con "" es una forma válida de saber si se le asignó algo. Este comportamiento predecible es el que elimina los bugs de "basura en memoria" de otros lenguajes.

Ejemplo 3: Constantes e iota para un sistema de días y niveles

Este ejemplo combina bloques const, iota y constantes con nombre para modelar dos situaciones reales: los días de la semana numerados y los niveles de membresía de una academia.


```

package main

import "fmt"

const (
    domingo = iota
    lunes
    martes
    miercoles
    jueves
    viernes
    sabado
)

const (
    nivelGratis = iota
    nivelBasico
    nivelPro
    nivelEmpresarial
)

const nombreAcademia = "Lucas EGA Academy"

func main() {
    fmt.Printf("El miércoles es el día %d de la semana\n", miercoles)
    fmt.Printf("El sábado es el día %d\n", sabado)
    fmt.Printf("Nivel Pro = %d, Nivel Empresarial = %d\n", nivelPro,
nivelEmpresarial)
    fmt.Println("Academia:", nombreAcademia)

    // Las constantes hacen el código autodocumentado
    hoy := miercoles
    if hoy == miercoles {
        fmt.Println("Hoy hay clase nueva en la academia")
    }
}

```

Salida:

```

El miércoles es el día 3 de la semana
El sábado es el día 6
Nivel Pro = 2, Nivel Empresarial = 3
Academia: Lucas EGA Academy
Hoy hay clase nueva en la academia

```

Nota que cada bloque const tiene su propio iota empezando en 0: los días y los niveles son secuencias independientes. Y observa cómo comparar hoy == miercoles se lee mucho mejor que hoy == 3: el nombre explica el significado del número. Si mañana decides que la semana empieza en lunes, cambias un bloque y todo el programa sigue funcionando.

Ejemplo 4: Conversiones de tipos explícitas

Un programa que muestra conversiones correctas entre tipos numéricos y la trampa clásica de string(65). Ejecútalo y estudia cada línea.

```
package main

import "fmt"

func main() {
    entero := 42
    decimal := 3.99

    // Conversiones numéricas explícitas
    comoDecimal := float64(entero)
    comoEntero := int(decimal) // trunca: 3.99 -> 3

    fmt.Printf("entero %d como float64: %.2f\n", entero, comoDecimal)
    fmt.Printf("decimal %.2f como int: %d (se truncó, no se redondeó)\n", decimal,
comoEntero)

    // La trampa clásica: string(65) NO produce "65"
    trampa := string(65)
    fmt.Printf("string(65) produce: '%s' (es la letra A en Unicode)\n", trampa)

    // Conversión entre tamaños de entero
    var grande int64 = 1000000
    pequeno := int32(grande)
    fmt.Printf("int64 %d como int32: %d\n", grande, pequeno)
}
```

Salida:

```
entero 42 como float64: 42.00
decimal 3.99 como int: 3 (se truncó, no se redondeó)
string(65) produce: 'A' (es la letra A en Unicode)
int64 1000000 como int32: 1000000
```

Las lecciones clave: Go te obliga a escribir cada conversión, lo que te hace pensar si es segura. De flotante a entero se trunca, no se redondea: si necesitas redondeo usa `math.Round`. Y para convertir números a su texto ("65" -> "65") no uses `string()`: usa `strconv.Itoa`, como verás en el siguiente ejemplo.

Ejemplo 5: strconv en un programa real

Un programa que pide la edad como texto, la convierte a número, calcula el año de nacimiento aproximado y muestra resultados. Incluye manejo del error de conversión.

```

package main

import (
    "fmt"
    "strconv"
)

func main() {
    var textoEdad string

    fmt.Print("¿Cuántos años tienes? ")
    fmt.Scan(&textoEdad)

    edad, err := strconv.Atoi(textoEdad)
    if err != nil {
        fmt.Println("Eso no parece un número válido. Intenta de nuevo.")
        return
    }

    // Suponemos el año actual 2026 para el cálculo
    const anioActual = 2026
    nacimiento := anioActual - edad

    fmt.Printf("Tienes %d años.\n", edad)
    fmt.Printf("Naciste alrededor del año %d.\n", nacimiento)
    fmt.Println("Tu edad como texto: " + strconv.Itoa(edad) + " años")

    // ParseFloat para decimales
    precioTexto := "19.99"
    precio, err := strconv.ParseFloat(precioTexto, 64)
    if err != nil {
        fmt.Println("Precio inválido")
        return
    }
    fmt.Printf("Precio con IVA (16%): %.2f\n", precio*1.16)
}

```

Si el usuario escribe 30, la salida será:

```

¿Cuántos años tienes? 30
Tienes 30 años.
Naciste alrededor del año 1996.
Tu edad como texto: 30 años
Precio con IVA (16%): 23.19

```

Observa el patrón completo: leer como texto, convertir con `Atoi`, revisar el error con `if err != nil`, y solo entonces calcular. El `%%` dentro de `Printf` produce un signo de porcentaje literal. Y nota cómo `strconv.Itoa` convierte el entero de vuelta a texto para concatenarlo con `+`. Este es el flujo estándar de entrada de datos numéricos en Go.

Ejemplo 6: Ámbito y sombreado, el bug invisible

Este ejemplo demuestra el sombreado a propósito para que aprendas a reconocerlo. Léelo con atención antes de ejecutarlo y trata de predecir la salida.

```
package main

import "fmt"

var appNombre = "MiApp v1.0" // ámbito de paquete

func main() {
    contador := 10
    fmt.Println("Al inicio, contador =", contador)

    if contador > 5 {
        contador := 99 // SOMBRA: variable nueva, solo vive dentro del if
        fmt.Println("Dentro del if, contador =", contador)
    }

    fmt.Println("Después del if, contador =", contador) // sigue siendo 10
    mostrarApp()
}

func mostrarApp() {
    fmt.Println("App:", appNombre) // bien: es de paquete
    // fmt.Println(contador) // ERROR: contador no existe en esta función
}
```

Salida:

```
Al inicio, contador = 10
Dentro del if, contador = 99
Después del if, contador = 10
App: MiApp v1.0
```

La variable exterior nunca cambió porque dentro del `if` se creó una variable nueva con `:=`. Si la intención hubiera sido modificar la exterior, habría que usar `=` sin los dos puntos. Este es el

bug más común relacionado con `:=` : cuando un valor "no se actualiza" dentro de un `if` o un `for`, busca un `:=` que debería ser `=`. Descomenta la última línea para ver el error de ámbito: contador no existe fuera de `main`.

Ejemplo 7: byte vs rune con texto en español

El español usa ñes y acentos, perfectos para demostrar la diferencia entre bytes y caracteres. Este ejemplo cuenta de las dos formas y recorre el texto carácter por carácter.

```
package main

import "fmt"

func main() {
    texto := "niño"

    fmt.Printf("Texto: %s\n", texto)
    fmt.Printf("len() en bytes: %d\n", len(texto))
    fmt.Printf("len() en caracteres: %d\n", len([]rune(texto)))

    fmt.Println("Recorrido byte por byte (incorrecto para mostrar):")
    for i := 0; i < len(texto); i++ {
        fmt.Printf("  byte %d: %v\n", i, texto[i])
    }

    fmt.Println("Recorrido por caracteres (correcto):")
    for i, r := range texto {
        fmt.Printf("  posición %d: %c (rune %d)\n", i, r, r)
    }
}
```

Salida:

```
Texto: niño
len() en bytes: 5
len() en caracteres: 4
Recorrido byte por byte (incorrecto para mostrar):
  byte 0: 110
  byte 1: 105
  byte 2: 195
  byte 3: 177
  byte 4: 111
Recorrido por caracteres (correcto):
  posición 0: n (rune 110)
  posición 1: i (rune 105)
  posición 2: ñ (rune 241)
  posición 3: o (rune 111)
```

La ñ ocupa dos bytes (195 y 177), por eso len da 5 aunque hay 4 caracteres. El recorrido con range sobre el string itera por runes y muestra cada carácter correctamente con el verbo %c. Moraleja: para texto con acentos, eñes o emojis, piensa en runes, no en bytes. El verbo %c muestra el carácter y %d su código numérico.

Ejemplo 8: Nombres bien puestos en un programa completo

Un programa de registro de alumnos que aplica las convenciones de nombres: camelCase, booleanos como preguntas y mayúscula inicial con sentido. Compara mentalmente cómo se leería con nombres crípticos.

```

package main

import "fmt"

// Constantes exportadas (mayúscula): visibles desde otros paquetes
const MaxAlumnos = 100
const NombreAcademia = "Lucas EGA Academy"

func main() {
    // Nombres descriptivos en camelCase
    nombreCompleto := "Ana García"
    edadAlumno := 22
    promedioFinal := 9.2

    // Booleanos nombrados como preguntas
    estaInscrito := true
    tieneBeca := false
    aproboCurso := promedioFinal >= 6.0

    fmt.Printf("Alumno: %s, edad: %d\n", nombreCompleto, edadAlumno)
    fmt.Printf("Promedio: %.1f\n", promedioFinal)

    if estaInscrito {
        fmt.Println("Estado: inscrito en", NombreAcademia)
    }
    if tieneBeca {
        fmt.Println("Tiene beca del 50%")
    } else {
        fmt.Println("Sin beca por el momento")
    }
    if aproboCurso {
        fmt.Println("Resultado: aprobó el curso. ¡Felicidades!")
    }

    // Contador con nombre corto pero claro en contexto pequeño
    for i := 0; i < 3; i++ {
        fmt.Printf("Módulo %d completado\n", i+1)
    }
}

```

Salida:


```
Alumno: Ana García, edad: 22
Promedio: 9.2
Estado: inscrito en Lucas EGA Academy
Sin beca por el momento
Resultado: aprobó el curso. ¡Felicidades!
Módulo 1 completado
Módulo 2 completado
Módulo 3 completado
```

Lee el código en voz alta: "si está inscrito", "si tiene beca", "si aprobó curso". Suena natural porque los nombres están bien elegidos. Las constantes con mayúscula inicial (MaxAlumnos, NombreAcademia) están listas para exportarse si este código creciera a varios paquetes. Y la i del ciclo es aceptable porque vive solo tres líneas: el contexto la explica.

Ejemplo 9: Laboratorio de tipos y conversiones

Un programa de exploración que junta inferencia, %T, conversiones y constantes no tipadas en un solo lugar. Úsalo como referencia rápida cuando dudes de un tipo.

```

package main

import "fmt"

func main() {
    fmt.Println("=== Inferencia de tipos ===")
    a := 42
    b := 3.14
    c := "Go"
    d := true
    e := 'A' // comillas simples: es un rune, no un string
    fmt.Printf("a=%v %T | b=%v %T | c=%v %T | d=%v %T | e=%v %T\n",
        a, a, b, b, c, c, d, d, e, e)

    fmt.Println("=== Constantes no tipadas ===")
    const factor = 10
    var entero int = factor
    var decimal float64 = factor
    var largo int64 = factor * 100
    fmt.Printf("entero=%d(%T) decimal=%.1f(%T) largo=%d(%T)\n",
        entero, entero, decimal, decimal, largo, largo)

    fmt.Println("=== Conversiones explícitas ===")
    x := 7
    y := 2
    // Sin conversión, x/y sería división entera: 3
    divisionEntera := x / y
    divisionReal := float64(x) / float64(y)
    fmt.Printf("7/2 entero = %d | 7/2 real = %.2f\n", divisionEntera, divisionReal)

    fmt.Println("=== Truncamiento al convertir ===")
    pi := 3.99
    fmt.Printf("int(3.99) = %d (se pierde la parte decimal)\n", int(pi))
    fmt.Printf("int(-3.99) = %d (trunca hacia cero)\n", int(-3.99))
}

```

Salida:

```

=== Inferencia de tipos ===
a=42 int | b=3.14 float64 | c=Go string | d=true bool | e=65 int32
=== Constantes no tipadas ===
entero=10(int) decimal=10.0(float64) largo=1000(int64)
=== Conversiones explícitas ===
7/2 entero = 3 | 7/2 real = 3.50
=== Truncamiento al convertir ===
int(3.99) = 3 (se pierde la parte decimal)
int(-3.99) = -3 (trunca hacia cero)

```

Tres lecciones en un ejemplo: primero, las comillas simples 'A' crean un rune (alias de int32 con valor 65), no un string: es un detalle que sorprende a todos la primera vez. Segundo, la división entre enteros da entero ($7/2 = 3$, no 3.5): para obtener decimales debes convertir al menos un operando a float64 antes de dividir. Tercero, la conversión de flotante a entero trunca hacia cero, tanto para positivos como para negativos. Este ejemplo es un excelente "cheat sheet" ejecutable: guárdalo y consúltalo cuando dudes.

Práctica

Ejercicio 1: Declara variables usando las tres formas (var sin inicializar, var con tipo e inicialización, y :=) para guardar tu nombre, tu edad y si te gusta programar. Imprime cada variable con su valor y su tipo usando los verbos %v y %T.

Ejercicio 2: Define un bloque const con iota para los cuatro puntos cardinales (norte, sur, este, oeste) numerados desde 0. Luego escribe un programa que guarde una dirección en una variable, la compare con las constantes e imprima a dónde apunta. Agrega una constante con nombre para los grados de una brújula completa (360).

Ejercicio 3: Escribe un programa que pida al usuario un precio como texto (por ejemplo "150.50"), lo convierta a float64 con strconv.ParseFloat, calcule el precio con 16% de IVA y muestre el resultado con dos decimales. Si la conversión falla, muestra un mensaje de error amable en vez de romperse.

Ejercicio 4: Declara una variable a nivel de paquete llamada versionApp con el valor "2.0". En main, declara una variable local con el mismo nombre usando := con otro valor, imprímela, y luego llama a una función mostrarVersion que imprima la variable de paquete. Explica en un comentario qué fenómeno estás demostrando.

Ejercicio 5: Escribe un programa que demuestre los valores cero: declara variables de tipo int, float64, string y bool sin inicializarlas, imprímelas, y luego escribe una condición que verifique

si el string está vacío y otra que verifique si el booleano es falso. Agrega comentarios explicando por qué los valores cero hacen seguro este código.

Ejercicio 6 (reto): Escribe un programa que pida al usuario una palabra en español con eñes o acentos (por ejemplo "murciélago"), muestre cuántos bytes ocupa con `len()`, cuántos caracteres tiene convirtiendo a `[]rune`, y luego recorra la palabra con `range` imprimiendo cada carácter con su número de rune. Al final, muestra la palabra en mayúsculas usando `strings.ToUpper`. Maneja todo en funciones separadas: una para contar y otra para mostrar los caracteres.

Solución 1: La clave es usar cada sintaxis al menos una vez y mostrar valor y tipo. Recuerda que `%v` muestra el valor en formato genérico y `%T` muestra el tipo.

```
package main

import "fmt"

func main() {
    // Forma 1: var sin inicializar (toma el valor cero)
    var nombre string
    nombre = "Esteban"

    // Forma 2: var con tipo explícito e inicialización
    var edad int = 30

    // Forma 3: declaración corta con inferencia
    meGustaProgramar := true

    fmt.Printf("nombre: %v (%T)\n", nombre, nombre)
    fmt.Printf("edad: %v (%T)\n", edad, edad)
    fmt.Printf("meGustaProgramar: %v (%T)\n", meGustaProgramar, meGustaProgramar)
}
```

Explicación: `nombre` se declaró con `var` sin valor inicial y se asignó después en una línea separada, lo que demuestra que declarar y asignar pueden ir por separado. `edad` usa `var` con tipo explícito, útil cuando quieres dejar claro el tipo o necesitas uno específico. `meGustaProgramar` usa `:=` y Go infiere `bool`. Los verbos `%v` y `%T` juntos son tu herramienta de inspección: `%v` muestra qué hay en la cajita y `%T` muestra de qué tipo es la cajita.

Solución 2: El bloque `const` con `iota` numera automáticamente. Solo se escribe `iota` en la primera línea; las demás lo heredan.

```

package main

import "fmt"

const (
    norte = iota // 0
    sur           // 1
    este          // 2
    oeste         // 3
)

const gradosBrujula = 360

func main() {
    var direccion int
    fmt.Print("¿A dónde apuntas? (0=norte, 1=sur, 2=este, 3=oeste): ")
    fmt.Scan(&direccion)

    if direccion == norte {
        fmt.Println("Apuntas al norte")
    } else if direccion == sur {
        fmt.Println("Apuntas al sur")
    } else if direccion == este {
        fmt.Println("Apuntas al este")
    } else if direccion == oeste {
        fmt.Println("Apuntas al oeste")
    } else {
        fmt.Println("Dirección desconocida")
    }

    fmt.Printf("Una brújula completa tiene %d grados\n", gradosBrujula)
}

```

Explicación: `iota` asignó 0, 1, 2 y 3 automáticamente. Comparar `direccion == norte` es mucho más legible que `direccion == 0`: el nombre explica el significado. La constante `gradosBrujula` documenta el valor 360 con un nombre. Si algún día agregas "noreste" entre norte y sur, `iota` renumera todo solo y las comparaciones siguen funcionando porque usan nombres, no números literales.

Solución 3: El patrón es leer texto, convertir con `ParseFloat`, revisar el error y calcular. El 64 en `ParseFloat` indica precisión de `float64`.

```

package main

import (
    "fmt"
    "strconv"
)

func main() {
    var textoPrecio string

    fmt.Print("Escribe el precio (ej. 150.50): ")
    fmt.Scan(&textoPrecio)

    precio, err := strconv.ParseFloat(textoPrecio, 64)
    if err != nil {
        fmt.Println("Ese precio no es válido. Usa solo números y punto decimal.")
        return
    }

    const iva = 0.16
    total := precio * (1 + iva)

    fmt.Printf("Precio sin IVA: %.2f\n", precio)
    fmt.Printf("Precio con IVA: %.2f\n", total)
}

```

Explicación: ParseFloat devuelve el número y un error; si el usuario escribe "cien" o "12,50" (con coma), err no será nil y mostramos un mensaje amable en vez de calcular con basura. La constante iva evita el número mágico 0.16 suelto en la fórmula. El verbo %.2f muestra exactamente dos decimales, ideal para dinero. Prueba con "150.50" (debe dar 174.58), con "abc" (debe mostrar el error) y con "100" (debe dar 116.00).

Solución 4: Este ejercicio demuestra el sombreado (shadowing): la variable local oculta a la de paquete dentro de main, pero la función mostrarVersion ve la de paquete porque está fuera de main.

```

package main

import "fmt"

var versionApp = "2.0" // ámbito de paquete

func main() {
    // Sombreado: esta variable local oculta a la de paquete dentro de main
    versionApp := "3.0-beta"
    fmt.Println("En main, versionApp =", versionApp)
    mostrarVersion()
}

func mostrarVersion() {
    // Aquí no hay sombra: se ve la variable de paquete
    fmt.Println("En mostrarVersion, versionApp =", versionApp)
}

```

Salida:

```

En main, versionApp = 3.0-beta
En mostrarVersion, versionApp = 2.0

```

Explicación: dentro de main, versionApp se refiere a la variable local "3.0-beta" porque := creó una variable nueva que sombrea la de paquete. En mostrarVersion no existe esa sombra, así que se ve la variable de paquete "2.0". Este comportamiento es legal pero peligroso si no es intencional: si tu intención era modificar la variable de paquete desde main, debiste usar = en vez of :=. Cuando un valor "no se actualiza" como esperabas, el sombreado es el primer sospechoso.

Solución 5: Declarar sin inicializar y operar con confianza, gracias a los valores cero. Las condiciones verifican los valores iniciales de forma explícita.

```

package main

import "fmt"

func main() {
    // Declaradas sin inicializar: Go les da el valor cero automáticamente
    var cantidad int
    var precio float64
    var cliente string
    var pagado bool

    fmt.Printf("cantidad=%d precio=%.2f cliente='%s' pagado=%t\n",
        cantidad, precio, cliente, pagado)

    // Los valores cero son predecibles, así que estas condiciones son seguras
    if cliente == "" {
        // Un string recién declarado siempre es "", nunca basura aleatoria
        fmt.Println("Aún no hay cliente registrado (valor cero de string)")
    }

    if !pagado {
        // Un bool recién declarado siempre es false
        fmt.Println("El pedido aún no está pagado (valor cero de bool)")
    }

    // Operar con valores cero es seguro desde el primer momento
    cantidad = cantidad + 5
    precio = precio + 99.99
    fmt.Printf("Después de operar: cantidad=%d precio=%.2f\n", cantidad, precio)
}

```

Explicación: en otros lenguajes, leer cantidad antes de asignarla podría dar un valor aleatorio; en Go siempre da 0. Por eso incrementar un contador recién declarado funciona a la primera, y por eso comparar cliente == "" es una forma válida de detectar "aún no asignado". Los comentarios explican el porqué, no el qué: documentan la garantía del lenguaje que hace seguro este código.

Solución 6: El reto combina runes, funciones separadas y el paquete strings. Cada función tiene una responsabilidad clara.


```

package main

import (
    "fmt"
    "strings"
)

// contarElementos muestra la diferencia entre bytes y caracteres.
func contarElementos(palabra string) {
    bytes := len(palabra)
    caracteres := len([]rune(palabra))
    fmt.Printf("La palabra '%s' ocupa %d bytes y tiene %d caracteres\n",
        palabra, bytes, caracteres)
}

// mostrarCaracteres recorre la palabra rune por rune.
func mostrarCaracteres(palabra string) {
    fmt.Println("Caracteres uno por uno:")
    for _, r := range palabra {
        fmt.Printf("  %c (código %d)\n", r, r)
    }
}

func main() {
    var palabra string

    fmt.Print("Escribe una palabra con eñes o acentos: ")
    fmt.Scan(&palabra)

    contarElementos(palabra)
    mostrarCaracteres(palabra)
    fmt.Printf("En mayúsculas: %s\n", strings.ToUpper(palabra))
}

```

Si el usuario escribe "murciélago", la salida será:

```
Escribe una palabra con eñes o acentos: murciélago
La palabra 'murciélago' ocupa 11 bytes y tiene 10 caracteres
Caracteres uno por uno:
  m (código 109)
  u (código 117)
  r (código 114)
  c (código 99)
  i (código 105)
  é (código 233)
  l (código 108)
  a (código 97)
  g (código 103)
  o (código 111)
En mayúsculas: MURCIÉLAGO
```

Explicación: la é ocupa 2 bytes, por eso hay 11 bytes y 10 caracteres. El range sobre el string itera por runes, así que cada carácter se muestra completo con %c y su código con %d. Dividir el programa en contarElementos y mostrarCaracteres aplica una buena práctica fundamental: cada función hace una sola cosa y su nombre dice cuál. strings.ToUpper maneja correctamente los acentos al convertir a mayúsculas. Nota que fmt.Scan lee hasta el primer espacio, así que pide una sola palabra.

Resumen de la unidad

En esta unidad le diste memoria a tus programas. Aprendiste que una variable es un espacio de memoria con nombre y tipo, y que en Go tienes tres operaciones básicas: declarar con var, asignar con = e inicializar combinando ambas. Dominaste las dos sintaxis de declaración: var, con tipo explícito y permitida en todos los ámbitos, y :=, la declaración corta con inferencia de tipos que solo vive dentro de funciones. Memorizaste sus reglas: := exige al menos una variable nueva a la izquierda, no funciona a nivel de paquete, y usarla donde debías usar = crea el famoso sombreado que oculta variables sin que te des cuenta.

Descubriste el valor cero, una de las garantías más elegantes de Go: toda variable nace con un valor predecible (0, "", false, nil) en vez de basura aleatoria, lo que elimina una categoría entera de bugs no deterministas. Conociste las constantes con const, incluyendo las constantes no tipadas y los bloques const para agrupar valores relacionados, y aprendiste a evitar los números mágicos dándoles nombre. Y dominaste iota, el contador automático que numera secuencias dentro de bloques const, con sus trucos: expresiones como 1 << iota para potencias de dos, el identificador vacío _ para saltar valores, y la regla de que se reinicia en cada bloque.

Recorriste los tipos básicos: `int` como el entero de uso general, la familia `int8/int16/int32/int64` para cuando el tamaño importa, `uint` para valores no negativos (con cuidado del underflow), `float64` para decimales (recordando que son aproximaciones), `bool` para lógica, `string` para texto inmutable en UTF-8, y la dupla `byte/rune` para entender que `len` cuenta bytes mientras que los caracteres se cuentan con `len` `rune`. Viste cómo Go infiere tipos a partir de literales y cómo verificar la inferencia con el verbo `%T`.

Aprendiste la regla de oro de las conversiones: en Go nada se convierte a escondidas; toda conversión es explícita con la sintaxis `T(v)`, y las conversiones riesgosas (como `float` a `int`, que trunca) quedan visibles en el código. Dominaste `strconv` para el puente entre texto y números: `Atoi`, `Itoa`, `ParseFloat` y `FormatFloat`, siempre revisando el error con el patrón `if err != nil`. Entendiste el ámbito: las llaves definen dónde vive cada variable, con niveles de paquete, función y bloque. Y te enfrentaste al error más famoso de Go, "declared and not used", entendiendo que no es tiranía sino higiene automática, con el identificador vacío `_` como válvula de escape oficial.

Finalmente, adoptaste las convenciones de nombres que hacen que tu código se sienta nativo: `camelCase`, mayúscula inicial para lo exportado y minúscula para lo privado, booleanos nombrados como preguntas, y nombres cuya longitud crece con el alcance de la variable.

En la siguiente unidad entrarás al terreno donde los programas empiezan a tomar decisiones: las estructuras de control. Con las variables y tipos que dominas ahora, aprenderás a bifurcar el camino con `if` y `switch`, y a repetir tareas con los ciclos `for`. Todo lo que hagas de aquí en adelante (validar datos, buscar en listas, repetir cálculos) se construye sobre los cimientos de esta unidad. Nos vemos en la unidad 3.

UNIDAD 3: Operadores y control de flujo

Si las variables son los sustantivos de un programa y los tipos de datos son los adjetivos que los describen, entonces los operadores son los verbos y el control de flujo es la gramática que decide que se dice y en que orden. En la unidad anterior aprendiste a guardar información: números enteros, decimales, textos, valores lógicos. Eso está muy bien, pero un programa que solo guarda datos es como una libreta: útil, pero pasiva. Lo que convierte a esos datos en un programa de verdad es la capacidad de transformarlos, compararlos y tomar decisiones con ellos. Eso es exactamente lo que vas a dominar en esta unidad.

Piénsalo así: una calculadora no es interesante porque muestra números en pantalla, sino porque los suma, los resta, los multiplica y decide que hacer con el resultado. Un videojuego no es divertido porque guarda la posición del personaje, sino porque calcula hacia donde se mueve, detecta si chocó con una pared y decide si el jugador gana o pierde. Una tienda en línea no sirve de mucho si solo almacena precios: necesita calcular totales con impuestos, comparar el saldo del cliente con el costo del carrito y decidir si aprueba o rechaza la compra. Operadores y control de flujo son las dos herramientas que hacen posible todo eso, y por eso esta unidad es el verdadero punto de partida de la programación seria.

Los operadores son símbolos especiales que le dicen a Go que haga una operación con uno o dos valores. Cuando escribes $5 + 3$, el signo de más es un operador aritmético que toma dos operandos, el 5 y el 3, y produce un resultado, el 8. Parece trivial, pero hay matices que sorprenden a muchos principiantes y que en Go están diseñados con una filosofía muy clara: el lenguaje prefiere ser explícito y predecible antes que mágico. Por ejemplo, en Go no puedes mezclar enteros y decimales en una operación sin convertirlos primero de forma explícita. Otros lenguajes lo hacen por ti en silencio y a veces el resultado no es el que esperabas. Go te obliga a decir exactamente lo que quieres, y aunque al principio parezca más trabajo, a la larga te ahorra una cantidad enorme de errores difíciles de encontrar. En esta unidad vas a ver cada familia de operadores: los aritméticos, los de asignación, los de comparación y los lógicos, además de las reglas de precedencia que deciden que operación se hace primero cuando hay varias en la misma expresión.

El control de flujo es la otra mitad de la unidad y es, si se quiere, todavía más importante. Un programa se ejecuta línea por línea, de arriba hacia abajo, pero los programas interesantes casi nunca siguen un camino recto: toman decisiones y repiten tareas. El control de flujo es el conjunto de estructuras que te permiten romper esa secuencia lineal. Con la sentencia `if` puedes decirle al programa: si el usuario es mayor de edad, muéstralo este contenido; si no, muestra otro. Con `switch` puedes elegir entre muchas opciones de forma elegante, como un menú de restaurante donde cada plato lleva a una preparación distinta. Con `for`, que en Go es

el unico bucle que existe, puedes repetir una tarea diez veces, cien veces o un millon de veces sin escribir el codigo un millon de veces. Y con `break` y `continue` puedes afinar ese comportamiento: salir de un bucle antes de tiempo o saltarte una vuelta que no te interesa.

Go toma decisiones de diseno muy deliberadas en este terreno y vale la pena entenderlas porque revelan la personalidad del lenguaje. A diferencia de otros lenguajes que te dan tres o cuatro tipos de bucle (`for`, `while`, `do-while`, `foreach`), Go te da uno solo: `for`. Y resulta que ese unico `for` puede hacer todo lo que hacen los demas, con una sintaxis mas simple y menos cosas que memorizar. A diferencia de otros lenguajes donde el `switch` se comporta de forma traicionera si olvidas escribir `break`, en Go cada caso del `switch` termina automaticamente: no hay caidas accidentales de un caso a otro. Y a diferencia de lenguajes donde puedes usar el incremento `i++` dentro de cualquier expresion y terminar con codigo casi ilegible, en Go el incremento es una sentencia sencilla que hace una sola cosa. Cada una de estas decisiones tiene el mismo objetivo: que tu codigo sea facil de leer, facil de predecir y dificil de romper por accidente.

Esta unidad es larga porque el tema lo merece: es la base sobre la que se construye todo lo demas. Las funciones que veras en la unidad 4 reciben parametros y devuelven resultados usando operadores y decisiones. Los arreglos y mapas de unidades posteriores se recorren con `for`. El manejo de errores depende de `if`. Practicamente no hay programa en Go que no use lo que vas a aprender aqui. Asi que tomate tu tiempo, prueba cada ejemplo en tu computadora y no avances hasta sentirte comodo. La mejor forma de estudiar esta unidad es tener una terminal abierta al lado: cada vez que veas un bloque de codigo, copialo a un archivo, ejecutalo con `go run` y luego modificalo para romperlo a proposito. Cambia un operador, quita un parentesis, invierte una condicion y observa que pasa. Esa experimentacion es la que convierte la lectura en conocimiento real.

Antes de entrar en materia, un consejo practico sobre como probar los ejemplos. Todos los programas de esta unidad siguen la misma estructura minima: pertenecen al paquete `main`, importan el paquete `fmt` para imprimir en pantalla y tienen una funcion `main` que es el punto de entrada. Para probar cualquiera de ellos, crea un archivo con extension `.go`, por ejemplo `prueba.go`, pega el codigo y ejecuta en tu terminal: `go run prueba.go`. Si el codigo tiene un error, el compilador de Go te lo dira con un mensaje bastante claro que incluye el numero de linea. Acostumbrate desde ahora a leer esos mensajes: son tu mejor maestro. Y recuerda que Go es estricto con cosas como las llaves y el formato; si algo no compila, revisa primero los detalles pequenos antes de sospechar de la logica.

Explicación detallada

3.1 Operadores aritméticos: la calculadora del lenguaje

Los operadores aritmeticos son los que ya conoces de la escuela: la suma (+), la resta (-), la multiplicacion (*) y la division (/). Go agrega el operador modulo (%), que calcula el residuo de una division entera, y también tiene operadores para cambiar el signo de un numero, como el menos unario. Con estos cinco operadores puedes construir cualquier calculo numerico, desde el total de una cuenta de restaurante hasta la fisica de un videojuego. Vamos a verlos uno por uno, con atencion especial a los detalles donde Go se comporta distinto a lo que muchos esperan.

La suma, la resta y la multiplicacion funcionan exactamente como imaginas, con una regla de oro que debes grabarte desde hoy: en Go, los dos operandos de una operacion aritmetica deben ser del mismo tipo. No puedes sumar un int con un float64 directamente. Si lo intentas, el compilador te detendra con un error de tipos no coincidentes. Esto es intencional. En otros lenguajes, mezclar tipos provoca conversiones automaticas silenciosas que a veces redondean o pierden precision sin avisarte. Go prefiere que la conversion sea explicita, para que siempre sepas que esta pasando. Asi que si tienes una variable entera edad y una decimal precio, y quieres operar con ambas, conviertes primero: `float64(edad) + precio`. Esa conversion es una promesa que le haces al compilador: se lo que estoy haciendo y acepto el resultado.

La division es donde aparecen las primeras sorpresas interesantes, asi que prestale atencion extra. En Go, el resultado de dividir dos enteros es siempre un entero, y cualquier parte decimal simplemente se descarta, no se redondea. La expresion `7 / 2` vale 3, no 3.5. La expresion `10 / 3` vale 3, no 3.33. Esto se llama division entera o truncamiento, y es una fuente clasica de errores para principiantes. Imagina que quieres calcular el promedio de tres calificaciones: `(9 + 8 + 7) / 3`. Si las calificaciones son enteros, el resultado sera 8, porque `24 / 3` es exactamente 8 y no hay problema. Pero si las calificaciones son 9, 8 y 8, la suma es 25 y `25 / 3` en division entera da 8, cuando el promedio real es 8.33. El programa no te avisa: simplemente te da un numero equivocado desde tu punto de vista. La solucion es asegurarte de que al menos uno de los operandos sea decimal: `float64(suma) / 3` te dara `8.333333333333334`. Recuerda esta regla como un mantra: si quieres decimales en el resultado, necesita haber decimales en la operacion.

El operador modulo (%) calcula el residuo de una division entera y solo funciona con numeros enteros. Por ejemplo, $7 \% 3$ vale 1, porque 7 entre 3 cabe 2 veces (2 por 3 es 6) y sobra 1. El modulo tiene usos muy practicos que veras una y otra vez: saber si un numero es par o impar ($\text{numero} \% 2 == 0$ significa par), repartir turnos en forma circular, o limitar un contador para que se reinicie. Un ejemplo bonito: si tienes un reloj que cuenta horas de 0 a 23 y quieres sumarle horas sin pasarte, puedes usar $(\text{horaActual} + \text{horasASumar}) \% 24$. Si son las 22 y sumas 5, obtienes $27 \% 24 = 3$, o sea las 3 de la madrugada. Elegante y sin condicionales.

Hay dos errores clasicos con la aritmetica que debes conocer desde ahora. El primero es la division entre cero con enteros: provoca un panico en tiempo de ejecucion y tu programa se detiene de golpe. Con decimales, en cambio, Go sigue las reglas del estandar IEEE 754 y te da valores especiales como infinito positivo, lo cual rara vez es lo que quieres pero al menos no rompe el programa. El segundo error es el desbordamiento de enteros: cada tipo entero tiene un rango maximo, y si lo excedes el valor se envuelve y empieza desde el otro extremo, como un odometro de automovil que al pasar del maximo vuelve a cero. Go no te avisa del desbordamiento en tiempo de ejecucion, asi que si trabajas con numeros muy grandes elige tipos amplios como `int64` o `uint64` desde el principio.

```

package main

import "fmt"

func main() {
    // Division entera: el decimal se descarta, no se redondea
    fmt.Println(7 / 2)    // 3
    fmt.Println(10 / 3)   // 3
    fmt.Println(-7 / 2)   // -3 (trunca hacia cero)

    // Division flotante: al menos un operando debe ser decimal
    fmt.Println(7.0 / 2) // 3.5
    suma := 25
    fmt.Println(float64(suma) / 3) // 8.333333333333334

    // Modulo: solo con enteros
    fmt.Println(7 % 3)    // 1
    fmt.Println(10 % 5)   // 0

    // El modulo sirve para detectar pares e impares
    numero := 42
    if numero%2 == 0 {
        fmt.Println("El numero es par")
    }

    // Reloj circular con modulo
    hora := (22 + 5) % 24
    fmt.Println("Hora resultante:", hora) // 3
}

```

Como buena practica, cuando hagas calculos donde el resultado decimal importa (promedios, porcentajes, precios con impuestos), convierte a float64 al inicio de la operacion y trabaja todo en decimales. Y cuando necesites contar cosas discretas (personas, vueltas de un bucle, elementos de una lista), usa enteros. Esa separacion mental entre contar y medir te evitara la mayoria de los dolores de cabeza con la aritmetica.

3.2 Operadores de asignación: guardar y actualizar

El operador de asignacion mas basico es el signo de igual =, que guarda un valor en una variable. Pero no lo confundas con el igual de las matematicas: en programacion, $x = x + 1$ no es una ecuacion imposible, es una instruccion que significa toma el valor actual de x, sumale 1 y guarda el resultado de nuevo en x. Se lee de derecha a izquierda: primero se evalua la

expresion de la derecha y después se almacena en la variable de la izquierda. Go además tiene la declaracion corta con `:=` que ya conoces, la cual declara la variable y le asigna un valor al mismo tiempo, infiriendo el tipo automaticamente. Usa `:=` cuando crees una variable nueva y `=` cuando actualices una que ya existe: esa distincion mantiene tu codigo claro.

Sobre esa base, Go ofrece los operadores de asignacion compuesta, que combinan una operacion aritmetica con la asignacion en un solo paso. Son cinco: `+=`, `-=`, `*=`, `/=` y `%=`. La expresion `total += precio` es una forma corta y elegante de escribir `total = total + precio`. Estos operadores no son solo un ahorro de tecleo: comunican intencion. Cuando alguien lee `saldo -= retiro`, entiende de inmediato que el saldo se esta reduciendo, sin tener que analizar una expresion mas larga. Usalos siempre que actualices una variable con base en su propio valor anterior, que es una de las operaciones mas comunes en programacion: acumuladores que suman, contadores que crecen, saldos que disminuyen.

```
package main

import "fmt"

func main() {
    total := 0
    total += 150 // total = total + 150
    total += 200
    fmt.Println("Total:", total) // 350

    saldo := 1000
    saldo -= 250 // total = total - 250
    fmt.Println("Saldo:", saldo) // 750

    precio := 100.0
    precio *= 1.16 // agregar 16% de IVA
    fmt.Println("Precio con IVA:", precio) // 116

    // El operador %= también existe, util para contadores circulares
    turno := 7
    turno %= 5 // turno = turno % 5
    fmt.Println("Turno:", turno) // 2
}
```

Y llegamos a dos operadores diminutos pero famosos: `++` y `--`, que incrementan o decrementan una variable en uno. Escribir `contador++` es lo mismo que `contador += 1`, que a su vez es lo mismo que `contador = contador + 1`. Son la forma mas corta y la mas usada en bucles, donde veras `for i := 0; i < 10; i++` hasta en sueños. Pero atencion, porque aqui Go se

diferencia de lenguajes como C, Java o JavaScript de una forma que al principio desconcierta: en Go, `++` y `--` son sentencias, no expresiones. Eso significa que hacen su trabajo y punto: no devuelven ningun valor y no puedes usarlos dentro de otra expresion. La linea `j := i++` es un error de compilacion en Go, mientras que en otros lenguajes asignaria el valor viejo de `i` a `j`. Del mismo modo, no existe el pre-incremento `++i`: solo el post-incremento `i++`. Esta restriccion es deliberada y es una bendicion disfrazada: elimina toda una categoria de codigo confuso donde nadie esta seguro de que valor se uso primero. En Go, el incremento hace exactamente una cosa y la hace de forma obvia.

Un error comun es intentar usar `++` o `--` con valores que no son variables, como escribir `(a + b)++` o usarlo sobre el resultado de una funcion. El incremento necesita una direccion de memoria donde guardar el nuevo valor, asi que solo funciona con variables. Otro detalle: aunque veas `i++` por todos lados en los bucles, dentro del cuerpo de un programa normal es perfectamente valido usar la forma larga cuando comunica mejor la intencion. La legibilidad manda: elige la forma que haga mas obvio lo que esta pasando.

3.3 Operadores de comparación: hacer preguntas al programa

Si los operadores aritmeticos calculan, los operadores de comparacion preguntan. Cada comparacion toma dos valores del mismo tipo y devuelve un booleano: `true` o `false`. Son seis: `==` (igual a), `!=` (diferente de), `<` (menor que), `>` (mayor que), `<=` (menor o igual que) y `>=` (mayor o igual que). Fijate bien en el doble signo de igual `==`: un solo `=` asigna, dos `==` comparan, y confundirlos es uno de los errores mas clasicos de la programacion. Por suerte, Go te protege parcialmente: si escribes `if x = 5` por accidente, el compilador se quejara porque una asignacion no es una condicion valida. Aun asi, entrena tu ojo para distinguirlos al leer codigo.

Las comparaciones numericas funcionan como en matematicas: `5 > 3` es `true`, `10 <= 9` es `false`. También puedes comparar cadenas de texto con `==` y `!=`, lo cual compara el contenido exacto, caracter por caracter, distinguiendo mayusculas de minusculas: `"Hola" == "hola"` es `false`. Y puedes ordenar cadenas con `<` y `>`, que las compara en orden lexicografico según los valores de sus bytes, util para ordenar nombres alfabeticamente. Los booleanos se comparan con `==` y `!=`. Cada tipo se compara con valores de su mismo tipo: no puedes comparar un `int` con un `string`, el compilador no lo permite, lo cual evita comparaciones sin sentido.

Hay una trampa famosa que debes conocer: nunca compares numeros decimales con `==` cuando provienen de calculos. Los flotantes se representan en binario con precision finita, asi que operaciones inocentes como `0.1 + 0.2` no dan exactamente `0.3` sino `0.30000000000000004`. Si tu programa pregunta si el resultado `== 0.3`, la respuesta sera `false` aunque matematicamente deberia ser `true`. La solucion profesional es comparar con una

tolerancia o epsilon: en lugar de preguntar si $a == b$, preguntas si la diferencia absoluta entre a y b es menor que un valor muy pequeño, como 0.000001 . Así: $\text{math.Abs}(a - b) < 1e-6$. Esta técnica aparece en videojuegos, simulaciones físicas y cálculos financieros, y conocerla te distingue como alguien que entiende como funcionan realmente las computadoras.

```
package main

import (
    "fmt"
    "math"
)

func main() {
    edad := 20
    fmt.Println(edad >= 18) // true: es mayor de edad
    fmt.Println(edad == 21) // false
    fmt.Println(edad != 20) // false

    nombre1 := "Ana"
    nombre2 := "ana"
    fmt.Println(nombre1 == nombre2) // false: distingue mayúsculas
    fmt.Println(nombre1 < "Beto")   // true: orden alfabético

    // La trampa de los flotantes
    a := 0.1 + 0.2
    fmt.Println(a == 0.3) // false, aunque "debería" ser true
    fmt.Println(a)        // 0.30000000000000004

    // La solución: comparar con tolerancia
    tolerancia := 1e-6
    iguales := math.Abs(a-0.3) < tolerancia
    fmt.Println("Son iguales (con tolerancia):", iguales) // true
}
```

Como buena práctica, usa las comparaciones para expresar reglas de negocio de forma legible: $\text{edad} \geq 18$ dice es mayor de edad mucho mejor que un número mágico suelto. Y cuando compares flotantes resultado de cálculos, usa siempre la técnica de la tolerancia. Tu yo del futuro, depurando un programa financiero a medianoche, te lo agradecerá.

3.4 Operadores lógicos y el cortocircuito: combinar preguntas

Los operadores lógicos te permiten combinar varias comparaciones en una sola condición más expresiva. Son tres: `&&` (Y lógico), `||` (O lógico) y `!` (NO lógico, negación). El operador `&&` devuelve `true` solo si ambas condiciones son verdaderas: tienes INE `&&` tienes más de 18 años significa que cumples ambos requisitos. El operador `||` devuelve `true` si al menos una es verdadera: es fin de semana `||` es día festivo significa que descansas si se cumple cualquiera de las dos. El operador `!` invierte un valor lógico: `!lloviendo` significa no está lloviendo. Con estos tres puedes expresar cualquier regla de decisión, desde las más simples hasta las más enredadas.

Para dominarlos necesitas las tablas de verdad, que no son más que el resumen de todos los casos posibles. Para `&&`: `true && true` es `true`, cualquier otra combinación es `false`. Para `||`: `false || false` es `false`, cualquier otra combinación es `true`. Para `!`: `!true` es `false` y `!false` es `true`. Memorízalas con ejemplos de la vida real: para entrar a un antro necesitas ser mayor de edad Y traer identificación (`&&`: las dos); para pasar el semestre necesitas aprobar el examen O entregar el proyecto extra (`||`: con una basta); si NO tienes saldo (`!`), no puedes comprar. Cuando las condiciones se vuelven largas, usa parentesis para agrupar y que la intención quede clara: `(esSocio || tieneCupon) && tiendaAbierta` se lee mucho mejor que una cadena plana de operadores.

Ahora, el concepto estrella de esta sección: el cortocircuito. Go evalúa `&&` y `||` de izquierda a derecha y se detiene en cuanto ya sabe el resultado final, sin evaluar el resto. En una expresión con `&&`, si la primera condición es `false`, el resultado total ya será `false` sin importar lo demás, así que Go ni siquiera mira la segunda condición. En una con `||`, si la primera es `true`, el resultado ya será `true` y se salta la segunda. Esto no es solo una optimización: es una herramienta de seguridad que los programadores Go usan todos los días. El patrón más famoso es la guarda contra valores peligrosos: `if usuario != nil && usuario.Edad >= 18`. Si `usuario` es `nil`, la primera condición es `false`, Go se detiene ahí y nunca intenta leer `usuario.Edad`, lo que habría provocado un pánico. Sin cortocircuito tendrías que anidar dos `if`; con cortocircuito lo resuelves en una línea elegante y segura.

```

package main

import "fmt"

func main() {
    edad := 22
    tieneINE := true

    // Y logico: ambas deben cumplirse
    puedeEntrar := edad >= 18 && tieneINE
    fmt.Println("Puede entrar:", puedeEntrar) // true

    // O logico: basta con una
    esFinDeSemana := false
    esFestivo := true
    descansa := esFinDeSemana || esFestivo
    fmt.Println("Descansa hoy:", descansa) // true

    // Negacion
    lloviendo := false
    fmt.Println("Puedo salir sin paraguas:", !lloviendo) // true

    // Cortocircuito como guarda de seguridad:
    // si divisor es cero, la segunda parte ni se evalua y no hay panico
    divisor := 0
    if divisor != 0 && 100/divisor > 10 {
        fmt.Println("El cociente es grande")
    } else {
        fmt.Println("Division no realizada (divisor cero, sin panico)")
    }

    // Agrupar con parentesis para expresar reglas complejas
    esSocio := false
    tieneCupon := true
    tiendaAbierta := true
    puedeComprar := (esSocio || tieneCupon) && tiendaAbierta
    fmt.Println("Puede comprar:", puedeComprar) // true
}

```

Un error comun es abusar de la negacion hasta volver el codigo un acertijo: `!(a && !b) || c` es logicamente correcto pero humanamente ilegible. Cuando una condicion se vuelve asi de enredada, dividela en variables booleanas con nombres descriptivos o extraela a una funcion. Otra buena practica: ordena las condiciones poniendo primero las mas baratas de evaluar y las mas probables de decidir el resultado, para aprovechar el cortocircuito. Por ejemplo, en

usuario != nil && usuario.EsAdmin(), la comprobacion de nil es casi gratis y protege la segunda, que podria ser costosa. El cortocircuito trabaja para ti solo si lo pones en el orden correcto.

3.5 Precedencia de operadores: quién se evalúa primero

Cuando una expresion tiene varios operadores, Go necesita reglas para decidir el orden de evaluacion, igual que en matematicas donde la multiplicacion se hace antes que la suma. Esas reglas se llaman precedencia, y aunque la tabla completa tiene varios niveles, con memorizar lo esencial y usar parentesis generosamente tienes mas que suficiente. El orden, de mayor a menor precedencia, para los operadores que ya conoces es este: primero los unarios como ! y el menos de signo (-x) y las llamadas; después los multiplicativos *, / y %; después los aditivos + y -; después las comparaciones <, >, <=, >=; después == y !=; después && y al final ||. Los operadores de asignacion (=, += y familia) tienen la precedencia mas baja de todas, lo cual tiene sentido: primero se calcula todo y al final se guarda.

Veamos ejemplos concretos para que quede grabado. La expresion `2 + 3 * 4` vale 14, no 20, porque la multiplicacion va primero. La expresion `!!lueve && tengoParaguas` se evalua como `(!!lueve) && tengoParaguas`, porque ! tiene mas precedencia que &&. La expresion `edad >= 18 && tieneINE` funciona como `(edad >= 18) && tieneINE`, porque las comparaciones van antes que &&. Y la clasica trampa: `a == b && c == d` significa `(a == b) && (c == d)`, que es lo que normalmente quieres. Pero ojo con esta otra: `x & 1 == 1` no significa lo que parece si mezclas operadores de bits, aunque esos los veremos en otra unidad. La moraleja es simple y te la repito porque es oro: cuando tengas la mas minima duda, pon parentesis. Los parentesis siempre ganan a la precedencia, cuestan un par de caracteres y eliminan cualquier ambigüedad para ti y para quien lea tu codigo.

```

package main

import "fmt"

func main() {
    // La multiplicacion va antes que la suma
    fmt.Println(2+3*4)      // 14
    fmt.Println((2 + 3) * 4) // 20: los parentesis mandan

    // ! va antes que &&
    llueve := false
    tengoParaguas := true
    fmt.Println(!llueve && tengoParaguas) // true

    // Las comparaciones van antes que && y ||
    edad := 20
    tieneINE := true
    esVIP := false
    // Se lee: (edad >= 18 && tieneINE) || esVIP
    fmt.Println(edad >= 18 && tieneINE || esVIP) // true

    // Con parentesis explicitos, la intencion es inconfundible
    fmt.Println((edad >= 18 && tieneINE) || esVIP) // true
    fmt.Println(edad >= 18 && (tieneINE || esVIP)) // true (aqui da igual, pero se
    entiende mejor)
}

```

El error clasico de precedencia es escribir una condicion larga sin parentesis, que funcione por casualidad con los datos de prueba y falle con datos reales. Por ejemplo, `esAdmin || esDueno && cuentaActiva` se evalua como `esAdmin || (esDueno && cuentaActiva)`, porque `&&` va antes que `||`. Si lo que querias era `(esAdmin || esDueno) && cuentaActiva`, tienes un bug de seguridad: un administrador con la cuenta desactivada pasaria la validacion. Este tipo de errores son silenciosos y peligrosos, y la vacuna es barata: parentesis explicitos en cualquier condicion que mezcle `&&` con `||`. Adoptalo como habito y dormiras tranquilo.

3.6 La sentencia if: tomar decisiones

La sentencia `if` es la estructura de decision mas basica y la que mas vas a usar en tu vida como programador. Su logica es simple: evalua una condicion, que es cualquier expresion que produzca un booleano, y si el resultado es `true` ejecuta el bloque de codigo entre llaves; si es `false`, lo salta. La sintaxis es `if condicion { ... }`, sin parentesis alrededor de la condicion, a diferencia de otros lenguajes. Las llaves son obligatorias incluso si el bloque tiene una sola

línea: Go no permite el if sin llaves. Esta rigidez es intencional y evita una familia entera de bugs donde alguien agrega una segunda línea pensando que está dentro del if cuando en realidad está fuera.

Cuando necesitas una alternativa, agregas else: si la condición es true se ejecuta el primer bloque, si es false se ejecuta el segundo. Y cuando hay más de dos caminos, encadenas con else if, que evalúa sus condiciones en orden y ejecuta el primer bloque cuya condición sea verdadera. Puedes encadenar tantos else if como quieras y terminar con un else final que atrape todo lo demás. Un detalle importante de estilo en Go: se prefiere el retorno temprano sobre la anidación profunda. En lugar de meter todo el código dentro de un if gigante, compruebas la condición de error primero y sales: `if err != nil { return }`. Así el camino principal del programa queda al nivel más externo, fácil de leer de arriba hacia abajo, sin pirámides de llaves anidadas. Este patrón se llama guard clause y lo verás en todo el código Go profesional.

La joya de esta sección es la inicialización corta dentro del if, una característica elegante que Go te regala: puedes declarar una variable justo antes de la condición, separada por punto y coma: `if x := obtenerValor(); x > 0 { ... }`. Esa variable existe solo dentro del if y sus ramas else, y desaparece al terminar la sentencia. Esto es perfecto para el patrón más común de todos: llamar a una función que puede fallar y revisar el error de inmediato: `if err := guardar(); err != nil { ... }`. La variable `err` nace, se usa y muere en ese if, sin contaminar el resto de la función con variables temporales. También es útil con funciones que devuelven valor y error juntos: `if resultado, err := dividir(a, b); err != nil { ... } else { ... }`. Fíjate que en la rama else también puedes usar las variables declaradas, lo cual es muy conveniente.

```
package main

import (
    "errors"
    "fmt"
)

func dividir(a, b float64) (float64, error) {
    if b == 0 {
        return 0, errors.New("division entre cero")
    }
    return a / b, nil
}

func main() {
    edad := 17

    // if simple
```



```

if edad >= 18 {
    fmt.Println("Acceso permitido")
}

// if-else
if edad >= 18 {
    fmt.Println("Puedes votar")
} else {
    fmt.Println("Aun no puedes votar")
}

// else-if encadenado: calificacion con letra
puntos := 85
if puntos >= 90 {
    fmt.Println("Calificacion: A")
} else if puntos >= 80 {
    fmt.Println("Calificacion: B")
} else if puntos >= 70 {
    fmt.Println("Calificacion: C")
} else {
    fmt.Println("Calificacion: reprobado")
}

// Inicializacion corta: la variable vive solo dentro del if/else
if resultado, err := dividir(100, 4); err != nil {
    fmt.Println("Error:", err)
} else {
    fmt.Println("100 / 4 =", resultado)
}
// fmt.Println(resultado) // ERROR: resultado no existe aqui

// Guard clause: retorno temprano en vez de anidacion
numero := -5
if numero < 0 {
    fmt.Println("Numero invalido, saliendo")
    return
}
fmt.Println("Procesando numero:", numero)
}

```

Los errores comunes con if son pocos pero persistentes. El primero es olvidar que la variable de la inicializacion corta no existe fuera del if: si necesitas el valor después, declara la variable antes. El segundo es anidar demasiados if hasta crear la famosa piramide de la perdicion, codigo tan indentado que es imposible de seguir: la cura son las guard clauses con retorno temprano. Y el tercero es escribir condiciones redundantes como `if esValido == true`: el `== true`

sobra, porque esValido ya es un booleano; escribe simplemente if esValido. Peor aun es if !esValido == false, un trabalenguas que significa lo mismo que if esValido. Mantien las condiciones simples y directas: tu codigo se lee muchas mas veces de las que se escribe.

3.7 La sentencia switch: elegir entre muchos caminos

Cuando tienes que elegir entre muchas opciones basadas en el valor de una expresion, encadenar else if se vuelve tedioso y dificil de leer. Para eso existe switch, que evalua una expresion una sola vez y la compara contra una lista de casos. La sintaxis es switch expresion { case valor1: ...; case valor2, valor3: ...; default: ... }. Cada case lista uno o varios valores separados por comas, y default atrapa todo lo que no coincidio con ningun caso. Es como un menu: la expresion es lo que pide el cliente y cada case es un platillo con su receta.

Aqui viene la gran diferencia con otros lenguajes y una de las decisiones de diseno mas celebradas de Go: en Go, cada caso termina automaticamente, sin necesidad de escribir break. En C o Java, si olvidas el break al final de un caso, la ejecucion se cae al siguiente caso por accidente, un bug clasico y dificil de detectar. Go elimino ese problema de raiz: los casos no se caen solos. Si por alguna razon extraordinaria quieres que la ejecucion continue al siguiente caso, debes pedirlo explicitamente con la palabra fallthrough, y debe ser la ultima instruccion del caso. En la practica, fallthrough se usa poquissimo; saber que existe y que casi nunca lo necesitaras es suficiente.

El switch de Go tiene dos superpoderes mas. El primero es el switch sin expresion: si omites la expresion, Go asume switch true, y cada case lleva una condicion completa. Esto convierte al switch en un reemplazo elegante para cadenas largas de else if: switch { case puntos >= 90: ...; case puntos >= 80: ... }. El segundo superpoder es la inicializacion corta, igual que en el if: switch dia := hoy(); dia { ... } o switch n := len(lista); { case n == 0: ... }. También puedes usar switch para comparar tipos con el type switch, que veremos cuando estudiemos interfaces, pero por ahora quedate con la idea de que switch es mucho mas flexible de lo que parece.

```
package main

import (
    "fmt"
    "time"
)

func main() {
    // Switch clasico: cada case termina solo, sin break
    dia := 3
    switch dia {
    case 1:
```

```

    fmt.Println("Lunes: a empezar la semana")
case 2, 3, 4:
    fmt.Println("Mitad de semana: a darle")
case 5:
    fmt.Println("Viernes: ya casi")
case 6, 7:
    fmt.Println("Fin de semana: a descansar")
default:
    fmt.Println("Dia invalido")
}

// Switch sin expresion: reemplaza else-if encadenados
puntos := 85
switch {
case puntos >= 90:
    fmt.Println("Calificacion: A")
case puntos >= 80:
    fmt.Println("Calificacion: B")
case puntos >= 70:
    fmt.Println("Calificacion: C")
default:
    fmt.Println("Reprobado")
}

// Switch con inicializacion corta
switch hora := time.Now().Hour(); {
case hora < 12:
    fmt.Println("Buenos dias")
case hora < 20:
    fmt.Println("Buenas tardes")
default:
    fmt.Println("Buenas noches")
}

// fallthrough: casi nunca lo necesitaras, pero existe
numero := 2
switch numero {
case 1:
    fmt.Println("uno")
    fallthrough
case 2:
    fmt.Println("dos (y tambien cae aqui desde el 1)")
case 3:
    fmt.Println("tres")
}

```

```
}
```

Como buena practica, usa switch cuando compares una misma expresion contra varios valores posibles, y usa el switch sin expresion cuando tengas varias condiciones booleanas relacionadas. Pon siempre un default aunque creas que cubriste todos los casos: es tu red de seguridad para valores inesperados y documenta que pensaste en ellos. Y evita fallthrough salvo que tengas una razon muy clara y la documentes con un comentario: si un lector ve fallthrough sin explicacion, asumira que es un bug.

3.8 El bucle for: uno solo para gobernarlos a todos

En la mayoría de los lenguajes hay varios tipos de bucle: for, while, do-while, foreach. Go decidio que uno basta, y tiene razon: el for de Go puede hacer todo lo que hacen los demas, con menos sintaxis que aprender. El for clasico tiene tres partes separadas por punto y coma: inicializacion, condicion y post-instruccion. En `for i := 0; i < 10; i++`, primero se ejecuta `i := 0` una sola vez, después antes de cada vuelta se evalua `i < 10` y si es false el bucle termina, y al final de cada vuelta se ejecuta `i++`. Es el bucle contador de toda la vida, perfecto cuando sabes cuantas veces quieres repetir algo: recorrer del 1 al 100, procesar diez intentos, dibujar una piramide de asteriscos.

Si omites la inicializacion y la post-instruccion y dejas solo la condicion, el for se convierte en un while: `for saldo > 0 { ... }`. El bucle se repite mientras la condicion sea verdadera, y es tu responsabilidad que algo dentro del cuerpo acerque la condicion a su fin; si no, tienes un bucle infinito. Y si omites hasta la condicion y escribes `for { ... }`, obtienes un bucle infinito puro, que solo se detiene con un `break` desde adentro o con un `return` de la funcion. Los bucles infinitos suenan peligrosos pero son muy utiles: servidores que atienden peticiones sin parar, programas que leen la entrada del usuario hasta que escriba salir, reintentos de una operacion. La clave es que siempre haya una salida clara y alcanzable.

La cuarta forma es el for-range, la manera idiomática de recorrer colecciones en Go: `for indice, valor := range miSlice { ... }`. En cada vuelta te entrega el indice y una copia del elemento, sin que tengas que manejar contadores manualmente. Si solo necesitas el indice, escribe `for i := range miSlice`; si solo necesitas el valor, ignora el indice con el guion bajo: `for _, v := range miSlice`. El range también funciona con mapas, cadenas (entregando runas, o sea caracteres Unicode) y canales, como veras en unidades posteriores. Un detalle moderno importante: desde Go 1.22, la variable del bucle es nueva en cada iteracion, lo que significa que cada vuelta tiene su propia copia. En versiones anteriores se reutilizaba la misma variable, lo que causaba sorpresas famosas al crear closures dentro del bucle. Como este libro apunta a Go 1.21 en adelante, conviene que sepas del cambio: si ves codigo viejo que copia la variable del bucle a otra variable dentro del cuerpo, era un parche para ese comportamiento antiguo.

```

package main

import "fmt"

func main() {
    // For clasico: contador de 1 a 5
    for i := 1; i <= 5; i++ {
        fmt.Println("Vuelta", i)
    }

    // For como while: se repite mientras la condicion sea true
    saldo := 100
    for saldo > 0 {
        fmt.Println("Comprando con saldo:", saldo)
        saldo -= 30 // importante: acercar el fin del bucle
    }

    // For infinito con break: pedir hasta que el usuario acierte
    secreto := 7
    intento := 0
    for {
        intento++
        fmt.Println("Intento", intento)
        if intento == secreto {
            fmt.Println("Acertaste en el intento", intento)
            break // salir del bucle infinito
        }
        if intento > 10 {
            fmt.Println("Demasiados intentos")
            break
        }
    }

    // For-range: recorrer un slice sin contadores manuales
    frutas := []string{"mango", "papaya", "guayaba"}
    for i, fruta := range frutas {
        fmt.Printf("Fruta %d: %s\n", i, fruta)
    }

    // Solo los valores, ignorando el indice
    for _, fruta := range frutas {
        fmt.Println("Me gusta la", fruta)
    }
}

```

Los errores clasicos con for merecen su propia lista de advertencias. El bucle infinito accidental es el numero uno: una condicion que nunca se vuelve false, como `for i := 0; i < 10; i--` donde el contador se aleja en vez de acercarse, o un `for saldo > 0` que nunca modifica el saldo. El numero dos es el error de uno mas o uno menos en los limites: usar `<=` cuando debia ser `<` y procesar un elemento de mas, o viceversa. El numero tres es modificar la coleccion mientras la recorres con `range`, lo cual produce resultados impredecibles. Y el numero cuatro, especifico de `range`: recordar que el valor es una copia, asi que modificar valor dentro del bucle no cambia el elemento original del slice; si necesitas modificarlo, usa el indice: `miSlice[i] = nuevoValor`. Graba estas cuatro advertencias y te ahorraran horas de depuracion.

3.9 break y continue: control fino del bucle

A veces necesitas alterar el flujo normal de un bucle: salir antes de tiempo o saltarte una vuelta. Para eso existen `break` y `continue`, dos palabras pequenas con efectos precisos. La sentencia `break` termina el bucle de inmediato, sin importar cuantas vueltas faltaban, y la ejecucion continua con la primera linea después del bucle. Es la salida de emergencia: la usas cuando ya encontraste lo que buscabas, cuando el usuario pidio salir, o cuando detectaste una condicion que hace inutil seguir. La sentencia `continue` hace algo mas sutil: abandona la vuelta actual y salta directamente a la siguiente, pasando por la post-instruccion del `for`. Es el filtro: la usas para decir esta vuelta no me interesa, sigue con la proxima.

Un detalle que sorprende a muchos: `break` dentro de un `switch` sale del `switch`, no de un `for` que lo rodee. Si tienes un `for` con un `switch` adentro y quieres que un caso del `switch` termine el bucle completo, un `break` a secas no te sirve: solo saldria del `switch` y el `for` seguiria. Para esos casos existen las etiquetas, que veremos en la siguiente seccion. Por otro lado, `continue` dentro de un `switch` que esta dentro de un `for` si afecta al `for`, porque `continue` siempre se refiere al bucle, nunca al `switch`. Son matices que solo se aprenden usandolos, asi que practica con ejemplos pequenos hasta que el comportamiento te parezca obvio.

```

package main

import "fmt"

func main() {
    // break: salir al encontrar lo buscado
    numeros := []int{3, 7, 12, 5, 9}
    buscado := 12
    for i, n := range numeros {
        if n == buscado {
            fmt.Printf("Encontrado %d en la posicion %d\n", buscado, i)
            break // ya no hay nada que buscar
        }
        fmt.Println("Revisando:", n)
    }

    // continue: saltar los que no interesan
    fmt.Println("Numeros impares del 1 al 10:")
    for i := 1; i <= 10; i++ {
        if i%2 == 0 {
            continue // los pares se saltan
        }
        fmt.Println(i)
    }

    // continue dentro de switch+for: afecta al for
    fmt.Println("Procesando, saltando errores:")
    for i := 1; i <= 5; i++ {
        switch {
        case i == 3:
            fmt.Println(" vuelta 3: error, se salta")
            continue // salta a la siguiente vuelta del for
        default:
            fmt.Println(" vuelta", i, "procesada")
        }
    }
}

```

Como buena practica, usa break para expresar encuentre lo que buscaba o ya no tiene caso seguir, y usa continue al inicio del cuerpo del bucle como filtro, para que el resto del codigo quede al nivel principal sin anidacion extra. Evita usar break y continue de formas creativas o multiples por bucle: si un bucle necesita tres breaks en distintos lugares, probablemente la logica esta enredada y conviene reorganizarla, quiza extrayendo el bucle a una funcion donde puedas usar return.

3.10 Etiquetas y goto: poder con responsabilidad

Cuando tienes bucles anidados, un break o un continue a secas solo afectan al bucle mas interno. Pero a veces necesitas mas alcance: salir de dos bucles a la vez porque ya encontraste lo que buscabas en una matriz, por ejemplo. Para eso existen las etiquetas o labels: marcas un bucle con un nombre seguido de dos puntos, como Busqueda:, y luego puedes escribir break Busqueda o continue Busqueda para referirte a ese bucle en especifico. Es una herramienta precisa y legitima, y es la solucion idiomatica al problema del break dentro de switch anidado en for que mencionamos antes: etiquetas el for y haces break Etiqueta desde el caso del switch.


```

package main

import "fmt"

func main() {
    // Etiqueta para salir de bucles anidados
    matriz := [][]int{
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9},
    }
    buscado := 5
    encontrado := false

    Busqueda:
    for fila, renglon := range matriz {
        for col, valor := range renglon {
            if valor == buscado {
                fmt.Printf("Encontrado %d en [%d][%d]\n", buscado, fila, col)
                encontrado = true
                break Busqueda // sale de AMBOS bucles
            }
        }
    }
    if !encontrado {
        fmt.Println("No se encontro el valor")
    }

    // continue con etiqueta: saltar a la siguiente vuelta del bucle externo
    fmt.Println("Tablas, saltando la del 3:")
    Tablas:
    for i := 1; i <= 5; i++ {
        for j := 1; j <= 3; j++ {
            if i == 3 {
                continue Tablas // salta a la siguiente i
            }
            fmt.Printf("%d x %d = %d  ", i, j, i*j)
        }
        fmt.Println()
    }
}

```

Y ahora hablemos de goto, la palabra reservada mas polemica de la programacion. La sentencia goto salta la ejecucion a una etiqueta en cualquier punto de la funcion: goto Fin salta

a la línea marcada con `Fin`:. Existe en Go, compila y funciona. Y sin embargo, la recomendación es clara: evítalo. La razón es histórica y práctica a la vez: el `goto` indiscriminado produce código espagueti, donde el flujo salta de un lado a otro sin estructura y nadie puede seguir la lógica ni probar que el programa hace lo correcto. Fue precisamente el abuso del `goto` lo que motivó la invención de la programación estructurada con `if`, `for` y funciones en primer lugar. Go incluye `goto` más por completitud que por necesidad, y tiene restricciones: no puedes saltar por encima de declaraciones de variables nuevas, por ejemplo.

Dicho eso, la honestidad intelectual obliga a mencionar que `goto` tiene un par de usos legítimos y raros, como salir de bucles profundamente anidados en código de muy bajo nivel o manejar limpieza de errores en código estilo C. Pero en el código Go cotidiano, incluyendo todo lo que harás en este libro, nunca lo necesitarás: las etiquetas con `break` y `continue`, los retornos tempranos y las funciones pequeñas cubren todos los casos de forma más legible. Si alguna vez sientes la tentación de usar `goto`, tómalo como una señal de que tu función es demasiado grande o tu lógica demasiado enredada, y reorganízala. Tus compañeros de equipo y tu yo del futuro te lo agradecerán.

Ejemplos paso a paso

Ejemplo 1: Calculadora de cuenta de restaurante

Problema: un grupo de amigos quiere dividir la cuenta. El programa debe sumar los platillos, agregar 16% de IVA, agregar 10% de propina sobre el subtotal y dividir el total entre el número de personas. Este ejemplo practica operadores aritméticos con flotantes y asignación compuesta.

```

package main

import "fmt"

func main() {
    // Precios de los platillos
    platillo1 := 185.50
    platillo2 := 220.00
    platillo3 := 149.99
    personas := 3

    // Subtotal con +=
    subtotal := 0.0
    subtotal += platillo1
    subtotal += platillo2
    subtotal += platillo3

    // IVA del 16% y propina del 10% sobre el subtotal
    iva := subtotal * 0.16
    propina := subtotal * 0.10
    total := subtotal + iva + propina

    // Division flotante: el resultado conserva decimales
    porPersona := total / float64(personas)

    fmt.Printf("Subtotal: $.2f\n", subtotal)
    fmt.Printf("IVA (16%): $.2f\n", iva)
    fmt.Printf("Propina (10%): $.2f\n", propina)
    fmt.Printf("Total: $.2f\n", total)
    fmt.Printf("Cada persona paga: $.2f\n", porPersona)
}

```

Observa que usamos float64 en toda la operacion porque el dinero necesita decimales. Si personas fuera entero y total también, la division seria entera y perderiamos centavos. La conversion float64(personas) es el puente explicito que Go exige entre tipos.

Ejemplo 2: El promedio y la trampa de la división entera

Problema: calcular el promedio de tres calificaciones enteras. La primera version muestra el bug clasico; la segunda lo corrige. Este ejemplo te ensena a detectar la division entera antes de que te muerda en un programa real.

```

package main

import "fmt"

func main() {
    cal1, cal2, cal3 := 9, 8, 8
    suma := cal1 + cal2 + cal3 // 25

    // Version con bug: division entera descarta el decimal
    promedioMal := suma / 3
    fmt.Println("Promedio (entero):", promedioMal) // 8, pero el real es 8.33

    // Version correcta: convertir a flotante antes de dividir
    promedioBien := float64(suma) / 3
    fmt.Printf("Promedio (decimal): %.2f\n", promedioBien) // 8.33

    // Regla de oro: si quieres decimales en el resultado,
    // tiene que haber decimales en la operacion.
}

```

Ejemplo 3: Par, impar y múltiplos con el operador módulo

Problema: dado un numero, decir si es par o impar, y además indicar si es multiplo de 3 y de 5. El modulo es la herramienta perfecta para preguntas de divisibilidad.

```

package main

import "fmt"

func main() {
    numero := 30

    if numero%2 == 0 {
        fmt.Println(numero, "es par")
    } else {
        fmt.Println(numero, "es impar")
    }

    // Multiplos: el residuo debe ser cero
    if numero%3 == 0 && numero%5 == 0 {
        fmt.Println(numero, "es multiplo de 3 y de 5")
    } else if numero%3 == 0 {
        fmt.Println(numero, "es multiplo de 3")
    } else if numero%5 == 0 {
        fmt.Println(numero, "es multiplo de 5")
    } else {
        fmt.Println(numero, "no es multiplo ni de 3 ni de 5")
    }
}

```

Ejemplo 4: Control de acceso con operadores lógicos

Problema: un sistema decide si un usuario puede entrar a una zona VIP. Reglas: debe ser mayor de edad Y (ser socio O tener invitacion). Además, si la cuenta esta suspendida, nadie entra. Este ejemplo combina &&, ||, ! y parentesis.

```

package main

import "fmt"

func main() {
    edad := 25
    esSocio := false
    tieneInvitacion := true
    cuentaSuspendida := false

    // (esSocio || tieneInvitacion) && edad >= 18 && !cuentaSuspendida
    puedeEntrar := (esSocio || tieneInvitacion) && edad >= 18 && !cuentaSuspendida

    if puedeEntrar {
        fmt.Println("Bienvenido a la zona VIP")
    } else {
        fmt.Println("Acceso denegado")
    }

    // Cortocircuito como proteccion: si codigo es "", no se evalua codigo[0]
    codigo := ""
    if codigo != "" && codigo[0] == 'A' {
        fmt.Println("Codigo especial detectado")
    } else {
        fmt.Println("Sin codigo especial (y sin panico por indice)")
    }
}

```

Sin el cortocircuito, evaluar `codigo[0]` con un string vacío provocaría un pánico por índice fuera de rango. El `&&` nos protege gratis.

Ejemplo 5: Cajero automático con if e inicialización corta

Problema: simular un retiro de cajero. El programa recibe el saldo y el monto a retirar, valida que el monto sea positivo y que haya fondos suficientes, y actualiza el saldo. Usa inicialización corta y guard clauses.

```

package main

import "fmt"

func main() {
    saldo := 1500.0
    retiro := 600.0

    // Guard clauses: validar primero, salir rapido si algo esta mal
    if retiro <= 0 {
        fmt.Println("Error: el monto debe ser positivo")
        return
    }
    if retiro > saldo {
        fmt.Println("Error: fondos insuficientes")
        return
    }

    // Camino feliz al nivel principal, sin anidacion
    saldo -= retiro
    fmt.Printf("Retiro exitoso. Nuevo saldo: $%.2f\n", saldo)

    // Inicializacion corta: la comision vive solo dentro del if
    if comision := retiro * 0.02; comision > 50 {
        fmt.Printf("Comision alta: $%.2f\n", comision)
    } else {
        fmt.Printf("Comision aplicada: $%.2f\n", comision)
    }
}

```

Ejemplo 6: Menú de restaurante con switch

Problema: mostrar el platillo del día según el número elegido del 1 al 5. Varios números pueden compartir el mismo platillo. Este ejemplo muestra casos múltiples y default.

```

package main

import "fmt"

func main() {
    opcion := 2

    switch opcion {
    case 1:
        fmt.Println("Lunes: Tacos al pastor")
    case 2, 4:
        fmt.Println("Martes y jueves: Enchiladas verdes")
    case 3:
        fmt.Println("Miercoles: Pozole rojo")
    case 5:
        fmt.Println("Viernes: Pescado a la talla")
    default:
        fmt.Println("Opcion invalida, elige del 1 al 5")
    }
    // No se necesita break: cada caso termina solo.
}

```

Ejemplo 7: Categorías por edad con switch sin expresión

Problema: clasificar a una persona en niño, adolescente, adulto o adulto mayor según su edad. El switch sin expresión reemplaza elegantemente una cadena de else if.


```

package main

import "fmt"

func main() {
    edad := 34

    switch {
    case edad < 0:
        fmt.Println("Edad invalida")
    case edad < 13:
        fmt.Println("Categoria: nino")
    case edad < 20:
        fmt.Println("Categoria: adolescente")
    case edad < 60:
        fmt.Println("Categoria: adulto")
    default:
        fmt.Println("Categoria: adulto mayor")
    }
}

```

Fijate en el orden: los casos se evalúan de arriba hacia abajo y se ejecuta el primero que sea verdadero. Si pusieras `case edad < 60` antes que `case edad < 13`, un niño de 8 años caería en adulto. El orden de los casos es parte de la lógica.

Ejemplo 8: Tabla de multiplicar con for clásico

Problema: imprimir la tabla de multiplicar del 7, del 1 al 10. El for clásico con contador es ideal cuando sabes exactamente cuántas vueltas necesitas.

```

package main

import "fmt"

func main() {
    numero := 7

    for i := 1; i <= 10; i++ {
        fmt.Printf("%d x %d = %d\n", numero, i, numero*i)
    }
}

```

Ejemplo 9: Encontrar el mayor con for-range

Problema: dada una lista de ventas del día, encontrar la venta mayor y en que posición ocurrió. El for-range entrega índice y valor sin contadores manuales.

```
package main

import "fmt"

func main() {
    ventas := []float64{120.50, 89.99, 340.00, 210.75, 340.00, 95.00}

    mayor := ventas[0]
    posicion := 0

    for i, v := range ventas {
        if v > mayor {
            mayor = v
            posicion = i
        }
    }

    fmt.Printf("La mayor venta fue %.2f en la posicion %d\n", mayor, posicion)
}
```

Nota que si hay dos ventas iguales al máximo, nos quedamos con la primera, porque la condición es > estricto. Si quisieras la última, usarías >=. Esos detalles de comparación cambian el comportamiento y debes decidirlos conscientemente.

Ejemplo 10: FizzBuzz, el clásico de las entrevistas

Problema: imprimir los números del 1 al 20, pero si el número es múltiplo de 3 imprimir Fizz, si es múltiplo de 5 imprimir Buzz, y si es múltiplo de ambos imprimir FizzBuzz. Este ejercicio legendario combina for, módulo, if-else y continue.

```

package main

import "fmt"

func main() {
    for i := 1; i <= 20; i++ {
        switch {
            case i%3 == 0 && i%5 == 0:
                fmt.Println("FizzBuzz")
            case i%3 == 0:
                fmt.Println("Fizz")
            case i%5 == 0:
                fmt.Println("Buzz")
            default:
                fmt.Println(i)
        }
    }
}

```

El orden de los casos es crucial: el caso de multiplo de ambos debe ir primero, porque si evaluaras el de 3 primero, el 15 diria solo Fizz. Es el mismo principio del ejemplo 7: los casos mas especificos van antes que los generales.

Ejemplo 11: Búsqueda en matriz con etiqueta

Problema: buscar un valor en una tabla de dos dimensiones y detenerse en cuanto se encuentre, saliendo de ambos bucles a la vez. Sin etiqueta, el break solo saldria del bucle interno.

```

package main

import "fmt"

func main() {
    tabla := [][]int{
        {11, 22, 33},
        {44, 55, 66},
        {77, 88, 99},
    }
    buscado := 55

    Busqueda:
    for f := 0; f < len(tabla); f++ {
        for c := 0; c < len(tabla[f]); c++ {
            if tabla[f][c] == buscado {
                fmt.Printf("Encontrado en fila %d, columna %d\n", f, c)
                break Busqueda
            }
        }
    }
    fmt.Println("Busqueda terminada")
}

```

Ejemplo 12: El mayor de tres números

Problema: dados tres numeros, determinar cual es el mayor usando if-else-if encadenado. Este ejemplo practica comparaciones multiples y el orden de evaluacion.

```

package main

import "fmt"

func main() {
    a, b, c := 15, 42, 27

    // Se evalua en orden: el primer true gana
    if a >= b && a >= c {
        fmt.Println("El mayor es a:", a)
    } else if b >= a && b >= c {
        fmt.Println("El mayor es b:", b)
    } else {
        fmt.Println("El mayor es c:", c)
    }
}

```

Usamos `>=` en lugar de `>` para que los empates se resuelvan de forma determinista: si a y b valen lo mismo y son los mayores, gana a por ir primero. Cuando hay empates posibles, decide y documenta la regla de desempate en lugar de dejarla al azar.

Ejemplo 13: Cuenta regresiva con for como while

Problema: simular la cuenta regresiva de un despegue desde 10 hasta 0, imprimiendo cada numero y al final Despegue. El for con solo condicion es perfecto cuando el numero de vueltas depende de una variable que cambia dentro del bucle.

```

package main

import (
    "fmt"
    "time"
)

func main() {
    contador := 10

    // For como while: se repite mientras contador sea mayor que cero
    for contador > 0 {
        fmt.Println(contador, "...")
        contador-- // sin este decremento, el bucle seria infinito
        time.Sleep(300 * time.Millisecond)
    }
    fmt.Println("Despegue!")
}

```

Este patron aparece en reintentos de red (intentar mientras queden intentos), procesamiento por lotes (mientras haya elementos en la cola) y animaciones. La regla de seguridad es siempre la misma: algo dentro del cuerpo debe acercar la condicion a su fin.

Ejemplo 14: Descuento por membresía

Problema: una tienda aplica descuentos según el tipo de membresia del cliente: oro 20%, plata 10%, bronce 5%, sin membresia 0%. El programa calcula el precio final usando switch con inicializacion corta y muestra el ahorro.

```

package main

import "fmt"

func main() {
    precio := 1250.0
    membresia := "plata"

    // Inicializacion corta: el descuento vive solo dentro del switch
    switch descuento := membresia; descuento {
    case "oro":
        precio *= 0.80 // 20% de descuento
        fmt.Println("Membresia oro: 20% de descuento")
    case "plata":
        precio *= 0.90 // 10% de descuento
        fmt.Println("Membresia plata: 10% de descuento")
    case "bronce":
        precio *= 0.95 // 5% de descuento
        fmt.Println("Membresia bronce: 5% de descuento")
    default:
        fmt.Println("Sin membresia: sin descuento")
    }

    fmt.Printf("Precio final: $%.2f\n", precio)
}

```

Este ejemplo combina tres temas de la unidad en pocas líneas: switch con inicialización corta, casos múltiples implícitos (cada membresía es un caso) y asignación compuesta (*=) para aplicar el descuento. Es el tipo de lógica de negocio que aparece en sistemas reales todos los días.

Práctica

Ejercicio 1: Calculadora con switch

Escribe un programa que lea dos números y un operador (+, -, *, /) y muestre el resultado. Si el operador es división y el divisor es cero, debe mostrar un mensaje de error en lugar de provocar un pánico. Usa switch para elegir la operación.

Solución explicada: el switch elige la operación según el operador. El caso de la división incluye una guarda con if para el divisor cero: esa combinación de switch por fuera e if por

dentro es muy comun. El default atrapa operadores invalidos. Nota como cada caso termina solo, sin break.

```
package main

import "fmt"

func main() {
    a, b := 20.0, 4.0
    operador := "/"

    switch operador {
    case "+":
        fmt.Println("Resultado:", a+b)
    case "-":
        fmt.Println("Resultado:", a-b)
    case "*":
        fmt.Println("Resultado:", a*b)
    case "/":
        if b == 0 {
            fmt.Println("Error: no se puede dividir entre cero")
        } else {
            fmt.Println("Resultado:", a/b)
        }
    default:
        fmt.Println("Operador invalido")
    }
}
```

Ejercicio 2: Año bisiesto

Un año es bisiesto si es divisible entre 4, excepto los divisibles entre 100, a menos que también sean divisibles entre 400. Escribe un programa que diga si un año dado es bisiesto. Ejemplo: 2024 si, 1900 no, 2000 si.

Solucion explicada: la regla se traduce directamente a operadores logicos: $(\text{año} \% 4 == 0 \ \&\& \ \text{año} \% 100 != 0) \ || \ \text{año} \% 400 == 0$. Los parentesis hacen explicita la agrupacion aunque la precedencia ya la resolveria igual: la legibilidad manda. Pruebalo con 2024 (true), 1900 (false) y 2000 (true) para verificar los tres caminos.


```
package main

import "fmt"

func main() {
    ano := 2024

    esBisiesto := (ano%4 == 0 && ano%100 != 0) || ano%400 == 0

    if esBisiesto {
        fmt.Println(ano, "es bisiesto")
    } else {
        fmt.Println(ano, "no es bisiesto")
    }
}
```

Ejercicio 3: Suma de pares hasta N

Escribe un programa que sume todos los numeros pares del 1 hasta N (incluyendo N si es par) usando un for. Luego resuelvelo de una segunda forma usando continue para saltar los impares. Compara ambas versiones.

Solucion explicada: la primera version avanza el contador de 2 en 2, asi solo toca pares: es la mas eficiente. La segunda version recorre todos los numeros y usa continue para saltar los impares: es mas legible para algunos y muestra el uso de continue como filtro. Ambas dan el mismo resultado; elegir entre ellas es cuestion de estilo y contexto.

```

package main

import "fmt"

func main() {
    n := 10

    // Version 1: avanzar de 2 en 2
    suma1 := 0
    for i := 2; i <= n; i += 2 {
        suma1 += i
    }
    fmt.Println("Suma de pares (version 1):", suma1) // 30

    // Version 2: recorrer todo y filtrar con continue
    suma2 := 0
    for i := 1; i <= n; i++ {
        if i%2 != 0 {
            continue // saltar impares
        }
        suma2 += i
    }
    fmt.Println("Suma de pares (version 2):", suma2) // 30
}

```

Ejercicio 4: ¿Es primo?

Escribe un programa que determine si un número es primo. Un primo solo es divisible entre 1 y el mismo. Usa un for que pruebe divisores desde 2 hasta la raíz cuadrada del número, y sal con break en cuanto encuentres un divisor.

Solución explicada: si ningún número entre 2 y la raíz cuadrada divide a n , entonces n es primo. Probar solo hasta la raíz cuadrada es una optimización matemática: si n tuviera un divisor mayor que su raíz, el divisor complementario sería menor que la raíz y ya lo habríamos encontrado. El break evita trabajo inútil en cuanto sabemos la respuesta. Los casos especiales: números menores que 2 no son primos, y el 2 sí lo es.

```

package main

import (
    "fmt"
    "math"
)

func main() {
    n := 29

    if n < 2 {
        fmt.Println(n, "no es primo")
        return
    }

    esPrimo := true
    limite := int(math.Sqrt(float64(n)))
    for i := 2; i <= limite; i++ {
        if n%i == 0 {
            esPrimo = false
            break // ya sabemos que no es primo, no seguir buscando
        }
    }

    if esPrimo {
        fmt.Println(n, "es primo")
    } else {
        fmt.Println(n, "no es primo")
    }
}

```

Ejercicio 5: Invertir un número

Escribe un programa que invierta los dígitos de un número entero (1234 se vuelve 4321) usando un for como while. Pista: en cada vuelta, toma el último dígito con `% 10`, agrégalo al resultado multiplicado por 10, y quita el último dígito con `/ 10`.

Solución explicada: este es el algoritmo clásico de manipulación de dígitos. Con `numero = 1234`: $1234 \% 10 = 4$, `resultado = 0*10 + 4 = 4`; $1234 / 10 = 123$; $123 \% 10 = 3$, `resultado = 4*10 + 3 = 43`; y así hasta que `numero` llegue a 0. El for como while (`for numero > 0`) es natural aquí porque no sabemos cuántas vueltas serán: depende de cuántos dígitos tenga el número.

```

package main

import "fmt"

func main() {
    numero := 1234
    original := numero
    invertido := 0

    for numero > 0 {
        digito := numero % 10          // ultimo digito
        invertido = invertido*10 + digito // agregarlo a la derecha
        numero = numero / 10          // quitar el ultimo digito
    }

    fmt.Println("Original:", original)
    fmt.Println("Invertido:", invertido) // 4321
}

```

Ejercicio 6: Triángulo de asteriscos

Escribe un programa que dibuje un triangulo rectangulo de asteriscos con N filas usando for anidados. La fila 1 tiene 1 asterisco, la fila 2 tiene 2, y así sucesivamente.

Solucion explicada: el bucle externo controla la fila actual (de 1 a N) y el bucle interno imprime tantos asteriscos como indique la fila. Después del bucle interno, un `fmt.Println()` baja a la siguiente linea. Los bucles anidados son la herramienta natural para dibujar figuras y recorrer tablas: el externo avanza en una dimension y el interno en la otra.

```

package main

import "fmt"

func main() {
    n := 5

    for fila := 1; fila <= n; fila++ {
        for col := 1; col <= fila; col++ {
            fmt.Print("* ")
        }
        fmt.Println() // salto de linea al terminar la fila
    }
    // Dibuja:
    // *
    // * *
    // * * *
    // * * * *
    // * * * * *
}

```

Ejercicio 7: Adivina el número

Escribe el juego de adivinar un numero secreto entre 1 y 20. El programa tiene el secreto guardado y simula una serie de intentos (puedes usar un slice con intentos fijos para practicar, o leer de la entrada). En cada intento debe decir si el secreto es mayor o menor, contar los intentos y terminar con break al acertar. Si se acaban los intentos, debe revelar el secreto.

Solucion explicada: usamos un for-range sobre los intentos y break al acertar, combinando comparaciones con mensajes de pista. El else-if encadenado decide entre las tres posibilidades: igual (acierto), menor (pista) o mayor (pista). Un booleano acerto registra si se gano; si el bucle termina sin acierto, se revela el secreto. Este ejercicio integra for, break, if-else-if y contadores en un programa con logica de juego real.

```

package main

import "fmt"

func main() {
    secreto := 13
    intentos := []int{5, 18, 11, 13, 20}
    acerto := false

    for i, intento := range intentos {
        fmt.Printf("Intento %d: dices %d... ", i+1, intento)
        if intento == secreto {
            fmt.Println("Correcto! Adivinaste el numero")
            acerto = true
            break
        } else if intento < secreto {
            fmt.Println("El secreto es MAYOR")
        } else {
            fmt.Println("El secreto es MENOR")
        }
    }

    if !acerto {
        fmt.Println("Se acabaron los intentos. El secreto era:", secreto)
    }
}

```

Ejercicio 8: Contador de vocales

Escribe un programa que cuente cuantas vocales (a, e, i, o, u, sin distinguir mayúsculas de minúsculas) hay en una frase, usando for-range sobre el string y un switch para detectarlas.

Solucion explicada: el for-range sobre un string entrega runas (caracteres), y el switch con casos multiples agrupa las cinco vocales en minúsculas y mayúsculas. Convertir cada runa a minúscula con `unicode.ToLower` simplificaría los casos, pero la version explicita muestra bien el uso de casos multiples. Es un buen ejemplo de como range, switch y contadores trabajan juntos en tareas de procesamiento de texto.

```

package main

import "fmt"

func main() {
    frase := "Lucas EGA Academy ensena programacion"
    vocales := 0

    for _, letra := range frase {
        switch letra {
            case 'a', 'e', 'i', 'o', 'u',
                  'A', 'E', 'I', 'O', 'U':
                vocales++
        }
    }

    fmt.Printf("La frase tiene %d vocales\n", vocales)
}

```

Resumen de la unidad

En esta unidad convertiste datos pasivos en programas que calculan y deciden. Aprendiste los operadores aritmeticos (+, -, *, /, %) y la regla de oro de Go: los operandos deben ser del mismo tipo, la division entre enteros trunca los decimales y el modulo solo trabaja con enteros. Dominaste los operadores de asignacion (=, +=, -=, *=, /=, %=) y el incremento y decremento (++ , --), con su particularidad gofer de ser sentencias y no expresiones. Viste los operadores de comparacion (==, !=, <, >, <=, >=), incluyendo la trampa de comparar flotantes con == y su solucion con tolerancia. Combinaste condiciones con los operadores logicos (&&, ||, !), aprovechaste el cortocircuito como guarda de seguridad y aprendiste la precedencia para saber que se evalua primero, con los parentesis como tu mejor aliado.

En control de flujo, el if se volvio tu herramienta de decision, con su inicializacion corta (if x := f(); x > 0) y las guard clauses para evitar anidacion. El switch te dio una forma elegante de elegir entre muchos caminos, sin breaks olvidados gracias a que cada caso termina solo, con casos multiples, switch sin expresion y la inicializacion corta. Descubriste que for es el unico bucle de Go y que con sus cuatro formas (clasico, como while, infinito con break y range) puedes repetir cualquier tarea. Afinaste el control con break y continue, extendiste su alcance con etiquetas para bucles anidados y conociste goto lo suficiente para saber por que evitarlo.

Lo mas importante no es memorizar cada operador, sino interiorizar dos ideas: Go prefiere lo explicito y predecible sobre lo magico, y el codigo se lee muchas mas veces de las que se escribe, asi que la claridad siempre gana. Practica los ejercicios hasta que te salgan sin mirar, rompe los ejemplos a proposito para ver que pasa y lleva estas herramientas a la siguiente unidad, donde las empaquetaras en funciones reutilizables. El verdadero poder de un programador no esta en conocer muchas estructuras, sino en combinar unas pocas con maestria.

UNIDAD 4: Funciones y paquetes

Hay un momento en la vida de todo programador en que escribe el mismo pedazo de código por tercera vez, lo copia, lo pega, cambia un par de nombres y piensa: tiene que haber una forma mejor de hacer esto. Ese momento es una señal de madurez, porque significa que tu intuición ya detectó uno de los principios más importantes de la programación: no te repitas. Las funciones son la respuesta que los lenguajes de programación inventaron para ese problema, y los paquetes son la respuesta al problema siguiente, que aparece cuando ya tienes tantas funciones que necesitas organizarlas. Esta unidad trata de esas dos ideas, y juntas van a cambiar por completo la forma en que escribes programas.

Piensa en una función como una pequeña máquina dentro de tu programa. Le entregas materia prima por un lado, que son los parámetros o argumentos, la máquina hace su trabajo por dentro sin que tengas que preocuparte por los detalles, y te devuelve un producto terminado por el otro lado, que es el valor de retorno. Una función que calcula el área de un círculo recibe el radio, eleva al cuadrado, multiplica por pi y te devuelve el área. Tú no necesitas recordar la fórmula cada vez: solo llamas a la función. Esa es la magia de la abstracción: esconder la complejidad detrás de un nombre claro. Los buenos programas están llenos de funciones con nombres tan descriptivos que leer el código se parece a leer una receta de cocina: calentar horno, mezclar ingredientes, hornear treinta minutos, dejar enfriar. Cada paso es una función, y cada función sabe hacer bien una sola cosa.

En la unidad anterior aprendiste a tomar decisiones y a repetir tareas con `if`, `switch` y `for`. Ahora vas a aprender a empaquetar esas decisiones y repeticiones en unidades reutilizables. Verás que en Go las funciones son ciudadanas de primera clase: no solo las declaras y las llamas, también puedes guardarlas en variables, pasarlas como argumentos a otras funciones y crearlas sobre la marcha sin nombre, como funciones anónimas. Aprenderás el famoso patrón de múltiples valores de retorno, donde una función devuelve su resultado acompañado de un error, que es la forma idiomática en que Go maneja los fallos sin necesidad de excepciones. Conocerás `defer`, una palabra mágica que programa una llamada para que se ejecute al final de la función pase lo que pase, perfecta para cerrar archivos y liberar recursos. Tocaras `panic` y `recover`, el mecanismo de emergencia para situaciones realmente excepcionales. Y dominarás la recursión, esa técnica elegante donde una función se llama a sí misma para resolver problemas que se parecen a versiones más pequeñas de sí mismos.

La segunda mitad de la unidad sube un nivel de abstracción: los paquetes. Si las funciones son máquinas individuales, un paquete es la fábrica completa: un conjunto de funciones y tipos relacionados que viven juntos en uno o varios archivos y se comparten bajo un mismo nombre. Go está construido sobre paquetes: cada vez que escribes `import "fmt"` estás trayendo un

paquete de la biblioteca estandar. Pero lo interesante es que tu también puedes crear tus propios paquetes, y de hecho debes hacerlo en cuanto tus programas crezcan. Aprenderas las reglas de visibilidad, ese detalle tan gofer donde una letra mayuscula al inicio de un nombre significa que es publico y una minuscula significa que es privado. Aprenderas a inicializar un modulo con `go mod init`, a entender que son los archivos `go.mod` y `go.sum` que aparecen en tus proyectos, y a organizar tu codigo en carpetas de forma profesional. También conoceras la funcion especial `init`, que se ejecuta sola antes que `main` y sirve para preparar el terreno.

Hay una razon profunda por la que esta unidad viene justo después de operadores y control de flujo. Las funciones no son un tema aislado: son el pegamento que une todo lo anterior. Dentro de una funcion usas operadores para calcular, `if` para decidir y `for` para repetir. Y a partir de esta unidad, casi todo lo que aprendas vivira dentro de funciones y paquetes: los metodos de las estructuras, las gorutinas de la concurrencia, los manejadores de un servidor web. Dominar funciones y paquetes no es opcional, es el pasaporte al resto del lenguaje. Asi que estudia esta unidad con calma y practica mucho: escribe funciones pequenas para todo, inventa tus propios paquetes de juguete y acostumbra a pensar en terminos de que recibe, que hace y que devuelve cada pieza de tu programa.

Un consejo practico antes de empezar: a partir de ahora tus programas van a tener varios archivos y carpetas, asi que necesitas un modulo de Go para que todo funcione. No te preocupes, es muy sencillo y lo veremos paso a paso en la seccion de paquetes. Por lo pronto, para los ejemplos de funciones que viven en un solo archivo, basta con lo que ya conoces: un archivo `.go` con `package main`, la funcion `main` como punto de entrada y `go run` para ejecutarlo. Cuando llegemos a los paquetes crearemos la estructura completa juntos.

Explicación detallada

4.1 Declarar funciones: la sintaxis fundamental

Declarar una funcion en Go sigue un patron fijo que vas a escribir miles de veces, asi que vale la pena desmenuzarlo con calma hasta que te salga de forma natural. La palabra clave `func` anuncia que viene una funcion, después va el nombre, después los parametros entre parentesis con su tipo, después el tipo de retorno y finalmente el cuerpo entre llaves. Un ejemplo minimo: `func sumar(a int, b int) int`. Se lee asi: la funcion `sumar` recibe dos parametros enteros llamados `a` y `b`, y devuelve un entero. Dentro del cuerpo, la palabra `return` entrega el resultado y termina la funcion. Si dos parametros seguidos comparten el mismo tipo, puedes abreviar: `func sumar(a, b int) int` es exactamente lo mismo y es la forma mas comun de escribirlo.

El nombre de una función debe ser descriptivo y seguir la convención de Go: se usa camelCase, empezando con minúscula para funciones privadas y con mayúscula para las que quieres exportar desde un paquete, como veras más adelante. Un buen nombre dice que hace la función sin que tengas que leer su interior: `calcularPromedio` es mejor que `calc`, `esNumeroPar` es mejor que `check`. Los parámetros son las variables de entrada: existen solo dentro de la función y reciben una copia de los valores que les pasas al llamar. Esto último es importante: en Go, por defecto, los parámetros se pasan por valor, lo que significa que si la función modifica un parámetro, la variable original que pasaste no se ve afectada. Es como darle una fotocopia de un documento a alguien: puede rayar la copia todo lo que quiera, tu original sigue intacto. Más adelante, cuando veas apuntadores, aprenderás como pasar el documento original, pero por ahora quedate con esta idea: las funciones no pueden danar tus variables por accidente, lo cual es una gran garantía de seguridad.

Toda función en Go devuelve algo o no devuelve nada. Si no devuelve nada, simplemente omite el tipo de retorno: `func saludar(nombre string)`. Si devuelve un valor, lo declaras después de los parámetros y usas `return` para entregarlo. Una regla estricta del compilador: si declaras un tipo de retorno, todos los caminos de la función deben terminar en un `return` con un valor de ese tipo. Go no te deja olvidar el `return` en una rama de un `if`, por ejemplo. Y un detalle de estilo: en Go las llaves de apertura van en la misma línea que la declaración, no en la línea siguiente como en otros lenguajes. El compilador de hecho te obliga a hacerlo así, porque el punto y coma implícito que Go inserta al final de cada línea rompería el código si pusieras la llave abajo. No luches contra esto: es una de esas decisiones que hacen que todo el código Go del mundo se vea igual, y cuando leas código ajeno lo agradecerás.

```

package main

import "fmt"

// Funcion que recibe dos enteros y devuelve un entero.
// Los parametros se pasan por valor: son copias.
func sumar(a, b int) int {
    resultado := a + b
    return resultado
}

// Funcion sin valor de retorno: solo hace algo.
func saludar(nombre string) {
    fmt.Println("Hola,", nombre, "bienvenido a Lucas EGA Academy")
}

func main() {
    total := sumar(15, 27)
    fmt.Println("La suma es:", total) // 42

    saludar("Esteban")

    // Los argumentos originales no cambian: se paso una copia.
    x := 10
    fmt.Println("Doble de x:", sumar(x, x)) // 20
    fmt.Println("x sigue valiendo:", x)    // 10
}

```

Un error muy comun de principiantes es declarar una funcion y olvidarse de llamarla, o llamarla sin usar su resultado cuando el resultado era lo importante. Otro error tipico es confundir parametros con argumentos: los parametros son las variables en la declaracion (a y b), los argumentos son los valores concretos que pasas al llamar (15 y 27). La distincion suena academica, pero te ayuda a comunicarte con precision cuando pidas ayuda o leas documentacion. Como buena practica, mantien tus funciones cortas y enfocadas: si una funcion hace mas de una cosa, probablemente deberia ser dos funciones. Y dale a cada funcion una sola responsabilidad clara, como una herramienta de un taller: el martillo clava, el desarmador aprieta, y ninguno intenta hacer el trabajo del otro.

4.2 Múltiples valores de retorno: el patrón value, err

Aquí llegamos a una de las características más distintivas y queridas de Go: las funciones pueden devolver más de un valor. En la mayoría de los lenguajes, una función devuelve una sola cosa y si algo sale mal tienes que lanzar una excepción o inventar códigos de error. Go tomó otro camino: una función que puede fallar devuelve dos valores, el resultado y un error. Si todo salió bien, el error es nil, que es la forma de Go de decir nada o nulo. Si algo salió mal, el resultado suele ser el valor cero de su tipo y el error describe qué pasó. Este patrón es tan central en Go que tiene nombre propio entre los programadores: el patrón value, err, y lo vas a ver en cada esquina del lenguaje y de sus bibliotecas.

La sintaxis es directa: `func dividir(a, b float64) (float64, error)`. Los tipos de retorno van entre parentesis, separados por coma. Al llamar a la función, recibes ambos valores: resultado, err := `dividir(10, 2)`. Y aquí viene la costumbre idiomática que debes adoptar desde el primer día: inmediatamente después de la llamada, revisas si err es diferente de nil, y si lo es, manejas el error y sales. Este chequeo pegado a la llamada es lo que hace que el manejo de errores en Go sea tan legible: el camino feliz del programa avanza hacia abajo sin interrupciones, y los problemas se atienden en el momento, sin sorpresas escondidas. Verás este patrón literalmente miles de veces en código Go profesional, así que acostúmbrate a escribirlo hasta que te salga sin pensar.

```

package main

import (
    "errors"
    "fmt"
)

// Dividir devuelve el cociente y un error si el divisor es cero.
func dividir(a, b float64) (float64, error) {
    if b == 0 {
        return 0, errors.New("no se puede dividir entre cero")
    }
    return a / b, nil
}

func main() {
    // Caso exitoso: err es nil.
    resultado, err := dividir(10, 2)
    if err != nil {
        fmt.Println("Error:", err)
        return
    }
    fmt.Println("Resultado:", resultado) // 5

    // Caso con error: se atiende de inmediato.
    resultado2, err2 := dividir(10, 0)
    if err2 != nil {
        fmt.Println("Error:", err2) // no se puede dividir entre cero
        return
    }
    fmt.Println("Resultado:", resultado2) // nunca se ejecuta
}

```

El paquete `errors`, que importamos arriba, sirve para crear errores simples con `errors.New`. Mas adelante conoceras `fmt.Errorf`, que permite crear errores con formato, como `errors.New` pero con placeholders. Un detalle importante: cuando una funcion devuelve multiples valores y solo te interesa uno, puedes ignorar los demas con el identificador vacio `_`: `resultado, _ := dividir(10, 2)`. Usalo con moderacion: ignorar un error es una decision que debe ser consciente, no un habito. Como buena practica, nunca dejes un error sin revisar a menos que tengas una razon explicita y documentada. Los errores ignorados en silencio son la fuente de esos bugs misteriosos que aparecen a las tres de la manana en produccion.

Vale la pena comparar este enfoque con las excepciones de otros lenguajes para que entiendas la filosofía. Las excepciones interrumpen el flujo normal y saltan a un manejador lejano, lo que hace difícil saber, leyendo una llamada, que cosas pueden salir mal. En Go, la firma de la función te lo dice en la cara: devuelve (float64, error), así que sabes que puede fallar y el compilador te obliga a recibir ese error. No puedes fingir que no existe. Esa explicitud es deliberada y es parte de lo que hace a los programas Go tan robustos: los errores se manejan donde ocurren, de forma visible y predecible.

4.3 Parámetros y retornos con nombre: documentar desde la firma

Go te permite ponerle nombre no solo a los parámetros de entrada, sino también a los valores de retorno. Cuando escribes `func dividir(a, b float64) (cociente float64, err error)`, esos nombres `cociente` y `err` no son decoración: se convierten en variables locales de la función, inicializadas con el valor cero de su tipo, y existen durante todo el cuerpo. La ventaja inmediata es documentación viva: quien lee la firma entiende que devuelve la función sin tener que adivinar que significa cada valor. Compara `func buscar(texto string) (int, bool)` con `func buscar(texto string) (posicion int, encontrado bool)`: la segunda firma se explica sola. En funciones con múltiples retornos, que son comunes en Go, esta claridad vale oro.

Los retornos con nombre habilitan además el llamado retorno desnudo o `naked return`: puedes escribir simplemente `return` sin valores, y Go devuelve automáticamente el contenido actual de las variables de retorno. Esto brilla en funciones con varias salidas tempranas por error, donde cada rama solo tiene que asignar las variables y hacer `return`. Pero cuidado: es un arma de doble filo. En funciones largas, un `return` desnudo al final obliga al lector a desplazarse hacia arriba para averiguar que se está devolviendo, lo cual dana la legibilidad. La regla práctica de la comunidad Go es clara: usa retornos con nombre para documentar la firma, pero preferelos en funciones cortas, y en funciones largas escribe los valores explícitamente en el `return` para que cada salida sea evidente. Como en tantas cosas de Go, la herramienta existe para los casos donde ayuda, no para usarla en todas partes.

```

package main

import (
    "errors"
    "fmt"
    "strings"
)

// Los retornos con nombre documentan la firma y permiten naked return.
func dividirConNombre(a, b float64) (cociente float64, err error) {
    if b == 0 {
        err = errors.New("no se puede dividir entre cero")
        return // devuelve (0, err): cociente vale cero por defecto
    }
    cociente = a / b
    return // devuelve (cociente, nil)
}

// Otro ejemplo: partir un nombre completo en nombre y apellido.
func partirNombre(completo string) (nombre string, apellido string) {
    partes := strings.Split(completo, " ")
    nombre = partes[0]
    if len(partes) > 1 {
        apellido = partes[1]
    }
    return // apellido queda "" si no habia apellido
}

func main() {
    c, err := dividirConNombre(20, 4)
    fmt.Println("Cociente:", c, "Error:", err) // 5 <nil>

    n, a := partirNombre("Ada Lovelace")
    fmt.Println("Nombre:", n, "| Apellido:", a)

    n2, a2 := partirNombre("Platon")
    fmt.Println("Nombre:", n2, "| Apellido:", a2) // Apellido vacio
}

```

Un error típico con retornos nombrados es la sombra de variables: si dentro de la función declaras una variable nueva con `:=` usando el mismo nombre que el retorno, creas una variable distinta y el retorno nombrado queda intacto con su valor cero, así que la función devuelve algo inesperado. Por ejemplo, si el retorno se llama `resultado` y dentro escribes `resultado := calculo()`, ese `:=` crea una variable local nueva y el retorno nombrado sigue valiendo cero. La

solucion es usar `=` simple para asignar al retorno nombrado, no `:=`. El compilador no siempre te avisa de este problema, así que mantente alerta cuando combines ambas características.

4.4 Funciones variádicas: acepta cuantos argumentos quieras

Hay funciones a las que no les puedes fijar el número de argumentos de antemano. Piensa en una función `sumar`: a veces quieres sumar dos números, a veces cinco, a veces veinte. En lugar de escribir diez versiones distintas, Go te ofrece las funciones variádicas, que aceptan un número variable de argumentos usando tres puntos suspensivos antes del tipo: `func sumar(numeros ...int) int`. Esos tres puntos son la magia: le dicen a Go empaqueta todos los argumentos extra en un slice de enteros. Dentro de la función, `numeros` se comporta como un slice normal: puedes recorrerlo con `for-range`, saber su longitud con `len` y acceder a sus elementos por índice. Y al llamar, pasas los valores separados por comas: `sumar(1, 2, 3, 4, 5)`. Incluso puedes llamarla sin argumentos, en cuyo caso el slice llega vacío.

Hay dos detalles que debes dominar. Primero, el parámetro variádico debe ser el último de la lista: `func reporte(titulo string, valores ...float64)` es válido, pero `func datos(...int, etiqueta string)` no compila. Tiene lógica: si hubiera algo después, Go no sabría donde terminan los argumentos variables. Segundo, si ya tienes un slice y quieres pasarlo a una función variádica, debes desempaquetarlo con tres puntos al llamar: `sumar(miSlice...)`. Sin esos puntos, Go intentaría pasar el slice completo como un solo argumento y el compilador se quejaría de tipos. Este patrón de desempaquetar aparece constantemente cuando conectas funciones entre sí, así que practícalo hasta que te salga natural.

```

package main

import "fmt"

// Suma cualquier cantidad de enteros.
func sumar(numeros ...int) int {
    total := 0
    for _, n := range numeros {
        total += n
    }
    return total
}

// El parametro variadico va al final; antes puede haber parametros normales.
func promediar(etiqueta string, valores ...float64) {
    if len(valores) == 0 {
        fmt.Println(etiqueta, ": sin datos")
        return
    }
    suma := 0.0
    for _, v := range valores {
        suma += v
    }
    fmt.Printf("%s: promedio = %.2f\n", etiqueta, suma/float64(len(valores)))
}

func main() {
    fmt.Println(sumar(1, 2, 3))           // 6
    fmt.Println(sumar(10, 20, 30, 40)) // 100
    fmt.Println(sumar())                 // 0: sin argumentos

    // Desempaquetar un slice con ...
    calificaciones := []int{9, 8, 10, 7}
    fmt.Println("Total calificaciones:", sumar(calificaciones...))

    promediar("Matematicas", 9.5, 8.0, 10.0)
    promediar("Sin datos")
}

```

Las funciones variadicadas son perfectas para utilidades como sumas, promedios, concatenaciones o constructores de mensajes, y de hecho la funcion mas famosa de Go, `fmt.Println`, es variadica: por eso puedes pasarle cuantos valores quieras. Como buena practica, usalas cuando el numero de argumentos sea genuinamente variable y todos compartan el mismo tipo y significado. Si los parametros tienen significados distintos, mejor

declararlos por separado con nombres claros: una funcion crearUsuario(nombre string, edad int, correos ...string) comunica mucho mas que una lista plana de strings donde nadie sabe cual es cual.

4.5 Funciones como valores: el poder de primera clase

En Go, las funciones son ciudadanas de primera clase, una frase elegante que significa algo muy practico: puedes tratar una funcion como cualquier otro valor. Puedes guardarla en una variable, pasarla como argumento a otra funcion y devolverla como resultado. Para guardar una funcion en una variable, declaras la variable con el tipo de la firma: `var operacion func(int, int) int`, y luego le asignas cualquier funcion compatible: `operacion = sumar`. A partir de ese momento, llamar `operacion(3, 4)` es lo mismo que llamar `sumar(3, 4)`. Esto abre la puerta a un estilo de programacion muy expresivo donde el comportamiento se vuelve un dato mas que puedes elegir, combinar y pasar de un lado a otro.

El uso mas comun de esta capacidad es pasar funciones como argumentos, lo que permite escribir funciones de orden superior: funciones que reciben otras funciones y las aplican. Imagina que quieres procesar una lista de numeros, pero a veces quieres duplicarlos, a veces triplicarlos y a veces calcular su cuadrado. En lugar de escribir tres bucles casi identicos, escribes una sola funcion transformar que recibe la lista y la operacion a aplicar: `func transformar(valores []int, op func(int) int) []int`. Luego la llamas con distintas operaciones: `transformar(datos, duplicar)`, `transformar(datos, alCuadrado)`. El bucle se escribe una vez y el comportamiento varia según la funcion que inyectes. Este patron, conocido como estrategia, aparece en ordenamientos personalizados, filtros, validaciones y en casi cualquier biblioteca seria de Go.

```
package main

import "fmt"

func sumar(a, b int) int      { return a + b }
func restar(a, b int) int     { return a - b }
func multiplicar(a, b int) int { return a * b }

// Recibe una funcion como argumento y la aplica.
func calcular(a, b int, operacion func(int, int) int) int {
    return operacion(a, b)
}

// Aplica una operacion a cada elemento de un slice.
func transformar(valores []int, op func(int) int) []int {
    resultado := make([]int, len(valores))
```

```

for i, v := range valores {
    resultado[i] = op(v)
}
return resultado
}

func duplicar(n int) int { return n * 2 }

func main() {
    // Guardar funciones en variables y elegir según el caso.
    var op func(int, int) int
    seleccion := "multiplicar"
    switch seleccion {
    case "sumar":
        op = sumar
    case "restar":
        op = restar
    default:
        op = multiplicar
    }
    fmt.Println("Resultado:", calcular(6, 7, op)) // 42

    // Pasar funciones como argumentos.
    datos := []int{1, 2, 3, 4, 5}
    fmt.Println("Duplicados:", transformar(datos, duplicar))
    fmt.Println("Al cuadrado:", transformar(datos, func(n int) int {
        return n * n
    })))
}

```

Fijate en el ultimo ejemplo: en lugar de declarar una funcion con nombre para el cuadrado, pasamos una funcion anonima directamente como argumento. Esas funciones sin nombre son el tema de la siguiente seccion, y combinadas con esta capacidad de pasar funciones como valores, forman una de las herramientas mas elegantes del lenguaje. Como buena practica, cuando una funcion recibe otra funcion como parametro, dale al parametro un nombre que describa su rol (operacion, filtro, validar) y documenta que se espera de ella: que recibe, que devuelve y que garantias debe cumplir. Tu codigo se vuelve un sistema de piezas intercambiables donde cada pieza tiene un contrato claro.

4.6 Funciones anónimas y closures: funciones con memoria

Una función anónima es exactamente lo que su nombre dice: una función sin nombre, que defines en el lugar donde la necesitas. La sintaxis es la misma que ya conoces, solo que sin el identificador: `func(x int) int { return x * 2 }`. Lo interesante no es la falta de nombre, sino donde puedes usarlas: puedes asignarlas a variables, pasarlas como argumentos como acabas de ver, e incluso ejecutarlas inmediatamente después de definir las, envolviéndolas entre parentesis: `func() { fmt.Println("hola") }()`. Las funciones anónimas brillan para lógica pequeña y de un solo uso: un filtro rápido, una transformación puntual, un callback. Si la lógica crece o se reutiliza, dale un nombre y declárala como función normal: el nombre es documentación.

Pero la verdadera magia aparece cuando una función anónima captura variables del ámbito que la rodea. Eso es un closure: una función que recuerda el entorno donde nació, incluso después de que ese entorno haya terminado. El ejemplo clásico es el contador: una función `crearContador` que declara una variable local `cuenta` en cero y devuelve una función anónima que incrementa y devuelve esa cuenta. Cada vez que llamas a la función devuelta, la variable `cuenta` sigue viva, guardada dentro del closure, aunque `crearContador` ya terminó hace mucho. Es como si la función anónima llevara consigo una mochila con las variables que necesita. Y lo más potente: cada llamada a `crearContador` produce un contador independiente, con su propia mochila. Así puedes tener `c1` y `c2` contando por separado sin interferirse.

```

package main

import "fmt"

// Devuelve una funcion que cuenta: cada llamada suma 1.
// La variable "cuenta" vive dentro del closure.
func crearContador() func() int {
    cuenta := 0
    return func() int {
        cuenta++
        return cuenta
    }
}

// Otro closure util: acumulador con valor inicial configurable.
func crearAcumulador(inicial int) func(int) int {
    total := inicial
    return func(cantidad int) int {
        total += cantidad
        return total
    }
}

func main() {
    contadorA := crearContador()
    contadorB := crearContador()

    fmt.Println(contadorA()) // 1
    fmt.Println(contadorA()) // 2
    fmt.Println(contadorA()) // 3
    fmt.Println(contadorB()) // 1: independiente del contador A

    puntos := crearAcumulador(100)
    fmt.Println(puntos(25)) // 125
    fmt.Println(puntos(25)) // 150: recuerda el total anterior

    // Funcion anonima ejecutada de inmediato (IIFE en otros lenguajes).
    func() {
        fmt.Println("Me ejecuto una sola vez, al definirme")
    }()
}

```

Hay una trampa historica con closures y bucles que debes conocer, aunque las versiones recientes de Go la suavizaron. Antes de Go 1.22, la variable de un for se reutilizaba en cada

iteracion, asi que si creabas closures dentro del bucle que capturaban esa variable, todos terminaban viendo el ultimo valor. Desde Go 1.22, cada iteracion tiene su propia variable y el problema desaparecio en la mayoria de los casos. Como este libro apunta a Go 1.21 en adelante, menciono el cambio para que no te confundas si lees codigo o articulos viejos que copian la variable manualmente dentro del bucle: era la solucion a ese problema, hoy ya no es necesaria, pero la veras en codigo heredado. Como buena practica, usa closures para encapsular estado pequeno y bien definido, como contadores, acumuladores o configuraciones, y evita que un closure modifique variables capturadas de formas sorprendentes: la mochila es poderosa, usala con responsabilidad.

4.7 defer: programa la limpieza y olvídate

Hay una categoria de tareas en programacion que son faciles de olvidar y caras cuando se olvidan: cerrar un archivo que abriste, liberar una conexion a la base de datos, soltar un bloqueo, registrar que una funcion termino. La palabra clave defer existe para que nunca mas se te olvide ninguna: cuando escribes defer algunaFuncion(), le dices a Go ejecuta esto al final, justo antes de que la funcion actual termine, pase lo que pase. Es como dejar una nota en la puerta: salgas por donde salgas de la casa, la ves. El uso mas famoso es con archivos: abres el archivo, e inmediatamente escribes defer archivo.Close(). A partir de ese momento puedes olvidarte del cierre: ocurra lo que ocurra en el resto de la funcion, incluso si hay un return temprano o un panic, el archivo se cerrara.

Puedes registrar varios defer en la misma funcion, y aqui hay un detalle crucial: se ejecutan en orden inverso al que los registraste, como una pila LIFO (el ultimo en entrar es el primero en salir). Si escribes defer a(), defer b(), defer c(), se ejecutaran c, luego b, luego a. Esto tiene mucho sentido cuando piensas en recursos anidados: abres la conexion, luego abres la transaccion; al salir quieres cerrar primero la transaccion y después la conexion, que es exactamente el orden inverso. Otro detalle sutil pero importante: los argumentos de la llamada diferida se evaluan en el momento en que escribes el defer, no cuando se ejecuta. Si escribes defer fmt.Println(i) cuando i vale 1, y después i cambia a 2, al final se imprimira 1, porque el valor se capturo al registrar el defer. En cambio, si difieres un closure que lee la variable, vera el valor final, porque el closure se evalua hasta el momento de ejecutarse.

```

package main

import (
    "fmt"
    "os"
)

func ejemploOrden() {
    // Los defer se ejecutan en orden inverso (LIFO).
    defer fmt.Println("tercero: me registre primero, salgo al ultimo")
    defer fmt.Println("segundo")
    defer fmt.Println("primero: me registre al ultimo, salgo primero")
    fmt.Println("cuerpo de la funcion")
}

func escribirArchivo() error {
    // Abrir (o crear) el archivo.
    archivo, err := os.Create("saludo.txt")
    if err != nil {
        return err
    }
    // Programar el cierre INMEDIATAMENTE después de abrir.
    // Pase lo que pase abajo, el archivo se cierra.
    defer archivo.Close()

    _, err = archivo.WriteString("Hola desde Go\n")
    if err != nil {
        return err // el defer cierra el archivo antes de salir
    }

    fmt.Println("Archivo escrito correctamente")
    return nil // el defer cierra el archivo aqui también
}

func main() {
    ejemploOrden()
    if err := escribirArchivo(); err != nil {
        fmt.Println("Error:", err)
    }
}

```

Hay una trampa clasica con defer que debes conocer: usarlo dentro de un bucle largo. Cada defer se acumula hasta que la funcion termine, asi que si abres mil archivos en un for y difieres sus cierres, los mil quedaran abiertos hasta el final de la funcion, lo que puede agotar los

recursos del sistema. La solución es extraer el cuerpo del bucle a una función auxiliar: el `defer` dentro de esa función se ejecuta cuando la función auxiliar termina, o sea en cada vuelta. Como buena práctica, escribe el `defer` inmediatamente después de adquirir el recurso, en la línea siguiente si es posible: abrir y programar el cierre juntos forma una unidad visual que nadie romperá por accidente al editar el código después.

4.8 panic y recover: el botón de emergencia

La mayoría de los errores en Go se manejan con el patrón `value, err` que ya dominas: la función devuelve el error y tú decides qué hacer. Pero hay situaciones tan graves o tan inesperadas que no tiene sentido seguir intentando: un índice fuera de rango, una división entre cero con enteros, acceder a un mapa `nil` de cierta forma, o una condición que según tu diseño nunca debería ocurrir. Para esos casos existe `panic`: detiene la ejecución normal de inmediato, ejecuta todos los `defer` pendientes y termina el programa mostrando un mensaje y la pila de llamadas. Puedes provocar un pánico tú mismo con `panic("mensaje descriptivo")`, aunque debería ser raro.

Su contraparte es `recover`, una función que detiene un pánico en curso y permite que el programa continúe, pero con una restricción fundamental: `recover` solo funciona si se llama desde una función diferida con `defer`, mientras el pánico todavía se está propagando. El patrón completo es: al inicio de una función que quieres proteger, escribes `defer func() { if r := recover(); r != nil { ...manejar... } }()`. Si ocurre un pánico más abajo, los `defer` se ejecutan, tu función diferida lo intercepta con `recover`, y en lugar de morir el programa, puedes registrar el error, limpiar y continuar. Es el equivalente a la red de seguridad bajo el trapecista: no la usas en cada acto, pero cuando la necesitas, salva la función.

```

package main

import "fmt"

// divisionSegura convierte un posible panic en un error manejable.
func divisionSegura(a, b int) (resultado int, err error) {
    // La red de seguridad: debe ir al inicio con defer.
    defer func() {
        if r := recover(); r != nil {
            err = fmt.Errorf("panico recuperado: %v", r)
            resultado = 0
        }
    }()

    // Si b es cero, esta division provoca un panic automaticamente.
    resultado = a / b
    return resultado, nil
}

func main() {
    r, err := divisionSegura(10, 2)
    fmt.Println("10 / 2 =", r, "| error:", err)

    r2, err2 := divisionSegura(10, 0)
    fmt.Println("10 / 0 =", r2, "| error:", err2)
    fmt.Println("El programa sigue vivo despues del panic recuperado")
}

```

Ahora, la advertencia importante, y quiero ser muy claro: panic y recover NO son el mecanismo normal de manejo de errores en Go. En otros lenguajes las excepciones se usan para todo; en Go, el camino idiomático es devolver errores como valores. Reserva panic para lo verdaderamente excepcional: errores de programación que indican un bug, o fallos al iniciar el programa donde no hay forma sensata de continuar, como no poder leer el archivo de configuración sin el cual nada funciona. Y reserva recover para las fronteras de tu programa: un servidor web que no debe caerse porque un manejador fallo, una biblioteca que protege a sus usuarios de panicos internos. En el código de todos los días, si dudas entre devolver un error o hacer panic, devuelve el error. En la unidad 9 profundizaremos en el manejo de errores avanzado y veras con mas detalle cuando cada herramienta es la correcta.

4.9 Recursión: funciones que se llaman a sí mismas

La recursión es una de las ideas más elegantes de la programación: una función que resuelve un problema llamándose a sí misma con una versión más pequeña del mismo problema. Para que funcione necesita dos ingredientes: el caso base, que es la versión tan pequeña del problema que se resuelve directamente sin más llamadas, y el paso recursivo, que reduce el problema y combina los resultados. El ejemplo clásico es el factorial: $5! = 5 * 4 * 3 * 2 * 1$. Observa el patrón: $n! = n * (n-1)!$. El caso base es $0! = 1$, que se responde sin calcular nada. El paso recursivo dice: para calcular el factorial de n , calcula el factorial de $n-1$ y multiplícalo por n . La función se llama a sí misma con números cada vez más pequeños hasta llegar al caso base, y entonces los resultados se combinan de regreso hacia arriba.

Visualízalo como una pila de platos: cada llamada recursiva pone un plato encima con un problema más pequeño, hasta llegar al caso base que es el plato de la parte superior y se resuelve solo. Luego vas quitando platos de la parte superior hacia abajo, combinando resultados. Cada llamada tiene sus propias variables locales, independientes de las demás, así que no hay confusión entre los distintos niveles. Pero aquí está el peligro: si olvidas el caso base o el problema nunca se hace más pequeño, las llamadas se acumulan hasta agotar la pila de memoria y el programa muere con un desbordamiento de pila o stack overflow. Por eso, antes de escribir una función recursiva, define primero el caso base por escrito, como un contrato contigo mismo.

```

package main

import "fmt"

// Factorial recursivo:  $n! = n * (n-1)!$ , con caso base  $0! = 1$ .
func factorial(n int) int {
    if n <= 1 { // caso base: aqui se detiene la recursion
        return 1
    }
    return n * factorial(n-1) // paso recursivo: problema mas pequeno
}

// Fibonacci recursivo:  $fib(n) = fib(n-1) + fib(n-2)$ .
func fibonacci(n int) int {
    if n <= 1 {
        return n
    }
    return fibonacci(n-1) + fibonacci(n-2)
}

// Suma de digitos recursiva: 1234 ->  $1+2+3+4 = 10$ .
func sumaDigitos(n int) int {
    if n < 10 { // caso base: un solo digito se devuelve tal cual
        return n
    }
    return n%10 + sumaDigitos(n/10)
}

func main() {
    fmt.Println("5! =", factorial(5))           // 120
    fmt.Println("fib(10) =", fibonacci(10))     // 55
    fmt.Println("sumaDigitos(1234) =", sumaDigitos(1234)) // 10
}

```

La recursion brilla en problemas que tienen estructura recursiva natural: recorrer arboles de archivos, evaluar expresiones matematicas anidadas, resolver laberintos, implementar busquedas como el famoso quicksort. Pero ten presente su costo: cada llamada consume memoria de la pila y las soluciones ingenuas pueden repetir muchisimo trabajo, como el fibonacci de arriba, que recalcula los mismos valores una y otra vez y se vuelve lentisimo con numeros grandes. Como buena practica, usa recursion cuando el problema sea naturalmente recursivo y la profundidad sea moderada; si la profundidad puede ser enorme o el rendimiento es critico, considera la version iterativa con un bucle, que hace lo mismo con memoria constante. Saber ambas formas y elegir con criterio es lo que distingue a un programador maduro.

4.10 Paquetes: organiza tu código como profesional

Un paquete en Go es una coleccion de archivos fuente que viven en la misma carpeta, comparten el mismo nombre de paquete y trabajan juntos como una unidad. Es la forma de organizar el codigo cuando un solo archivo ya no basta. Ya has usado paquetes desde el primer dia: `fmt` para imprimir, `math` para matematicas, `strings` para texto, `errors` para crear errores. Todos forman parte de la biblioteca estandar, pero la idea poderosa es que tu puedes crear los tuyos con las mismas reglas. Un paquete bien disenado agrupa funciones y tipos relacionados bajo un nombre claro: un paquete geometria con funciones de areas y perimetros, un paquete usuarios con la logica de cuentas, un paquete pagos con la integracion de cobros.

Crear un paquete es sencillo: creas una carpeta con el nombre del paquete, y dentro pones uno o mas archivos `.go` que empiecen con `package nombredelpaquete`. Por ejemplo, una carpeta `matematicas` con un archivo `operaciones.go` que declare `package matematicas` y contenga tus funciones. Para usarlo desde tu programa principal, lo importas con la ruta del modulo mas la carpeta: `import "mimodulo/matematicas"`, y luego llamas a sus funciones con el nombre del paquete como prefijo: `matematicas.Sumar(3, 4)`. Ese prefijo es importante: evita colisiones de nombres y deja claro de donde viene cada funcion cuando lees el codigo.

Y aquí está la regla de visibilidad más famosa de Go, simple y genial: si un nombre empieza con mayúscula, es exportado, o sea público y usable desde otros paquetes; si empieza con minúscula, es privado y solo vive dentro de su propio paquete. Así, en tu paquete `matematicas`, la función `Sumar` con `S` mayúscula puede usarse desde `main`, pero una función auxiliar `sumarInterno` con `s` minúscula solo existe para uso interno del paquete. Lo mismo aplica a variables, constantes, tipos y campos de estructuras. No hay palabras clave como `public` o `private`: la mayúscula es todo el sistema. Esta convención obliga a diseñar una superficie pública clara: lo que expones con mayúscula es tu contrato con el mundo, lo demás son detalles de implementación que puedes cambiar libremente.

```
// Archivo: mimodulo/matematicas/operaciones.go
package matematicas

// Sumar es exportada: se puede usar desde otros paquetes.
func Sumar(a, b int) int {
    return a + b
}

// Multiplicar también es exportada.
func Multiplicar(a, b int) int {
    return a * b
}

// cuadrado es privada: solo se usa dentro de este paquete.
func cuadrado(n int) int {
    return n * n
}

// PotenciaAlCuadrado usa la función privada por dentro.
func PotenciaAlCuadrado(n int) int {
    return cuadrado(n)
}
```

```
// Archivo: mimodulo/main.go
package main

import (
    "fmt"

    "mimodulo/matematicas"
)

func main() {
    fmt.Println(matematicas.Sumar(3, 4))           // 7
    fmt.Println(matematicas.Multiplicar(3, 4))     // 12
    fmt.Println(matematicas.PotenciaAlCuadrado(5)) // 25
    // fmt.Println(matematicas.cuadrado(5)) // ERROR: no es exportada
}
```

Un error comun es poner archivos con distinto nombre de paquete en la misma carpeta: todos los .go de una carpeta deben declarar el mismo package, sin excepcion. Otro es intentar importar un paquete usando una ruta de archivo en lugar de la ruta del modulo: los imports en Go usan rutas logicas del modulo, no rutas del disco. Y un tercero es la importacion circular: si el paquete A importa al paquete B y B importa a A, Go lo prohíbe terminantemente, porque seria un callejon sin salida logico. Si te topas con un ciclo de imports, es una senal de que tu division de paquetes necesita replantearse: quizá el código compartido deba vivir en un tercer paquete que ambos importen.

4.11 Módulos, go.mod y organización de proyectos

Hasta ahora tus programas eran archivos sueltos que compilabas con go run. Eso funciona para juguetes, pero los proyectos reales necesitan una identidad, un manejo de dependencias y una estructura de carpetas. Para eso existen los modulos de Go. Un modulo es un conjunto de paquetes con un nombre unico y un archivo go.mod que lo describe. Crear uno es un solo comando: go mod init mimodulo, ejecutado en la carpeta raiz de tu proyecto. El nombre suele ser una ruta de repositorio como github.com/tuusuario/miproyecto, pero para practicar puedes usar un nombre simple como mimodulo. Ese nombre se convierte en el prefijo de todos tus imports internos, como viste en el ejemplo anterior con mimodulo/matematicas.

El comando genera el archivo go.mod, que es el corazon administrativo de tu proyecto. Contiene el nombre del modulo, la version de Go que usas y la lista de dependencias externas con sus versiones exactas. Cuando importas un paquete de terceros y compilas, Go descarga la dependencia, la registra en go.mod y guarda sus sumas de verificacion en go.sum. El archivo go.sum es tu garantia de integridad: contiene hashes criptograficos de cada

dependencia, así que si alguien altera un paquete en el servidor, Go lo detecta y se niega a compilar. Nunca edites `go.sum` a mano: lo gestiona la herramienta. Y ambos archivos, `go.mod` y `go.sum`, deben vivir en tu control de versiones junto con tu código: son parte del proyecto.

```
miproyecto/  
  go.mod          # nombre del modulo y dependencias  
  go.sum          # hashes de verificacion de dependencias  
  main.go         # paquete main: punto de entrada  
  matematicas/  
    operaciones.go # package matematicas  
  usuarios/  
    usuarios.go   # package usuarios
```

Esta estructura es el patrón estándar para proyectos pequeños y medianos: la raíz con el `main` y el `go.mod`, y una carpeta por paquete con un nombre corto y descriptivo. Los comandos que usarás a diario son pocos: `go mod init` para crear el módulo, `go mod tidy` para agregar las dependencias que uses y quitar las que sobren, `go build` para compilar y `go run .` para ejecutar el programa completo desde la raíz. Con el punto en `go run .` le dices a Go ejecuta el paquete de la carpeta actual, incluyendo todos sus archivos. Acostúmbrate a esta organización desde tus primeros proyectos: es la misma que usan los equipos profesionales y te prepara para colaborar sin fricción.

Nos falta una pieza especial: la función `init`. En cualquier archivo de un paquete puedes declarar `func init()`, sin parámetros ni retornos, y Go la ejecutará automáticamente antes que `main`, sin que nadie la llame. Si hay varias funciones `init` en un paquete, se ejecutan en el orden en que aparecen los archivos; si varios paquetes tienen `init`, primero se inicializan las dependencias y al final el paquete `main`. Su propósito es preparar el terreno: inicializar variables globales complejas, registrar controladores, validar configuración. Pero úsala con moderación: un `init` hace trabajo invisible que el lector no ve al leer `main`, lo que dificulta entender que pasa y en qué orden. La comunidad Go prefiere inicialización explícita mediante funciones bien nombradas llamadas desde `main`, y reserva `init` para casos genuinos como registros automáticos. Como regla: si puedes hacerlo con una llamada explícita en `main`, hazlo así; deja `init` para lo que realmente necesita ocurrir solo, antes que todo.

Ejemplos paso a paso

Ejemplo 1: Funciones básicas que saludan y calculan

Problema: crear una función que salude por nombre y otra que calcule el área de un círculo dado su radio. Este ejemplo muestra la anatomía completa: parámetros con tipo, valor de retorno y llamada.

```
package main

import (
    "fmt"
    "math"
)

// Recibe un string, no devuelve nada.
func saludar(nombre string) {
    fmt.Println("Hola", nombre, ", bienvenido a Lucas EGA Academy")
}

// Recibe un float64 y devuelve un float64.
func areaCirculo(radio float64) float64 {
    return math.Pi * radio * radio
}

func main() {
    saludar("Esteban")

    // El resultado se puede guardar, imprimir o usar en otra operación.
    area := areaCirculo(5)
    fmt.Printf("Área de un círculo de radio 5: %.2f\n", area)

    fmt.Printf("Doble del área: %.2f\n", areaCirculo(5)*2)
}
```

Ejemplo 2: División segura con el patrón value, err

Problema: escribir una función dividir que no provoque un pánico al dividir entre cero, sino que devuelva un error descriptivo. El main debe manejar ambos casos con el patrón idiomático.

```

package main

import (
    "errors"
    "fmt"
)

// Devuelve el cociente y un error si algo sale mal.
func dividir(a, b float64) (float64, error) {
    if b == 0 {
        return 0, errors.New("no se puede dividir entre cero")
    }
    return a / b, nil
}

func main() {
    // Patron idiomático: llamar, revisar el error de inmediato.
    if resultado, err := dividir(20, 4); err != nil {
        fmt.Println("Error:", err)
    } else {
        fmt.Println("20 / 4 =", resultado)
    }

    if resultado, err := dividir(20, 0); err != nil {
        fmt.Println("Error:", err) // no se puede dividir entre cero
    } else {
        fmt.Println("20 / 0 =", resultado) // no se ejecuta
    }
}

```

Ejemplo 3: Estadísticas con retornos nombrados

Problema: escribir una función que reciba una lista de calificaciones y devuelva el mínimo, el máximo y el promedio, con retornos nombrados que documenten la firma.

```

package main

import (
    "errors"
    "fmt"
)

// Los nombres minimo, maximo y promedio documentan que devuelve la funcion.
func estadisticas(califs []float64) (minimo float64, maximo float64, promedio float64, err error) {
    if len(califs) == 0 {
        err = errors.New("lista vacia")
        return // devuelve ceros y el error
    }
    minimo, maximo = califs[0], califs[0]
    suma := 0.0
    for _, c := range califs {
        if c < minimo {
            minimo = c
        }
        if c > maximo {
            maximo = c
        }
        suma += c
    }
    promedio = suma / float64(len(califs))
    return // naked return: devuelve minimo, maximo, promedio, nil
}

func main() {
    min, max, prom, err := estadisticas([]float64{8.5, 9.0, 7.5, 10.0})
    if err != nil {
        fmt.Println("Error:", err)
        return
    }
    fmt.Printf("Min: %.1f, Max: %.1f, Promedio: %.2f\n", min, max, prom)
}

```

Ejemplo 4: Promedio flexible con función variádica

Problema: crear una funcion promedio que acepte cualquier cantidad de calificaciones, y una funcion etiquetar que combine un titulo con valores variables. Muestra también como pasar un slice existente.

```

package main

import "fmt"

// Acepta cualquier cantidad de float64.
func promedio(valores ...float64) float64 {
    if len(valores) == 0 {
        return 0
    }
    suma := 0.0
    for _, v := range valores {
        suma += v
    }
    return suma / float64(len(valores))
}

func main() {
    fmt.Printf("Promedio de 3: %.2f\n", promedio(9.0, 8.5, 10.0))
    fmt.Printf("Promedio de 5: %.2f\n", promedio(7.0, 8.0, 9.0, 6.5, 10.0))
    fmt.Printf("Sin datos: %.2f\n", promedio())

    // Pasar un slice existente: desempaquetar con ...
    parciales := []float64{8.0, 9.5, 7.0}
    fmt.Printf("Promedio del slice: %.2f\n", promedio(parciales...))
}

```

Ejemplo 5: Calculadora con funciones como valores

Problema: implementar una calculadora donde las operaciones se guardan en un mapa que asocia nombres con funciones. El usuario elige la operacion por nombre y el programa la aplica. Este ejemplo muestra el poder de tratar funciones como datos.

```

package main

import (
    "errors"
    "fmt"
)

func main() {
    // Mapa de nombre -> funcion. Cada valor es una funcion compatible.
    operaciones := map[string]func(float64, float64) (float64, error){
        "sumar": func(a, b float64) (float64, error) {
            return a + b, nil
        }
    }
}

```

```

    },
    "restar": func(a, b float64) (float64, error) {
        return a - b, nil
    },
    "multiplicar": func(a, b float64) (float64, error) {
        return a * b, nil
    },
    "dividir": func(a, b float64) (float64, error) {
        if b == 0 {
            return 0, errors.New("division entre cero")
        }
        return a / b, nil
    },
}

elegida := "multiplicar"
op, existe := operaciones[elegida]
if !existe {
    fmt.Println("Operacion desconocida")
    return
}
resultado, err := op(6, 7)
if err != nil {
    fmt.Println("Error:", err)
    return
}
fmt.Printf("%s(6, 7) = %.2f\n", elegida, resultado) // 42

// Agregar una operacion nueva es trivial: solo se registra en el mapa.
operaciones["potencia"] = func(a, b float64) (float64, error) {
    resultado := 1.0
    for i := 0; i < int(b); i++ {
        resultado *= a
    }
    return resultado, nil
}
r2, _ := operaciones["potencia"](2, 10)
fmt.Println("potencia(2, 10) =", r2) // 1024
}

```

Agregar operaciones nuevas no requiere tocar la logica principal: el mapa es extensible. Este es el patron estrategia en su forma mas simple.

Ejemplo 6: Generador de folios con closure

Problema: un sistema necesita folios consecutivos con prefijo, como FAC-0001, FAC-0002. Cada serie (facturas, tickets) debe llevar su propio contador independiente. Un closure por serie es la solución natural.

```
package main

import "fmt"

// Devuelve una función que genera folios consecutivos con el prefijo dado.
// El contador vive dentro del closure: nadie más puede alterarlo.
func crearGeneradorFolios(prefijo string) func() string {
    contador := 0
    return func() string {
        contador++
        return fmt.Sprintf("%s-%04d", prefijo, contador)
    }
}

func main() {
    folioFactura := crearGeneradorFolios("FAC")
    folioTicket := crearGeneradorFolios("TCK")

    fmt.Println(folioFactura()) // FAC-0001
    fmt.Println(folioFactura()) // FAC-0002
    fmt.Println(folioTicket())  // TCK-0001: contador independiente
    fmt.Println(folioFactura()) // FAC-0003
}
```

Cada generador encapsula su propio estado sin variables globales y sin estructuras: el closure es el objeto más pequeño posible que recuerda cosas.

Ejemplo 7: Copiar un archivo con defer

Problema: escribir una función que copie el contenido de un archivo a otro, asegurando que ambos archivos se cierren correctamente incluso si algo falla a la mitad. Los `defer` consecutivos garantizan la limpieza.

```

package main

import (
    "fmt"
    "io"
    "os"
)

func copiarArchivo(origen, destino string) (err error) {
    // Convertir panicos inesperados en error seria excesivo aqui;
    // los defer se encargan de la limpieza.
    entrada, err := os.Open(origen)
    if err != nil {
        return err
    }
    defer entrada.Close() // se cierra al final, pase lo que pase

    salida, err := os.Create(destino)
    if err != nil {
        return err // el primer defer cierra 'entrada' antes de salir
    }
    defer salida.Close() // LIFO: se cierra antes que 'entrada'

    _, err = io.Copy(salida, entrada)
    return err
}

func main() {
    // Crear un archivo de prueba.
    os.WriteFile("origen.txt", []byte("Hola desde Go\nSegunda linea\n"), 0644)

    if err := copiarArchivo("origen.txt", "copia.txt"); err != nil {
        fmt.Println("Error al copiar:", err)
        return
    }
    fmt.Println("Archivo copiado correctamente")

    datos, _ := os.ReadFile("copia.txt")
    fmt.Print(string(datos))
}

```

Observa el orden LIFO: salida se cierra antes que entrada, que es el orden correcto para recursos anidados. Y cada defer esta en la linea siguiente a la apertura del recurso: abrir y programar el cierre van juntos.

Ejemplo 8: Acceso seguro a un slice con panic y recover

Problema: crear una función obtener que devuelva el elemento en una posición, pero en lugar de dejar que un índice inválido mate el programa, lo convierta en un error controlado mediante recover.

```
package main

import "fmt"

// obtenerSeguro convierte el panic por índice fuera de rango en un error.
func obtenerSeguro(datos []string, indice int) (valor string, err error) {
    defer func() {
        if r := recover(); r != nil {
            err = fmt.Errorf("índice %d fuera de rango: %v", indice, r)
        }
    }()

    valor = datos[indice] // panic si el índice es inválido
    return valor, nil
}

func main() {
    nombres := []string{"Ana", "Beto", "Carla"}

    v, err := obtenerSeguro(nombres, 1)
    fmt.Println("Posición 1:", v, "| error:", err)

    v2, err2 := obtenerSeguro(nombres, 10)
    fmt.Println("Posición 10:", v2, "| error:", err2)
    fmt.Println("El programa sigue en pie")
}
```

Este patrón es útil en las fronteras del programa, pero recuerda la advertencia de la teoría: en código normal, mejor valida el índice con un if antes de acceder. El recover es la red de seguridad, no el camino principal.

Ejemplo 9: Potencia recursiva

Problema: calcular base elevada a exponente con recursión. Caso base: cualquier número elevado a 0 es 1. Paso recursivo: $\text{base}^{\text{exp}} = \text{base} * \text{base}^{(\text{exp}-1)}$.


```

package main

import "fmt"

// potencia recursiva: exp debe ser >= 0.
func potencia(base, exp int) int {
    if exp == 0 { // caso base
        return 1
    }
    return base * potencia(base, exp-1) // paso recursivo
}

func main() {
    fmt.Println("2^10 =", potencia(2, 10)) // 1024
    fmt.Println("3^4 =", potencia(3, 4))   // 81
    fmt.Println("5^0 =", potencia(5, 0))   // 1
}

```

Sigue la traza mental de `potencia(3, 2)`: llama a `potencia(3, 1)`, que llama a `potencia(3, 0)`, que devuelve 1; luego se combinan: $3 * 1 = 3$, y $3 * 3 = 9$. La pila de llamadas crece hacia abajo y los resultados suben de regreso.

Ejemplo 10: Mi primer paquete

Problema: crear el paquete `saludos` con una función exportada y una privada, y usarlo desde el programa principal. Este ejemplo muestra los dos archivos tal como vivirían en tu proyecto.

Estructura del proyecto:

```

miproyecto/
  go.mod
  main.go
  saludos/
    saludos.go

```

Archivo `go.mod` (creado con `go mod init miproyecto`):

```

module miproyecto

go 1.21

```

Archivo `saludos/saludos.go`:

```
// Paquete saludos: mensajes de bienvenida reutilizables.
package saludos

import "fmt"

// Bienvenida es exportada (mayuscula): se usa desde otros paquetes.
func Bienvenida(nombre string) string {
    return "Hola " + nombre + ", " + mensajeSecreto()
}

// mensajeSecreto es privada (minuscuala): solo vive en este paquete.
func mensajeSecreto() string {
    return "bienvenido a Lucas EGA Academy"
}

// Imprimir es otra funcion exportada de ejemplo.
func Imprimir(nombre string) {
    fmt.Println(Bienvenida(nombre))
}
```

Archivo main.go:

```
package main

import "miproyecto/saludos"

func main() {
    // Se accede con el nombre del paquete como prefijo.
    saludos.Imprimir("Esteban")
    // saludos.mensajeSecreto() // ERROR: no es exportada
}
```

Para probarlo: crea la carpeta, ejecuta `go mod init miproyecto` dentro de ella, guarda los archivos y corre `go run .` desde la raiz. Veras el mensaje completo, construido en parte por una funcion privada que el main ni siquiera puede ver.

Ejemplo 11: Fábrica de multiplicadores

Problema: crear una funcion que genere funciones multiplicadoras: le pides un factor y te devuelve una funcion que multiplica cualquier numero por ese factor. Asi obtienes duplicar, triplicar, etc., sin escribir cada una a mano. Es el patron fabrica aplicado a closures.

```

package main

import "fmt"

// Recibe un factor y devuelve una funcion que multiplica por ese factor.
// El factor queda capturado dentro del closure.
func crearMultiplicador(factor int) func(int) int {
    return func(n int) int {
        return n * factor
    }
}

func main() {
    duplicar := crearMultiplicador(2)
    triplicar := crearMultiplicador(3)
    porDiez := crearMultiplicador(10)

    fmt.Println("duplicar(7) =", duplicar(7))    // 14
    fmt.Println("triplicar(7) =", triplicar(7))  // 21
    fmt.Println("porDiez(7) =", porDiez(7))      // 70

    // Las fabricas también sirven para configurar comportamiento:
    // la misma logica, parametrizada una sola vez.
    numeros := []int{1, 2, 3, 4}
    for _, n := range numeros {
        fmt.Print(porDiez(n), " ") // 10 20 30 40
    }
    fmt.Println()
}

```

Este patron es la base de middlewares, decoradores y configuradores en Go profesional: una funcion externa recibe la configuracion y devuelve la funcion lista para usarse, con la configuracion ya capturada.

Ejemplo 12: defer que enriquece el error

Problema: una funcion con retorno nombrado err usa un defer para agregar contexto al error antes de devolverlo, por ejemplo registrando en que operacion fallo. Este es un patron real de codigo Go en produccion.

```

package main

import (
    "errors"
    "fmt"
)

// procesar devuelve un error con contexto agregado por el defer.
func procesar(datos string) (err error) {
    // El defer se ejecuta al salir y puede modificar el retorno nombrado.
    defer func() {
        if err != nil {
            err = fmt.Errorf("procesar(%q) fallo: %w", datos, err)
        }
    }()

    if datos == "" {
        return errors.New("datos vacios")
    }
    if len(datos) > 10 {
        return errors.New("datos demasiado largos")
    }
    fmt.Println("Procesado:", datos)
    return nil
}

func main() {
    if err := procesar(""); err != nil {
        fmt.Println("Error:", err)
    }
    if err := procesar("este texto es demasiado largo"); err != nil {
        fmt.Println("Error:", err)
    }
    procesar("hola")
}

```

El verbo %w de `fmt.Errorf` envuelve el error original conservando la cadena de errores, lo que permite inspeccionarla después con `errors.Is` y `errors.As`. Combinar retornos nombrados con `defer` de esta forma es idiomático, pero usalo solo donde aporte contexto real: un `defer` que no modifica nada es ruido.

Práctica

Ejercicio 1: Primo con error

Escribe la función `esPrimo(n int) (bool, error)` que devuelva un error si `n` es menor que 2 (no tiene sentido preguntar) y el resultado en caso contrario. El main debe probar con 2, 15, 29 y 1, manejando los errores con el patrón idiomático.

Solución explicada: la validación va primero como guard clause: si `n < 2`, devolvemos `false` y un error creado con `errors.New`. Después, el algoritmo prueba divisores hasta la raíz cuadrada. Cada llamada en el main sigue el patrón `valor, err := esPrimo(x)` seguido del `if err != nil` inmediato. Este ejercicio combina el patrón `value, err` con la lógica de primos de la unidad anterior.

```
package main

import (
    "errors"
    "fmt"
    "math"
)

func esPrimo(n int) (bool, error) {
    if n < 2 {
        return false, errors.New("n debe ser mayor o igual a 2")
    }
    limite := int(math.Sqrt(float64(n)))
    for i := 2; i <= limite; i++ {
        if n%i == 0 {
            return false, nil // no es primo, pero no es un error
        }
    }
    return true, nil
}

func main() {
    for _, n := range []int{2, 15, 29, 1} {
        primo, err := esPrimo(n)
        if err != nil {
            fmt.Printf("%d -> error: %v\n", n, err)
            continue
        }
        fmt.Printf("%d -> primo: %v\n", n, primo)
    }
}
```

Ejercicio 2: El mayor de todos (variádica)

Escribe la función `mayor(numeros ...int) int` que devuelva el mayor de los números recibidos. Si no recibe ninguno, debe devolver 0. Pruebala con llamadas directas y pasándole un slice desempquetado.

Solución explicada: si `len(numeros) == 0` devolvemos 0 como caso especial. Si no, inicializamos el máximo con el primer elemento y recorremos el resto comparando. Inicializar con el primer elemento (y no con 0) es importante: si todos los números son negativos, inicializar con 0 daría un resultado incorrecto. El desempquetado con `...` al pasar el slice es el detalle sintáctico que hay que practicar.

```
package main

import "fmt"

func mayor(numeros ...int) int {
    if len(numeros) == 0 {
        return 0
    }
    max := numeros[0]
    for _, n := range numeros[1:] {
        if n > max {
            max = n
        }
    }
    return max
}

func main() {
    fmt.Println(mayor(3, 7, 2, 9, 4)) // 9
    fmt.Println(mayor(-5, -2, -9))    // -2 (por eso no inicializamos con 0)
    fmt.Println(mayor())              // 0

    datos := []int{12, 45, 8, 33}
    fmt.Println(mayor(datos...)) // 45
}
```

Ejercicio 3: Generador de Fibonacci con closure

Escribe la función `crearFibonacci() func() int` que devuelva una función generadora: cada llamada entrega el siguiente número de la serie de Fibonacci (0, 1, 1, 2, 3, 5, 8...). El estado (los dos últimos números) debe vivir dentro del closure.

Solucion explicada: el closure captura dos variables, a y b, inicializadas en 0 y 1. En cada llamada, guardamos el valor actual de a para devolverlo, avanzamos $a = b$ y $b = \text{anterior} + b$. La secuencia de estados es: devuelve 0 ($a=0, b=1$), devuelve 1 ($a=1, b=1$), devuelve 1 ($a=1, b=2$), devuelve 2 ($a=2, b=3$)... Cada generador creado es independiente porque cada closure tiene sus propias variables capturadas.

```
package main

import "fmt"

func crearFibonacci() func() int {
    a, b := 0, 1
    return func() int {
        actual := a
        a, b = b, a+b
        return actual
    }
}

func main() {
    fib := crearFibonacci()
    for i := 0; i < 8; i++ {
        fmt.Print(fib(), " ")
    }
    fmt.Println() // 0 1 1 2 3 5 8 13
}
```

Ejercicio 4: Contar dígitos con recursión

Escribe la funcion recursiva `contarDigitos(n int) int` que devuelva cuantos digitos tiene un numero entero positivo. Por ejemplo, `contarDigitos(9875)` devuelve 4. Define primero el caso base por escrito.

Solucion explicada: el caso base es $n < 10$, un solo digito, que devuelve 1. El paso recursivo: un numero con mas digitos tiene 1 digito mas que el numero sin su ultimo digito, o sea $1 + \text{contarDigitos}(n/10)$. La division entera $n/10$ quita el ultimo digito en cada llamada, asi el problema se hace mas pequeno hasta llegar al caso base. Traza: $\text{contarDigitos}(9875) = 1 + \text{contarDigitos}(987) = 1 + 1 + \text{contarDigitos}(98) = \dots = 4$.

```

package main

import "fmt"

// Caso base: n < 10 -> 1 digito.
// Paso recursivo: 1 + contarDigitos(n sin su ultimo digito).
func contarDigitos(n int) int {
    if n < 10 {
        return 1
    }
    return 1 + contarDigitos(n/10)
}

func main() {
    fmt.Println(contarDigitos(9875)) // 4
    fmt.Println(contarDigitos(7))   // 1
    fmt.Println(contarDigitos(1000)) // 4
}

```

Ejercicio 5: Medir el tiempo con defer

Escribe la funcion `medir(nombre string, f func())` que ejecute la funcion `f`, mida cuanto tardo usando el paquete `time` y reporte la duracion. Usa `defer` para que la medicion ocurra al final, incluso si `f` provoca un `return` temprano (en este caso, si `f` termina de cualquier forma normal).

Solucion explicada: guardamos `time.Now()` al inicio y diferimos una funcion anonima que calcula `time.Since(inicio)` e imprime el resultado. Como el `defer` se ejecuta al salir de `medir`, la medicion abarca toda la ejecucion de `f`. Este es un ejemplo real de programacion de utilidades: envolver comportamiento (medicion, logging, reintentos) alrededor de funciones es un uso profesional y cotidiano de `defer` combinado con funciones como valores.


```

package main

import (
    "fmt"
    "time"
)

// Ejecuta f y reporta cuanto tardo, usando defer.
func medir(nombre string, f func()) {
    inicio := time.Now()
    defer func() {
        fmt.Printf("%s tardo %v\n", nombre, time.Since(inicio))
    }()
    f()
}

func trabajoPesado() {
    total := 0
    for i := 0; i < 100000000; i++ {
        total += i
    }
    _ = total
}

func main() {
    medir("trabajoPesado", trabajoPesado)
    medir("funcion anonima", func() {
        time.Sleep(100 * time.Millisecond)
    })
}

```

Ejercicio 6: Paquete geometría

Crea el paquete geometria con las funciones exportadas `AreaCirculo(radio float64) float64`, `AreaRectangulo(base, altura float64) float64` y `PerimetroRectangulo(base, altura float64) float64`, mas una constante privada para pi (usa `math.Pi` directamente o define la tuya). Luego escribe un main que importe el paquete y calcule el area de un circulo de radio 3 y el area y perimetro de un rectangulo de 4x5.

Solucion explicada: la carpeta geometria contiene un archivo con package geometria. Las tres funciones empiezan con mayuscula para ser exportadas. El main importa "miproyecto/geometria" (ajusta el nombre según tu modulo) y llama con el prefijo geometria. Este ejercicio cierra el circulo de la unidad: pasaste de funciones sueltas a codigo organizado en paquetes reutilizables, como los profesionales.

Archivo geometria/geometria.go:

```
// Paquete geometria: calculos de areas y perimetros.
package geometria

import "math"

// AreaCirculo calcula el area de un circulo dado su radio.
func AreaCirculo(radio float64) float64 {
    return math.Pi * radio * radio
}

// AreaRectangulo calcula base por altura.
func AreaRectangulo(base, altura float64) float64 {
    return base * altura
}

// PerimetroRectangulo calcula dos veces base mas dos veces altura.
func PerimetroRectangulo(base, altura float64) float64 {
    return 2*base + 2*altura
}
```

Archivo main.go:

```

package main

import (
    "fmt"

    "miproyecto/geometria"
)

func main() {
    fmt.Printf("Area circulo r=3: %.2f\n", geometria.AreaCirculo(3))
    fmt.Printf("Area rectangulo 4x5: %.2f\n", geometria.AreaRectangulo(4, 5))
    fmt.Printf("Perimetro rectangulo 4x5: %.2f\n", geometria.PerimetroRectangulo(4, 5))
}

```

Ejercicio 7: Validador componible

Escribe la funcion `validar(texto string, reglas ...func(string) error) error` que aplique cada regla validadora al texto, en orden, y devuelva el primer error que encuentre (o nil si todas pasan). Luego crea tres validadores: `noVacio`, `longitudMinima(5)` usando un closure fabrica, y `sinEspacios`. Prueba con varios textos.

Solucion explicada: este ejercicio combina tres temas de la unidad en uno solo: funciones variadicadas (`reglas ...func(string) error`), funciones como valores (cada regla es una funcion) y el patron `value, err` (se devuelve el primer error). El bucle `for-range` aplica cada regla y retorna en cuanto una falla: ese `return` temprano es la guarda que evita seguir validando algo que ya es invalido. El validador `longitudMinima` muestra la fabrica de closures: `crearValidadorLongitud(5)` captura el minimo y devuelve la funcion lista. Asi se construyen sistemas de validacion extensibles: agregar una regla nueva es solo escribir una funcion y pasarla.

```

package main

import (
    "errors"
    "fmt"
    "strings"
)

// Aplica cada regla en orden; devuelve el primer error o nil.
func validar(texto string, reglas ...func(string) error) error {
    for _, regla := range reglas {
        if err := regla(texto); err != nil {
            return err
        }
    }
    return nil
}

```

```

    }
}
return nil
}

func noVacio(s string) error {
    if s == "" {
        return errors.New("el texto no puede estar vacio")
    }
    return nil
}

// Fabrica de validadores: captura el minimo en un closure.
func crearValidadorLongitud(min int) func(string) error {
    return func(s string) error {
        if len(s) < min {
            return fmt.Errorf("minimo %d caracteres, tiene %d", min, len(s))
        }
        return nil
    }
}

func sinEspacios(s string) error {
    if strings.Contains(s, " ") {
        return errors.New("no se permiten espacios")
    }
    return nil
}

func main() {
    reglas := []func(string) error{noVacio, crearValidadorLongitud(5), sinEspacios}

    for _, texto := range []string{"", "abc", "hola mundo", "secreto123"} {
        if err := validar(texto, reglas...); err != nil {
            fmt.Printf("%q -> invalido: %v\n", texto, err)
        } else {
            fmt.Printf("%q -> valido\n", texto)
        }
    }
}

```

Resumen de la unidad

En esta unidad aprendiste a construir programas a partir de piezas reutilizables y bien organizadas. Dominaste la declaracion de funciones con `func`, parametros tipados y valores de retorno, entendiendo que los parametros se pasan por valor y que un buen nombre vale mas que un comentario. Adoptaste el patron `value`, `err` de multiples valores de retorno, la forma idiomática de Go para manejar errores de manera explicita y visible, revisando el error inmediatamente después de cada llamada. Viste como los retornos con nombre documentan la firma y permiten el `naked return` en funciones cortas, con cuidado de no sombrear variables. Usaste funciones variadicas para aceptar cualquier cantidad de argumentos y aprendiste a desempaquetar slices con

Descubriste que en Go las funciones son valores de primera clase: las guardaste en variables, las pasaste como argumentos y las organizaste en mapas de operaciones, abriendo la puerta a disenos flexibles y extensibles. Creaste funciones anonimas y closures que recuerdan su entorno, como el contador y el generador de folios, entendiendo la captura de variables y el cambio de semantica de las variables de bucle desde Go 1.22. Programaste limpieza garantizada con `defer`, recordando su orden LIFO, la evaluacion inmediata de argumentos y la trampa de usarlo dentro de bucles largos. Conociste `panic` y `recover` como mecanismo de emergencia, con la advertencia clara de que no sustituyen al manejo normal de errores con valores. Practicaste la recursion con el caso base y el paso recursivo, reconociendo cuando brilla y cuando conviene mas un bucle.

Finalmente, subiste al nivel de los paquetes: aprendiste a crearlos, a importarlos con rutas de modulo y la regla de oro de la visibilidad, donde la mayuscula inicial significa exportado y la minuscula significa privado. Inicializaste modulos con `go mod init`, entendiste el papel de `go.mod` y `go.sum`, adoptaste una estructura de proyecto profesional y conociste la funcion especial `init` con sus usos y sus riesgos.

La leccion profunda de la unidad es el principio de no repetirse: cada vez que copies y pegues codigo, es una funcion pidiendo nacer; cada vez que un archivo crezca sin control, es un paquete pidiendo existir. Con funciones y paquetes en tu caja de herramientas, ya no escribes guiones lineales: disenas sistemas de piezas pequenas, probables y combinables. En la siguiente unidad, esas piezas empezaran a guardar estado mas complejo con estructuras y metodos, y todo lo que aprendiste aqui sera la base sobre la que construyas.

UNIDAD 5: Arreglos, slices y mapas

Hasta ahora en este libro hemos trabajado con variables que guardan un solo valor a la vez: un numero, un texto, un booleano. Eso funciona para programas pequenos, pero el software real casi nunca vive de datos solitarios. Piensa en una lista de calificaciones de un grupo escolar, en los mensajes de un chat, en los productos de una tienda en linea, en los puntajes de los jugadores de un videojuego. Todos esos datos tienen algo en comun: son colecciones. Son varios valores del mismo tipo o de tipos relacionados que queremos guardar juntos, recorrer, modificar y consultar como si fueran un paquete.

En Go, las colecciones se manejan principalmente con tres estructuras: los arreglos, los slices y los mapas. Cada una tiene su personalidad, sus fortalezas y sus trampas. Los arreglos son la estructura mas basica y rigida: un contenedor de tamano fijo que guarda un numero exacto de elementos. Los slices son la estrella de Go: flexibles, eficientes, y la forma mas comun de trabajar con secuencias de datos. Los mapas son el equivalente a los diccionarios: guardan pares de clave y valor para buscar informacion por nombre en lugar de por posicion.

Dominar estas tres estructuras es lo que separa a alguien que "sabe sintaxis de Go" de alguien que puede escribir programas de verdad. Casi todo lo que haras en Go pasa por slices: leer lineas de un archivo, procesar respuestas de un servidor, manejar listas de usuarios, ordenar resultados. Los mapas aparecen en cuanto necesitas contar cosas, agrupar informacion o construir indices rapidos. Y los arreglos, aunque se usan menos directamente, son el fundamento sobre el que los slices estan contruidos, asi que entenderlos te da la intuicion para entender por que los slices se comportan como se comportan.

Esta unidad esta dividida en cinco partes. Primero, una explicacion detallada de cada estructura con analogias del mundo real, errores comunes que cometen hasta los programadores con experiencia, y buenas practicas. Después, ejemplos paso a paso con codigo Go real, correcto y compilable, que puedes copiar a tu computadora y ejecutar. Luego, una seccion de practica con ejercicios y soluciones explicadas para que pongas las manos en el teclado. Y al final, un resumen que condensa todo lo importante en una hoja de referencia rapida.

Un consejo antes de empezar: no te limites a leer el codigo. Go se aprende escribiendo Go. Abre tu editor, crea un archivo `main.go`, y prueba cada ejemplo con tus propias manos. Cambia valores, rompe el programa a proposito, observa los mensajes de error. Los errores son el mejor maestro: cada mensaje de error que entiendes hoy es un error que no cometeras en produccion manana. Y cuando llegues a las trampas de los slices (la seccion donde hablamos de `append` compartiendo el arreglo subyacente), tomate tu tiempo. Ese es el tema

que mas dolores de cabeza causa a quienes vienen de otros lenguajes, y entenderlo a fondo te va a ahorrar horas de depuracion en el futuro.

Vamos a empezar con la estructura mas sencilla y antigua: el arreglo.

Explicacion detallada

Arreglos: el armario de tamano fijo

Imagina un armario con exactamente cinco cajones. Ni cuatro, ni seis: cinco. Cada cajon tiene un numero, del 0 al 4, y en cada cajon puedes guardar una cosa. No puedes agregar un sexto cajon después; el armario se fabrico con cinco y con cinco se queda. Eso es un arreglo en Go: una secuencia de elementos del mismo tipo, con un tamano fijo que se decide al momento de crearlo y que nunca cambia.

La declaracion de un arreglo en Go tiene esta forma:

```
var calificaciones [5]int
```

Esto crea un arreglo llamado calificaciones con espacio para cinco enteros. Algo importante: en Go, el tamano forma parte del tipo. Un `[5]int` y un `[6]int` son tipos diferentes, tan diferentes como un `int` y un `string`. Esto significa que no puedes pasarle un arreglo de seis elementos a una funcion que espera uno de cinco, aunque ambos sean "arreglos de enteros". El compilador los ve como mundos distintos. Esta rigidez es intencional: le da al compilador mas informacion para atrapar errores y le permite al programa saber exactamente cuanta memoria necesita desde el principio.

Cuando declaras un arreglo sin inicializarlo, Go llena cada posicion con el valor cero del tipo. Para enteros, ceros; para strings, cadenas vacias; para booleanos, false. Asi que después de `var calificaciones [5]int`, tienes cinco ceros esperando a ser reemplazados.

Para asignar valores usas el indice entre corchetes. Ojo: los indices empiezan en cero, no en uno. El primer elemento es el `[0]` y el ultimo es el `[4]` en un arreglo de cinco elementos. Si intentas acceder al indice 5, el programa se detiene con un error de "index out of range". Go no perdona eso ni en compilacion (cuando puede detectarlo) ni en ejecucion.

```

package main

import "fmt"

func main() {
    var calificaciones [5]int
    calificaciones[0] = 90
    calificaciones[1] = 85
    calificaciones[2] = 92
    calificaciones[3] = 78
    calificaciones[4] = 88

    fmt.Println(calificaciones)
}

```

También puedes declarar e inicializar en una sola línea con un literal de arreglo:

```
calificaciones := [5]int{90, 85, 92, 78, 88}
```

Y si no quieres contar los elementos tu mismo, puedes usar `[...]` y dejar que el compilador cuente:

```
calificaciones := [...]int{90, 85, 92, 78, 88}
```

El compilador cuenta cinco valores y crea un `[5]int`. Comodo, pero usalo con moderacion: si mas adelante agregas un sexto valor al literal, el tipo cambia silenciosamente de `[5]int` a `[6]int`, y eso puede romper funciones que esperaban el tamaño original.

Para recorrer un arreglo, la forma idiomática en Go es el ciclo `for` con `range`:

```

for indice, valor := range calificaciones {
    fmt.Printf("Posicion %d: %d\n", indice, valor)
}

```

`range` te entrega dos cosas en cada vuelta: el índice y una copia del valor. Si solo necesitas el valor, puedes ignorar el índice con el identificador en blanco `\_`:


```

suma := 0
for _, valor := range calificaciones {
    suma += valor
}
fmt.Println("Promedio:", suma/len(calificaciones))

```

La funcion ``len()`` te dice cuantos elementos tiene el arreglo. Es tu mejor amiga para evitar errores de indice: recorre con ``for i := 0; i < len(calificaciones); i++`` o usa ``range`` y nunca tendras un "index out of range".

Un detalle que sorprende a quienes vienen de otros lenguajes: en Go, cuando pasas un arreglo a una funcion, se pasa por valor. Es decir, se copia completo. Si la funcion modifica el arreglo, el original no cambia. Esto es diferente a lenguajes como Python o JavaScript, donde las listas se pasan por referencia. En Go, copiar un arreglo grande es costoso, y por eso en la practica casi siempre trabajamos con slices en lugar de arreglos. Los arreglos son el cimiento; los slices son la casa que se construye encima.

Slices: la vista flexible sobre un arreglo

Si el arreglo es un armario de tamano fijo, el slice es como una ventana corrediza que puedes mover y ajustar sobre una fila de cajones. El slice no guarda los datos por si mismo: apunta a una seccion de un arreglo subyacente y te dice "aquí empiezan mis datos, aquí terminan, y tengo tanto espacio disponible". Esta descripcion tiene tres numeros importantes: el puntero al primer elemento, la longitud (`len`) y la capacidad (`cap`).

Declarar un slice es casi igual que declarar un arreglo, pero sin el tamano:

```

var numeros []int

```

Ojo con esto: ``var numeros []int`` crea un slice nil, es decir, un slice que no apunta a ningun arreglo. Su longitud y capacidad son cero. Es perfectamente valido y util, como veremos mas adelante, pero no puedes escribir en sus posiciones porque no hay posiciones. Para eso necesitas crearlo con ``make`` o con un literal:

```

numeros := []int{10, 20, 30, 40, 50}

```

Nota la diferencia sutil: ``[]int{...}`` es un slice, ``[5]int{...}`` es un arreglo. Un par de corchetes vacios cambia todo.

La funcion ``make`` es la forma mas controlada de crear un slice:

```
s := make([]int, 5)           // longitud 5, capacidad 5
s2 := make([]int, 5, 10)     // longitud 5, capacidad 10
```

`make([]int, 5)` crea un slice de cinco ceros con capacidad cinco. `make([]int, 5, 10)` crea un slice de cinco ceros pero con espacio reservado para diez: puedes crecer hasta diez elementos sin que Go tenga que pedir mas memoria. La capacidad es una optimizacion: si sabes de antemano cuantos elementos vas a manejar, reservar la capacidad desde el principio evita que Go tenga que reacomodar la memoria varias veces mientras el slice crece.

La longitud (len) es cuantos elementos tiene el slice ahora mismo. La capacidad (cap) es cuantos elementos puede contener antes de necesitar mas memoria. La longitud nunca puede ser mayor que la capacidad, pero si puede ser menor. Piensa en un vaso (capacidad) con agua (longitud): el vaso puede contener 250 mililitros, pero ahorita solo tiene 100.

Puedes crear un slice a partir de un arreglo existente usando la sintaxis de rebanado (slicing):

```
arreglo := [5]int{10, 20, 30, 40, 50}
s := arreglo[1:4]
fmt.Println(s) // [20 30 40]
```

`s` es un slice que apunta a los elementos del arreglo desde el indice 1 hasta el 3 (el limite superior no se incluye). Lo crucial es que `s` no copia los datos: comparte el mismo arreglo subyacente. Si modificas `s[0]`, estas modificando `arreglo[1]`. Esto es poderoso y peligroso a la vez, como veremos en la seccion de trampas.

También puedes rebanar un slice para obtener otro slice:

```
original := []int{10, 20, 30, 40, 50}
parte := original[1:3]
fmt.Println(parte) // [20 30]
```

append: como crece un slice

La funcion `append` es la forma de agregar elementos a un slice:

```
s := []int{1, 2, 3}
s = append(s, 4)
fmt.Println(s) // [1 2 3 4]
```

Fijate en algo fundamental: `s = append(s, 4)`. Hay que asignar el resultado de vuelta a `s`. Esto no es un capricho de estilo; es necesario porque `append` puede devolver un slice completamente nuevo. Cuando el slice tiene capacidad disponible, `append` escribe el nuevo elemento en el espacio libre y devuelve un slice con longitud mayor. Pero cuando la capacidad se agota, `append` hace algo mas dramático: pide un bloque de memoria mas grande, copia todos los elementos al nuevo bloque, agrega el nuevo elemento, y devuelve un slice que apunta al nuevo bloque. El slice original sigue apuntando al bloque viejo. Si no asignas el resultado, pierdes el crecimiento.

```
s := make([]int, 0, 2)
fmt.Println(len(s), cap(s)) // 0 2

s = append(s, 1)
fmt.Println(len(s), cap(s)) // 1 2

s = append(s, 2)
fmt.Println(len(s), cap(s)) // 2 2

s = append(s, 3)
fmt.Println(len(s), cap(s)) // 3 4 <- la capacidad se duplico
```

Cuando la capacidad se agota, Go normalmente la duplica (para slices pequeños; para slices grandes el crecimiento es mas moderado). Este crecimiento amortizado hace que agregar elementos uno por uno sea eficiente en promedio, aunque de vez en cuando haya una operacion costosa de copiar todo.

Puedes agregar varios elementos a la vez, e incluso agregar todos los elementos de otro slice usando `...`:

```
s := []int{1, 2}
s = append(s, 3, 4, 5)
otros := []int{6, 7}
s = append(s, otros...)
fmt.Println(s) // [1 2 3 4 5 6 7]
```

La trampa mas famosa: append compartiendo el arreglo subyacente

Aqui llegamos al tema que mas confunde a los principiantes y que causa los errores mas difíciles de rastrear. Recuerda: un slice es una vista sobre un arreglo subyacente. Dos slices pueden compartir el mismo arreglo. Y `append` escribe en ese arreglo compartido siempre que haya capacidad disponible.

Mira este ejemplo con atencion:

```
package main

import "fmt"

func main() {
    base := []int{1, 2, 3}
    a := base[0:2] // [1 2], pero con capacidad para 3
    b := base[0:2] // [1 2], comparte el mismo arreglo

    a = append(a, 99)
    fmt.Println("a:", a)    // [1 2 99]
    fmt.Println("b:", b)    // [1 2]
    fmt.Println("base:", base) // [1 2 99] <- cambio sin avisar
}
```

Que paso? `a := base[0:2]` crea un slice de longitud 2 pero con capacidad 3, porque el arreglo subyacente tiene un elemento mas disponible. Cuando hacemos `append(a, 99)`, como hay capacidad, Go escribe el 99 directamente en la tercera posicion del arreglo compartido. El slice `b` no ve el 99 porque su longitud sigue siendo 2, pero `base` si muestra `[1 2 99]` porque su longitud es 3 e incluye esa posicion. Modificamos `base` sin tocar `base` directamente. Este es el tipo de error que te hace perder una tarde entera.

La leccion practica: si vas a hacer `append` sobre un slice que viene de rebanar otro, y no quieres afectar al original, haz una copia primero con `copy` o crea un slice nuevo con `make` y copia los elementos. O usa la forma completa de slicing con limite de capacidad: `base[0:2:2]`, donde el tercer numero limita la capacidad a 2. Con capacidad igual a la longitud, cualquier `append` se ve obligado a crear un arreglo nuevo:

```
a := base[0:2:2] // longitud 2, capacidad 2
a = append(a, 99) // obligatoriamente crea un arreglo nuevo
fmt.Println(base) // [1 2 3] <- intacto
```

Este patron `s[j:k]` se llama "full slice expression" y es tu seguro contra modificaciones accidentales cuando pasas porciones de slices a otras funciones.

Otra trampa relacionada: guardar el resultado de `append` en una variable y seguir usando la original como si nada hubiera cambiado. Si `append` tuvo que reubicar la memoria, la variable original apunta a datos viejos. Por eso la regla de oro: siempre asigna el resultado de `append`, y considera que cualquier otro slice que compartia el arreglo puede haber quedado obsoleto o modificado.

copy: duplicar datos de verdad

Cuando necesitas una copia independiente, usa la funcion `copy`:

```
origen := []int{1, 2, 3, 4, 5}
destino := make([]int, len(origen))
n := copy(destino, origen)
fmt.Println(destino) // [1 2 3 4 5]
fmt.Println(n)       // 5, elementos copiados
```

`copy(destino, origen)` copia elementos del origen al destino y devuelve cuantos copio. Copia el minimo entre la longitud del destino y la del origen. Si el destino es mas pequeno, solo copia lo que cabe; no crece automaticamente. Después de `copy`, los dos slices son independientes: modificar uno no afecta al otro.

Un patron comun para clonar un slice es:

```
clon := append([]int(nil), original...)
```

Esto crea un slice nuevo y copia todos los elementos. Funciona incluso si `original` es nil (el resultado es nil, no un slice vacio). Es idiomático y lo veras mucho en código Go real.

Slices nil vs slices vacios

En Go hay una distincion sutil pero importante:

```
var s1 []int           // slice nil: no apunta a nada
s2 := []int{}          // slice vacio: apunta a un arreglo de tamaño cero
s3 := make([]int, 0)    // slice vacio: longitud y capacidad cero
```

Los tres tienen longitud cero y se comportan igual en casi todo: `len()` devuelve 0, `range` no ejecuta ninguna vuelta, `append` funciona en los tres. La diferencia es que `s1 == nil` es verdadero, mientras que para `s2` y `s3` es falso.

Cual usar? La convencion de Go favorece el slice nil cuando no tienes elementos todavía. Es mas simple declarar `var s []int` e ir agregando con `append`. Muchas funciones de la libreria estandar devuelven nil cuando no hay resultados, y el código que consume esos resultados funciona igual porque `len(nil)` es 0 y `range` sobre nil no hace nada. Usa slices vacios (`[]int{}` o `make`) cuando necesites distinguir explicitamente entre "sin datos" y "datos vacios", por ejemplo al convertir a JSON: un slice nil se convierte en `null`, mientras que un slice vacio se convierte en `[]`. Esa diferencia si le importa a quien recibe tu JSON.

Strings: slices de bytes disfrazados

Los strings en Go estan intimamente relacionados con los slices: un string es, en esencia, un slice de bytes de solo lectura. Puedes convertir entre ellos:

```
texto := "hola"
bytes := []byte(texto)    // slice de bytes
deNuevo := string(bytes)  // de vuelta a string
```

Puedes indexar un string como un slice: `texto[0]` te da el primer byte. Y puedes rebanarlo: `texto[1:3]` te da los bytes del 1 al 2. Pero aqui hay una trampa enorme para quienes trabajamos en espanol: los caracteres acentuados y la enie ocupan mas de un byte en UTF-8.

```
palabra := "nino"
fmt.Println(len(palabra)) // 4, un byte por letra
fmt.Println(palabra[2])   // 110, el byte de la 'n'... espera
```

Con "nino" todo bien porque son letras ASCII. Pero mira:

```
palabra := "niño"
fmt.Println(len(palabra)) // 5, no 4
```

"niño" tiene 4 letras pero 5 bytes, porque la `ñ` ocupa 2 bytes en UTF-8. Si recorres el string byte por byte, la `ñ` se rompe en dos pedazos sin sentido. La forma correcta de recorrer caracteres es con `range`, que decodifica runes (caracteres Unicode completos):

```
palabra := "niño"
for indice, runa := range palabra {
    fmt.Printf("byte %d: %c\n", indice, runa)
}
```

Esto imprime cada letra completa, con la `ñ` intacta. El indice que te da `range` es la posicion del byte donde empieza cada runa, no el numero de caracter. Si necesitas el numero de caracter, lleva tu propio contador o usa el paquete `unicode/utf8`:

```
import "unicode/utf8"

palabra := "niño"
fmt.Println(utf8.RuneCountInString(palabra)) // 4, caracteres reales
```

Y si necesitas manipular caracteres individualmente, convierte a slice de runes:

```
runas := []rune("niño")
fmt.Println(len(runas)) // 4
runas[2] = 'ñ'           // funciona, es la enie completa
fmt.Println(string(runas))
```

Regla practica: si tu programa maneja texto en espanol (acentos, enies, signos de apertura como `¿` y `¡`), piensa en runes cuando recorras o modifiques caracteres, y en bytes cuando solo necesites medir o copiar. El 99% de los errores con texto en Go vienen de tratar un string como si cada byte fuera un caracter.

Mapas: el diccionario de Go

Si los slices son listas ordenadas por posicion, los mapas son diccionarios: guardan pares de clave y valor, y te permiten buscar un valor por su clave en lugar de por su posicion. Piensa en una agenda telefonica: no buscas el telefono numero 42, buscas el telefono "de Maria". La clave es el nombre, el valor es el numero.

Se declaran asi:

```
var edades map[string]int
```

Pero cuidado: igual que con los slices, declarar un mapa con `var` crea un mapa nil. Puedes leer de un mapa nil (te devuelve el valor cero), pero no puedes escribir en el: el programa se detiene con un error. Para un mapa usable, inicializalo con `make` o con un literal:

```
edades := make(map[string]int)
edades["Ana"] = 25
edades["Luis"] = 30

// o con un literal:
edades := map[string]int{
    "Ana": 25,
    "Luis": 30,
}
```

Leer un valor es directo:

```
fmt.Println(edades["Ana"]) // 25
```

Pero que pasa si preguntas por una clave que no existe?

```
fmt.Println(edades["Pedro"]) // 0
```

Go no se queja: te devuelve el valor cero del tipo (0 para int). Esto es un problema: como distingues entre "Pedro tiene 0 anos" y "Pedro no esta en el mapa"? Con la coma-ok, el idioma mas importante de los mapas en Go:

```
edad, existe := edades["Pedro"]
if existe {
    fmt.Println("Pedro tiene", edad, "anos")
} else {
    fmt.Println("Pedro no esta registrado")
}
```

Cuando accedes a un mapa con dos variables, la segunda te dice si la clave existia. Usa este patron siempre que la ausencia de la clave sea un caso que tu programa deba manejar. Es tan comun que tiene nombre propio: "comma ok".

Para borrar una entrada usa ``delete``:

```
delete(edades, "Luis")
```

Si la clave no existe, ``delete`` no hace nada: no hay error. Para saber cuantas entradas tiene un mapa, usa ``len(edades)``.

Recorrer un mapa se hace con ``range``:

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

Pero atencion: el orden en que ``range`` recorre un mapa es aleatorio. No es un error, es intencional: Go mezcla el orden a proposito para que no escribas programas que dependan de un orden que no esta garantizado. Si necesitas un orden especifico, extrae las claves a un slice, ordenalo con el paquete ``sort``, y recorre el slice:


```
import "sort"

nombres := make([]string, 0, len(edades))
for nombre := range edades {
    nombres = append(nombres, nombre)
}
sort.Strings(nombres)
for _, nombre := range nombres {
    fmt.Printf("%s tiene %d anos\n", nombre, edades[nombre])
}
```

Los mapas también pueden guardar structs como valores, lo cual es extremly comun. Por ejemplo, un catalogo de productos donde la clave es el codigo y el valor es un struct con nombre y precio. Hablaremos de structs en la siguiente unidad, pero ve adelantando la idea: ``map[string]Producto``.

Un detalle importante: las claves de un mapa deben ser de un tipo comparable (strings, numeros, booleanos, y tipos compuestos por ellos). No puedes usar un slice como clave porque los slices no son comparables. Si necesitas una clave compuesta, usa un struct o concatena strings.

Cuando usar cada estructura

Con tres herramientas en la caja, la pregunta es cuando usar cada una. Aqui van las reglas practicas:

Usa arreglos cuando el tamano es fijo por naturaleza y conocido en compilacion: los dias de la semana, los meses del ano, una coordenada 3D, una direccion IP (4 bytes). En esos casos, que el tamano forme parte del tipo es una ventaja: el compilador te protege.

Usa slices para casi todo lo demas: listas de resultados, lineas de un archivo, elementos que vas acumulando, cualquier secuencia donde el tamano se conoce hasta la ejecucion. Si dudas entre arreglo y slice, elige slice. Es la respuesta correcta el 95% de las veces.

Usa mapas cuando necesites buscar por clave: contar ocurrencias, agrupar elementos, construir indices, caches, configuraciones por nombre. Si te encuentras recorriendo un slice con un ciclo para encontrar un elemento por su nombre, probablemente necesitas un mapa.

Y combina: un slice de mapas (`[]map[string]int``), un mapa de slices (``map[string][]string``), un mapa de structs. Las estructuras de datos reales son combinaciones. Por ejemplo, agrupar estudiantes por grupo escolar es un ``map[string][]string`` donde la clave es el nombre del grupo y el valor es la lista de alumnos.

El paquete sort: ordenar slices

Go trae el paquete `sort` en su libreria estandar para ordenar slices. Para los tipos basicos es trivial:

```
import "sort"

numeros := []int{5, 2, 8, 1, 9}
sort.Ints(numeros)
fmt.Println(numeros) // [1 2 5 8 9]

nombres := []string{"Pedro", "Ana", "Luis"}
sort.Strings(nombres)
fmt.Println(nombres) // [Ana Luis Pedro]
```

`sort.Ints`, `sort.Strings` y `sort.Float64s` ordenan en el lugar: modifican el slice original, no devuelven uno nuevo. Para ordenar con un criterio personalizado, usa `sort.Slice`:

```
personas := []string{"Ana", "Pedro", "Luis", "Sofia"}
// ordenar por longitud del nombre
sort.Slice(personas, func(i, j int) bool {
    return len(personas[i]) < len(personas[j])
})
```

La funcion que le pasas recibe dos indices y devuelve true si el elemento `i` debe ir antes que el elemento `j`. Este patron de "funcion comparadora" aparece en muchos lenguajes y vale la pena dominarlo.

También existe `sort.SliceStable`, que conserva el orden original de los elementos empatados. Si ordenas dos veces con criterios diferentes, el estable respeta el orden previo en los empates. Para datos pequenos no importa; para reportes y tablas, si.

Con esto cerramos la teoria. Ahora vamos a ver todo esto en accion con ejemplos paso a paso que puedes ejecutar en tu computadora.

Slices de slices: tablas y matrices

Un slice puede contener cualquier tipo, incluyendo otros slices. Un `[][]int` es un slice de slices de enteros, y es la forma natural de representar tablas, matrices o tableros de juego en Go. Cada fila es un slice independiente, lo cual significa que las filas pueden tener longitudes distintas (una "matriz dentada").

```
// Tablero de 3x3 para gato (tic-tac-toe)
tablero := make([][]string, 3)
for i := range tablero {
    tablero[i] = make([]string, 3)
}
tablero[1][1] = "X"
```

Fijate en el patron: primero creas el slice exterior con `make([][]string, 3)`, y luego en un ciclo creas cada fila. Si olvidas el ciclo, cada `tablero[i]` es nil y al intentar `tablero[0][0] = "X"` el programa se detiene. Este es un error muy comun: `make` solo inicializa un nivel.

Para recorrer una tabla usas dos ciclos anidados:

```
for i, fila := range tablero {
    for j, celda := range fila {
        fmt.Printf("[%d][%d]=%q ", i, j, celda)
    }
    fmt.Println()
}
```

Y si las filas pueden tener distinta longitud, `len(fila)` te dice cuanto mide cada una. Esta flexibilidad es una ventaja sobre las matrices de otros lenguajes: puedes representar un triangulo, una lista de listas de comentarios, o los renglones de un archivo CSV donde cada renglon tiene distinto numero de columnas.

Borrar y filtrar elementos de un slice

Go no trae una funcion `remove` para slices, pero el idioma para borrar el elemento en la posicion `i` es:

```
s := []int{10, 20, 30, 40, 50}
i := 2 // queremos borrar el 30
s = append(s[:i], s[i+1:]...)
fmt.Println(s) // [10 20 40 50]
```

Como funciona: `s[:i]` es `[10 20]` y `s[i+1:]` es `[40 50]`. Al hacer `append` del segundo sobre el primero, los elementos se recorren una posicion hacia la izquierda, pisando el 30. El slice resultante tiene un elemento menos. Pero ojo con dos detalles. Primero, esto modifica el arreglo subyacente: el ultimo elemento (50) queda duplicado en la memoria mas alla de la nueva longitud, lo cual normalmente no importa, pero si el slice contenia punteros u objetos grandes, conviene limpiar la cola para que el recolector de basura pueda liberarlos:

```
// Version cuidadosa para slices de punteros o structs grandes
s = append(s[:i], s[i+1:]...)
s[len(s)-1] = 0 // o nil, según el tipo: libera la referencia
s = s[:len(s)-1]
```

Segundo, si el orden no te importa, hay un truco $O(1)$: copia el ultimo elemento sobre el que quieres borrar y recorta el final:

```
s[i] = s[len(s)-1]
s = s[:len(s)-1]
```

Esto es mucho mas rapido para slices grandes porque no recorre nada, pero desordena los elementos. Usalo cuando el orden sea irrelevante, por ejemplo en una bolsa de objetos de un videojuego.

Filtrar (quedarte solo con los elementos que cumplen una condicion) se hace en el lugar con el mismo principio:

```
numeros := []int{1, 2, 3, 4, 5, 6}
// quedarnos solo con los pares, reutilizando el mismo arreglo
pares := numeros[:0]
for _, n := range numeros {
    if n%2 == 0 {
        pares = append(pares, n)
    }
}
fmt.Println(pares) // [2 4 6]
```

`numeros[:0]` crea un slice de longitud cero que comparte el arreglo. Al ir agregando solo los pares, los escribimos desde el principio del mismo arreglo. Es eficiente (cero memoria extra) pero destruye el contenido original: después del filtrado, `numeros` ya no contiene los datos originales. Si necesitas conservar el original, filtra sobre un slice nuevo.

Mapas anidados y el problema del mapa interno nil

Puedes anidar mapas: `map[string]map[string]int` podria representar, por ejemplo, las calificaciones de cada estudiante por materia. Pero hay una trampa: el mapa exterior no inicializa los interiores automaticamente.

```
califs := make(map[string]map[string]int)
// califs["Ana"] es nil aqui; escribir en el seria un error
califs["Ana"] = make(map[string]int) // primero crear el interior
califs["Ana"]["matematicas"] = 9
```

El patron seguro es verificar antes de escribir:

```
func registrar(califs map[string]map[string]int, alumno, materia string, cal int) {
    if califs[alumno] == nil {
        califs[alumno] = make(map[string]int)
    }
    califs[alumno][materia] = cal
}
```

Este patron aparece en configuraciones por secciones, caches de dos niveles, y conteos agrupados por categoria. Si te parece verboso, es porque lo es: Go prefiere ser explicito antes que magico. La ventaja es que nunca te preguntas "quien creo este mapa?": siempre fuiste tu, en una linea visible.

Reservar capacidad: cuando usar make con tres argumentos

Vimos que `append` duplica la capacidad cuando se agota. Eso significa que construir un slice de 10,000 elementos agregando uno por uno provoca alrededor de 14 realojamientos (2, 4, 8, 16... hasta 16384), cada uno copiando todo lo acumulado. Para 10,000 elementos no se nota; para millones, si.

Si sabes el tamano final (o un aproximado), reservalo:

```
// Mal: realojamientos repetidos
var s []int
for i := 0; i < 1000000; i++ {
    s = append(s, i)
}

// Bien: una sola reserva de memoria
s := make([]int, 0, 1000000)
for i := 0; i < 1000000; i++ {
    s = append(s, i)
}
```

La segunda version pide la memoria una sola vez. Es el mismo resultado, mucho menos trabajo para el recolector de basura. Esta optimizacion es de las mas rentables en Go: una

línea que cambia el rendimiento sin cambiar la lógica. Acostúmbrate a preguntarte "¿se cuántos elementos habrá?" cada vez que construyas un slice en un ciclo. Si la respuesta es sí, usa ``make`` con capacidad.

Los siete pecados capitales de esta unidad

Para cerrar la teoría, una lista de los errores que más se repiten, para que los reconozcas cuando aparezcan:

1. No asignar el resultado de ``append``. El crecimiento se pierde y el programa sigue con el slice viejo.
2. Escribir en un mapa nil. Declarar con ``var m map[string]int`` y luego ``m["x"] = 1`` detiene el programa.
3. Asumir orden en un mapa. El ``range`` sobre mapas es aleatorio a propósito.
4. Confundir ``len`` de un string con número de caracteres. Con texto en español, casi nunca coinciden.
5. Indexar un string con caracteres multibyte. ``s[i]`` te da un byte, no una letra.
6. Rebanar y luego hacer ``append`` sin pensar en quien más comparte el arreglo. El cambio fantasma del ejemplo 3.
7. Usar ``[5]int`` donde se necesitaba ``[]int``. El tamaño fijo es parte del tipo y no crece.

Si algún día tu programa hace algo "imposible", regresa a esta lista. Nueve de cada diez veces, el culpable está aquí.

Comparar slices y buscar elementos

Una pregunta frecuente: ¿cómo saber si dos slices son iguales? En Go no puedes compararlos con ``==`` (el compilador lo prohíbe, excepto contra nil). La forma moderna, disponible desde Go 1.21, es el paquete ``slices`` de la librería estándar:

```
import "slices"

a := []int{1, 2, 3}
b := []int{1, 2, 3}
fmt.Println(slices.Equal(a, b)) // true
```

El paquete ``slices`` trae varias utilidades que antes había que escribir a mano: ``slices.Contains(s, v)`` para saber si un valor está en el slice, ``slices.Index(s, v)`` para encontrar

su posicion (o -1 si no esta), ``slices.Sort(s)`` como alternativa a ``sort.Ints`` para enteros, ``slices.Clone(s)`` para clonar, y ``slices.Delete(s, i, j)`` para borrar un rango. Si tu Go es 1.21 o mas reciente (que es lo que pide este libro), usalas con confianza: estan probadas, son eficientes y hacen tu codigo mas legible.

```
nombres := []string{"ana", "luis", "pedro"}
fmt.Println(slices.Contains(nombres, "luis")) // true
fmt.Println(slices.Index(nombres, "pedro"))   // 2
nombres = slices.Delete(nombres, 1, 2)        // borra "luis"
fmt.Println(nombres)                          // [ana pedro]
```

Ojo: ``slices.Delete`` modifica el arreglo subyacente igual que el idioma manual con ``append``, asi que aplican las mismas precauciones sobre datos compartidos. Y ``slices.Contains`` hace una busqueda lineal $O(n)$: para membresias frecuentes sobre conjuntos grandes, un ``map[T]bool`` sigue siendo mas rapido. La regla de oro se mantiene: elige la estructura según el patron de acceso, no por comodidad del momento.

Ejemplos paso a paso

Ejemplo 1: Promedio de calificaciones con un arreglo

Empezamos con lo mas basico: guardar las calificaciones de un estudiante en un arreglo y calcular su promedio. Este ejemplo muestra la declaracion de un arreglo, la inicializacion con un literal, el recorrido con ``for-range`` y el uso de ``len``.

```

package main

import "fmt"

func main() {
    // Arreglo de 5 calificaciones
    calificaciones := [5]float64{9.5, 8.0, 7.5, 10.0, 8.5}

    suma := 0.0
    for _, cal := range calificaciones {
        suma += cal
    }

    promedio := suma / float64(len(calificaciones))
    fmt.Printf("Promedio: %.2f\n", promedio)
}

```

Observa los detalles: usamos `float64` porque las calificaciones tienen decimales. En el `range` ignoramos el índice con `_` porque no lo necesitamos. Y convertimos `len(calificaciones)` a `float64` porque en Go no puedes dividir un `float64` entre un `int` sin conversión explícita. El `%.2f` en `Printf` muestra el promedio con dos decimales.

Ejemplo 2: Lista de compras dinamica con slices y append

Ahora algo mas real: una lista de compras donde el usuario va agregando productos uno por uno. No sabemos cuantos seran, así que el arreglo no sirve; necesitamos un slice que crezca con `append`.


```

package main

import "fmt"

func main() {
    // Empezamos con un slice nil: es valido y append funciona sobre el
    var lista []string

    // Simulamos productos que el usuario agrega
    productos := []string{"tortillas", "huevos", "leche", "pan", "queso"}
    for _, p := range productos {
        lista = append(lista, p)
        fmt.Printf("Agregado: %s (total: %d)\n", p, len(lista))
    }

    fmt.Println("\nLista final de compras:")
    for i, item := range lista {
        fmt.Printf("%d. %s\n", i+1, item)
    }
}

```

Tres cosas a notar. Primera: `var lista []string` crea un slice nil y `append` funciona perfectamente sobre el; no necesitas `make` si vas a construir el slice agregando elementos. Segunda: siempre asignamos el resultado de `append` de vuelta a `lista`. Tercera: al mostrar la lista usamos `i+1` porque los humanos contamos desde 1 aunque Go cuente desde 0.

Ejemplo 3: Rebanar un slice y la trampa del arreglo compartido

Este ejemplo demuestra visualmente la trampa mas famosa de los slices. Vamos a rebanar un slice, modificar la rebanada con `append`, y observar como el original cambia sin que lo toquemos directamente. Después mostramos la solucion con la expresion de rebanado completa.

```

package main

import "fmt"

func main() {
    datos := []int{10, 20, 30, 40, 50}

    // Rebanada: longitud 2, pero capacidad 5 (llega al final del arreglo)
    ventana := datos[0:2]
    fmt.Println("ventana antes:", ventana) // [10 20]
    fmt.Println("datos antes: ", datos)    // [10 20 30 40 50]

    // Como hay capacidad disponible, append escribe sobre el arreglo compartido
    ventana = append(ventana, 99, 88)
    fmt.Println("ventana despues:", ventana) // [10 20 99 88]
    fmt.Println("datos despues: ", datos)    // [10 20 99 88 50] <- cambio fantasma

    fmt.Println("---- con proteccion ----")
    datos2 := []int{10, 20, 30, 40, 50}
    // El tercer indice limita la capacidad: append ya no puede reusar el arreglo
    segura := datos2[0:2:2]
    segura = append(segura, 99, 88)
    fmt.Println("segura:", segura) // [10 20 99 88]
    fmt.Println("datos2:", datos2) // [10 20 30 40 50] <- intacto
}

```

Ejecuta este programa y estudia la salida. La primera parte muestra el problema: `datos` cambio de `[10 20 30 40 50]` a `[10 20 99 88 50]` sin que escribieramos `datos` en ninguna linea. La segunda parte muestra la solucion: `datos2[0:2:2]` limita la capacidad a 2, asi que `append` se ve obligado a crear un arreglo nuevo y el original queda intacto. Este es uno de esos ejemplos que vale la pena ejecutar y modificar varias veces hasta que la idea quede grabada.

Ejemplo 4: Copiar un slice correctamente

Cuando necesitas duplicar un slice para trabajar sobre la copia sin afectar el original, `copy` es tu herramienta. Este ejemplo muestra la forma correcta y un error comun: crear un destino sin longitud suficiente.

```

package main

import "fmt"

func main() {
    original := []string{"lunes", "martes", "miercoles"}

    // Forma correcta: destino con la misma longitud
    copia := make([]string, len(original))
    n := copy(copia, original)
    fmt.Printf("Copiados %d elementos: %v\n", n, copia)

    // Modificar la copia no afecta al original
    copia[0] = "LUNES"
    fmt.Println("original:", original) // [lunes martes miercoles]
    fmt.Println("copia:  ", copia)     // [LUNES martes miercoles]

    // Error comun: destino sin longitud
    mala := make([]string, 0)
    n2 := copy(mala, original)
    fmt.Printf("Con destino vacio se copiaron %d elementos: %v\n", n2, mala)
}

```

El ultimo caso es instructivo: `copy` no crece el destino, asi que con un destino de longitud 0 se copian 0 elementos. Si quieres clonar en una linea, el idioma es `clon := append([]string(nil), original...)`.

Ejemplo 5: Contar palabras con un mapa

Los mapas brillan cuando necesitas contar o agrupar. Este ejemplo cuenta cuantas veces aparece cada palabra en un texto. Es el "hola mundo" de los mapas y aparece en entrevistas de trabajo con mas frecuencia de la que imaginas.

```

package main

import (
    "fmt"
    "sort"
    "strings"
)

func main() {
    texto := "el gato come pescado y el perro come croquetas y el gato duerme"

    conteo := make(map[string]int)
    palabras := strings.Fields(texto) // separa por espacios
    for _, p := range palabras {
        conteo[p] = conteo[p] + 1
        // Truco: si la clave no existe, conteo[p] vale 0, así que esto funciona
        // desde la primera vez sin necesidad de verificar existencia
    }

    // Para mostrar en orden alfabético, extraemos y ordenamos las claves
    claves := make([]string, 0, len(conteo))
    for palabra := range conteo {
        claves = append(claves, palabra)
    }
    sort.Strings(claves)

    fmt.Println("Conteo de palabras:")
    for _, palabra := range claves {
        fmt.Printf("  %s: %d\n", palabra, conteo[palabra])
    }
}

```

El truco central está en `conteo[p] = conteo[p] + 1`. La primera vez que vemos una palabra, `conteo[p]` devuelve 0 (el valor cero) porque la clave no existe; le sumamos 1 y la guardamos. Así evitamos escribir un `if` con la coma-ok para el caso de conteo simple. Después, como el orden de los mapas es aleatorio, extraemos las claves a un slice, las ordenamos con `sort.Strings`, y recorremos el slice para imprimir en orden. Este patrón (mapa para contar + slice ordenado de claves para mostrar) lo usarás una y otra vez.

Ejemplo 6: Agenda telefonica con coma-ok y delete

Este ejemplo usa todas las operaciones básicas de un mapa: insertar, leer con la coma-ok para distinguir "no existe", actualizar y borrar con `delete`.

```

package main

import "fmt"

func buscar(agenda map[string]string, nombre string) {
    telefono, existe := agenda[nombre]
    if existe {
        fmt.Printf("%s: %s\n", nombre, telefono)
    } else {
        fmt.Printf("%s no esta en la agenda\n", nombre)
    }
}

func main() {
    agenda := map[string]string{
        "Ana": "555-1234",
        "Luis": "555-5678",
    }

    buscar(agenda, "Ana") // Ana: 555-1234
    buscar(agenda, "Pedro") // Pedro no esta en la agenda

    // Actualizar es igual que insertar: se sobrescribe el valor
    agenda["Ana"] = "555-9999"
    buscar(agenda, "Ana") // Ana: 555-9999

    // Borrar
    delete(agenda, "Luis")
    buscar(agenda, "Luis") // Luis no esta en la agenda

    fmt.Println("Contactos restantes:", len(agenda))
}

```

Fijate en la funcion `buscar`: encapsula el patron de la coma-ok para no repetirlo en cada busqueda. Sin la coma-ok, buscar a "Pedro" devolveria una cadena vacia y no podriamos distinguir entre "no tiene telefono registrado" y "su telefono es una cadena vacia". En programas reales, esa distincion suele ser la diferencia entre un error silencioso y un comportamiento correcto.

Ejemplo 7: Recorrer texto en espanol sin romper los acentos

Este ejemplo muestra la diferencia entre recorrer un string por bytes (mal para texto en espanol) y por runes (bien). Es un error clasico que produce texto corrupto.

```

package main

import (
    "fmt"
    "unicode/utf8"
)

func main() {
    frase := "El niño come piña"

    fmt.Println("Longitud en bytes:", len(frase))
    fmt.Println("Longitud en runes:", utf8.RuneCountInString(frase))

    fmt.Println("\nRecorrido por bytes (MAL para español):")
    for i := 0; i < len(frase); i++ {
        fmt.Printf("%c ", frase[i])
    }
    fmt.Println("\n\nRecorrido por runes con range (BIEN):")
    for _, r := range frase {
        fmt.Printf("%c ", r)
    }
    fmt.Println()

    // Modificar una letra acentuada: convertir a []rune primero
    runas := []rune("camión")
    runas[3] = 'o' // la 'o' con acento esta en la posicion 3 como runa
    fmt.Println("\nModificado:", string(runas))
}

```

Si ejecutas este programa veras que el recorrido por bytes imprime simbolos raros donde deberian estar la `ñ` y la `i` con acento, porque cada uno de esos caracteres ocupa 2 bytes y el ciclo los parte por la mitad. El recorrido con `range` los muestra intactos. La ultima parte muestra como modificar un caracter: conviertes a `[]rune`, modificas por posicion de runa, y conviertes de vuelta a string. Memoriza este patron; lo necesitaras cada vez que manipules texto en espanol.

Ejemplo 8: Ordenar una tabla con sort.Slice

Para cerrar los ejemplos, algo que parece simple pero tiene su chiste: ordenar una lista de estudiantes por promedio, de mayor a menor, usando `sort.Slice` con una funcion comparadora.

```

package main

import (
    "fmt"
    "sort"
)

func main() {
    nombres := []string{"Ana", "Luis", "Sofia", "Pedro"}
    promedios := []float64{8.5, 9.2, 7.8, 9.2}

    // Creamos indices para ordenar sin perder la relacion nombre-promedio
    indices := []int{0, 1, 2, 3}
    sort.Slice(indices, func(i, j int) bool {
        return promedios[indices[i]] > promedios[indices[j]]
    })

    fmt.Println("Ranking por promedio:")
    for lugar, idx := range indices {
        fmt.Printf("%d. %s: %.1f\n", lugar+1, nombres[idx], promedios[idx])
    }
}

```

El truco aqui es ordenar un slice de indices en lugar de ordenar los datos directamente. Asi mantenemos la relacion entre cada nombre y su promedio sin tener que crear una estructura mas compleja (en la siguiente unidad veremos como los structs hacen esto mas elegante). La funcion comparadora usa `>` para ordenar de mayor a menor. Si dos estudiantes empatan (Luis y Pedro con 9.2), `sort.Slice` no garantiza cual queda primero; si el orden de empate te importa, usa `sort.SliceStable`.

Ejemplo 9: Tabla de calificaciones con slice de slices

Vamos a representar las calificaciones de varios estudiantes en varias materias usando un slice de slices, calcular el promedio de cada estudiante y encontrar al mejor. Este ejemplo junta slices bidimensionales, ciclos anidados y el idioma de reservar capacidad.

```

package main

import "fmt"

func main() {
    estudiantes := []string{"Ana", "Luis", "Sofia"}
    // Cada fila son las calificaciones de un estudiante en 3 materias
    califs := [][]float64{
        {9.0, 8.5, 9.5},
        {7.0, 8.0, 7.5},
        {10.0, 9.5, 10.0},
    }

    mejorIdx := 0
    mejorProm := 0.0
    for i, fila := range califs {
        suma := 0.0
        for _, c := range fila {
            suma += c
        }
        prom := suma / float64(len(fila))
        fmt.Printf("%s: promedio %.2f\n", estudiantes[i], prom)
        if prom > mejorProm {
            mejorProm = prom
            mejorIdx = i
        }
    }
    fmt.Printf("\nMejor estudiante: %s con %.2f\n", estudiantes[mejorIdx], mejorProm)
}

```

La estructura `[][]float64` nos permite pensar en renglones y columnas: `califs[i]` es el estudiante `i`, `califs[i][j]` es su calificación en la materia `j`. El ciclo exterior recorre estudiantes y el interior suma sus calificaciones. Llevamos el mejor promedio visto hasta ahora en dos variables, un patrón simple que evita ordenar cuando solo necesitas el máximo. Si las filas tuvieran distinta longitud (algun estudiante con materias extra), el código seguiría funcionando porque cada fila se mide con su propio `len`.

Ejemplo 10: Borrar un elemento por posición

Aquí aplicamos el idioma de borrado con `append(s[:i], s[i+1:]...)` en un caso práctico: quitar un producto agotado de un inventario por su índice, con validación del índice para no romper el programa.


```

package main

import "fmt"

// elimina borra el elemento en la posicion i y devuelve el slice resultante.
// Si i es invalido, devuelve el slice sin cambios.
func elimina(s []string, i int) []string {
    if i < 0 || i >= len(s) {
        return s
    }
    return append(s[:i], s[i+1:]...)
}

func main() {
    inventario := []string{"tortillas", "huevos", "leche", "pan"}
    fmt.Println("Antes:", inventario)

    inventario = elimina(inventario, 2) // se acabo la leche
    fmt.Println("Despues:", inventario) // [tortillas huevos pan]

    inventario = elimina(inventario, 10) // indice invalido: no pasa nada
    fmt.Println("Intento invalido:", inventario)
}

```

Dos lecciones en uno. Primera: validar el indice antes de rebanar evita el panico por "slice bounds out of range". Segunda: la funcion devuelve el slice resultante y el llamador lo asigna, siguiendo la misma disciplina que con `append`. Nota que `elimina` modifica el arreglo subyacente del llamador (los elementos se recorren a la izquierda), asi que si otro slice compartia ese arreglo, también vera el cambio. Para un inventario pequeno esto es aceptable; para datos compartidos, primero haz una copia.

Ejemplo 11: Las 3 palabras mas frecuentes

Combinemos todo: contar palabras con un mapa (ejemplo 5), pasar los resultados a un slice, y ordenarlo con `sort.Slice` para obtener un ranking. Es el tipo de programa que aparece en analisis de texto reales.

```

package main

import (
    "fmt"
    "sort"
    "strings"
)

type conteo struct {
    palabra string
    veces   int
}

func topPalabras(texto string, n int) []conteo {
    frec := make(map[string]int)
    for _, p := range strings.Fields(texto) {
        frec[p]++
    }

    // Pasamos el mapa a un slice para poder ordenarlo
    lista := make([]conteo, 0, len(frec))
    for palabra, veces := range frec {
        lista = append(lista, conteo{palabra, veces})
    }

    sort.Slice(lista, func(i, j int) bool {
        return lista[i].veces > lista[j].veces
    })

    if n > len(lista) {
        n = len(lista)
    }
    return lista[:n]
}

func main() {
    texto := "el sol sale y el sol se pone y vuelve a salir el sol"
    top := topPalabras(texto, 3)
    for i, c := range top {
        fmt.Printf("%d. %s (%d veces)\n", i+1, c.palabra, c.veces)
    }
}

```

Usamos un struct pequeño ``conteo`` para mantener junta cada palabra con su frecuencia (los structs los veremos a fondo en la siguiente unidad; por ahora basta saber que agrupan campos con nombre). El flujo es: mapa para contar, slice para ordenar, rebanado ``lista[:n]`` para quedarnos con el top. La proteccion ``if n > len(lista)`` evita pedir mas elementos de los que hay. Este patron de tres pasos (contar con mapa, volcar a slice, ordenar) resuelve una familia entera de problemas: rankings, histogramas, reportes de frecuencia.

Practica

Es hora de ensuciarte las manos. Los ejercicios estan ordenados de menor a mayor dificultad. Intenta resolver cada uno por tu cuenta antes de mirar la solucion: la frustracion de atorarte diez minutos vale mas que leer diez soluciones. Si te atoras por mas de veinte minutos, mira la solucion, entiendela, cierrala, y vuélvela a escribir desde cero sin copiar.

Ejercicio 1: Invertir un slice

Escribe un programa que tenga un slice de enteros y lo invierta en el lugar, es decir, sin crear un slice nuevo. Por ejemplo, ``[1, 2, 3, 4, 5]`` debe quedar como ``[5, 4, 3, 2, 1]``.

Pista: usa dos indices, uno que avance desde el principio y otro que retroceda desde el final, e intercambia los elementos hasta que se crucen.

Solucion:

```

package main

import "fmt"

func invertir(s []int) {
    izq, der := 0, len(s)-1
    for izq < der {
        s[izq], s[der] = s[der], s[izq]
        izq++
        der--
    }
}

func main() {
    numeros := []int{1, 2, 3, 4, 5}
    invertir(numeros)
    fmt.Println(numeros) // [5 4 3 2 1]
}

```

Explicacion: la funcion `invertir` recibe el slice y lo modifica en el lugar porque el slice comparte el arreglo subyacente con el original de `main`. El intercambio multiple `s[izq], s[der] = s[der], s[izq]` es un idioma de Go que evita la variable temporal. El ciclo se detiene cuando los indices se cruzan (`izq < der`), asi que el elemento del centro en un slice de longitud impar se queda quieto, lo cual es correcto. Este algoritmo es $O(n)$ en tiempo y $O(1)$ en memoria extra: lo optimo.

Ejercicio 2: Eliminar duplicados

Dado un slice de enteros que puede contener valores repetidos, escribe una funcion que devuelva un slice nuevo con los duplicados eliminados, conservando el orden de primera aparicion. Por ejemplo, `[3, 1, 3, 2, 1, 4]` debe devolver `[3, 1, 2, 4]`.

Pista: usa un mapa como conjunto de "ya vistos".

Solucion:

```

package main

import "fmt"

func sinDuplicados(s []int) []int {
    vistos := make(map[int]bool)
    resultado := make([]int, 0, len(s)) // capacidad reservada: optimizacion
    for _, v := range s {
        if !vistos[v] {
            vistos[v] = true
            resultado = append(resultado, v)
        }
    }
    return resultado
}

func main() {
    entrada := []int{3, 1, 3, 2, 1, 4}
    fmt.Println(sinDuplicados(entrada)) // [3 1 2 4]
    fmt.Println(entrada)                // [3 1 3 2 1 4] sin cambios
}

```

Explicacion: el mapa `vistos` funciona como un conjunto: la clave es el numero y el valor booleano solo indica presencia. Podriamos usar `map[int]struct{}` para ahorrar memoria, pero `map[int]bool` es mas legible para empezar. La coma-ok no es necesaria aqui porque el valor cero de `bool` es `false`, que significa "no visto". Reservamos la capacidad con `make([]int, 0, len(s))` porque en el peor caso (sin duplicados) el resultado tendra el mismo tamaño que la entrada; así evitamos realojamientos de memoria. Nota que la entrada no se modifica: construimos un slice nuevo, lo cual es mas seguro que filtrar en el lugar.

Ejercicio 3: El segundo mas grande

Escribe una funcion que reciba un slice de enteros y devuelva el segundo valor mas grande. Si el slice tiene menos de dos elementos distintos, la funcion debe indicarlo. Por ejemplo, para `[5, 9, 3, 9, 7]` el resultado es 7 (el 9 repetido no cuenta dos veces).

Pista: lleva registro del mayor y del segundo mayor en una sola pasada, o elimina duplicados primero y ordena.

Solucion:

```

package main

import (
    "fmt"
    "sort"
)

func segundoMayor(s []int) (int, bool) {
    // Primero eliminamos duplicados con un mapa
    unicos := make(map[int]bool)
    for _, v := range s {
        unicos[v] = true
    }
    if len(unicos) < 2 {
        return 0, false
    }
    // Pasamos las claves a un slice y ordenamos
    vals := make([]int, 0, len(unicos))
    for v := range unicos {
        vals = append(vals, v)
    }
    sort.Ints(vals)
    return vals[len(vals)-2], true
}

func main() {
    fmt.Println(segundoMayor([]int{5, 9, 3, 9, 7})) // 7 true
    fmt.Println(segundoMayor([]int{4, 4, 4}))      // 0 false
    fmt.Println(segundoMayor([]int{10}))           // 0 false
}

```

Explicacion: la funcion devuelve dos valores: el resultado y un booleano que indica si el calculo fue posible. Este patron de "valor + ok" es el hermano de la coma-ok de los mapas y es muy idiomatico en Go para funciones que pueden fallar sin que sea un error grave. Eliminamos duplicados con un mapa (reutilizando la idea del ejercicio 2), verificamos que queden al menos dos valores distintos, y ordenamos para tomar el penultimo. Ordenar es $O(n \log n)$; existe una solucion $O(n)$ con una sola pasada llevando los dos maximos, pero esta version es mas legible y para la mayoria de los casos es suficiente. En una entrevista, mencionar ambas opciones suma puntos.

Ejercicio 4: Agrupar palabras por su inicial

Dado un slice de palabras, construye un `map[string][]string` donde cada clave es la primera letra y el valor es el slice de palabras que empiezan con esa letra. Por ejemplo, `["ana", "pedro", "alicia", "pablo"]` produce `map[a:[ana alicia] p:[pedro pablo]]`.

Pista: la primera letra de una palabra `p` es `p[0:1]` si son letras ASCII; para hacerlo bien con runes, convierte a `[]rune` primero.

Solucion:

```
package main

import (
    "fmt"
    "sort"
)

func agruparPorInicial(palabras []string) map[string][]string {
    grupos := make(map[string][]string)
    for _, p := range palabras {
        if p == "" {
            continue // ignoramos cadenas vacias para no romper el programa
        }
        inicial := string([]rune(p)[0]) // primera runa, segura para UTF-8
        grupos[inicial] = append(grupos[inicial], p)
    }
    return grupos
}

func main() {
    palabras := []string{"ana", "pedro", "alicia", "pablo", "niño", "nube"}
    grupos := agruparPorInicial(palabras)

    // Mostramos ordenado por inicial
    iniciales := make([]string, 0, len(grupos))
    for ini := range grupos {
        iniciales = append(iniciales, ini)
    }
    sort.Strings(iniciales)
    for _, ini := range iniciales {
        fmt.Printf("%s: %v\n", ini, grupos[ini])
    }
}
```

Explicacion: la linea clave es `grupos[inicial] = append(grupos[inicial], p)``. Si la clave no existe, `grupos[inicial]` devuelve nil, y `append`` sobre nil crea un slice nuevo: funciona desde la primera palabra de cada grupo sin necesidad de inicializar. Este es uno de los idiomas mas elegantes de Go con mapas. Usamos `[]rune(p)[0]` para obtener la primera letra porque con `p[0:1]` romperiamos palabras que empiezan con ``ñ`` o vocales acentuadas (ocupan 2 bytes). Y protegemos contra cadenas vacias con `continue``, porque indexar una cadena vacia causaria un panico. Programar a la defensiva en los bordes (entradas vacias, nil) es lo que distingue el codigo de juguete del codigo de produccion.

Ejercicio 5: Fusionar dos slices ordenados

Dados dos slices de enteros ya ordenados de menor a mayor, escribe una funcion que los fusione en un slice nuevo también ordenado, sin usar `sort``. Por ejemplo, `[1, 3, 5]` y `[2, 4, 6, 8]` producen `[1, 2, 3, 4, 5, 6, 8]`.

Pista: usa dos indices, uno por slice, y en cada paso toma el menor de los dos elementos apuntados. Cuando un slice se agote, agrega el resto del otro.

Solucion:


```

package main

import "fmt"

func fusionar(a, b []int) []int {
    resultado := make([]int, 0, len(a)+len(b))
    i, j := 0, 0
    for i < len(a) && j < len(b) {
        if a[i] <= b[j] {
            resultado = append(resultado, a[i])
            i++
        } else {
            resultado = append(resultado, b[j])
            j++
        }
    }
    // Uno de los dos slices se agoto; agregamos lo que quede del otro
    resultado = append(resultado, a[i:]...)
    resultado = append(resultado, b[j:]...)
    return resultado
}

func main() {
    a := []int{1, 3, 5}
    b := []int{2, 4, 6, 8}
    fmt.Println(fusionar(a, b)) // [1 2 3 4 5 6 8]
}

```

Explicacion: este es el paso de "merge" del algoritmo MergeSort, y es un ejercicio clasico de entrevistas. La idea es avanzar con dos dedos, uno por cada slice, tomando siempre el menor disponible. Como ambos slices ya vienen ordenados, el menor global siempre esta en una de las dos puntas. Cuando un slice se agota, el resto del otro ya esta ordenado y se puede agregar en bloque con `append(resultado, a[i:]...)`. Reservamos la capacidad exacta desde el principio (`len(a)+len(b)`) para que no haya ningun realojamiento. Complejidad: $O(n+m)$ en tiempo, optimo para este problema.

Ejercicio 6: Frecuencia de caracteres en español

Escribe un programa que reciba un texto en espanol (con acentos y enies) y devuelva un mapa con la frecuencia de cada caracter, ignorando los espacios. Debe contar correctamente caracteres como ``ñ`` y vocales acentuadas. Pruebalo con la frase "el niño mimado come piña".

Pista: recorre con ``range`` para obtener runes, no bytes. La clave del mapa debe ser ``rune``.

Solucion:

```
package main

import (
    "fmt"
    "sort"
)

func frecuencia(texto string) map[rune]int {
    frec := make(map[rune]int)
    for _, r := range texto {
        if r == ' ' {
            continue
        }
        frec[r]++
    }
    return frec
}

func main() {
    texto := "el niño mimado come piña"
    frec := frecuencia(texto)

    // Ordenamos las runes para mostrar en orden
    runas := make([]rune, 0, len(frec))
    for r := range frec {
        runas = append(runas, r)
    }
    sort.Slice(runas, func(i, j int) bool { return runas[i] < runas[j] })

    for _, r := range runas {
        fmt.Printf("%c': %d\n", r, frec[r])
    }
}
```

Explicacion: este ejercicio junta dos temas de la unidad: mapas para contar y runes para texto en español. Si hubieramos recorrido por bytes, la `ñ` aparecería como dos entradas basura en lugar de una sola `ñ: 2`. La clave del mapa es de tipo `rune` (que en el fondo es un entero de 32 bits), así que cada carácter Unicode completo es una clave distinta. El `frec[r]++` aprovecha que el valor cero para una clave nueva es 0. Para mostrar ordenado, extraemos las claves, las ordenamos con `sort.Slice` y una comparadora simple, e imprimimos con `%c` para ver el carácter en lugar de su número. Si ejecutas el programa veras `ñ: 2` y `i: 1` contadas correctamente: la prueba de que el manejo de UTF-8 quedó bien.

Ejercicio 7: Interseccion de dos listas

Escribe una función que reciba dos slices de strings y devuelva un slice con los elementos que aparecen en ambas listas, sin duplicados. Por ejemplo, `["ana", "luis", "pedro"]` y `["luis", "sofia", "ana", "ana"]` devuelven `["ana", "luis"]` (el orden puede variar, pero mejor si conserva el orden de la primera lista).

Pista: mete los elementos de la segunda lista en un mapa y recorre la primera preguntando por cada elemento.

Solucion:

```

package main

import "fmt"

func interseccion(a, b []string) []string {
    enB := make(map[string]bool)
    for _, v := range b {
        enB[v] = true
    }

    yaAgregados := make(map[string]bool)
    resultado := []string{}
    for _, v := range a {
        if enB[v] && !yaAgregados[v] {
            yaAgregados[v] = true
            resultado = append(resultado, v)
        }
    }
    return resultado
}

func main() {
    a := []string{"ana", "luis", "pedro"}
    b := []string{"luis", "sofia", "ana", "ana"}
    fmt.Println(interseccion(a, b)) // [ana luis]
}

```

Explicacion: convertimos la lista `b` en un conjunto (mapa) para poder preguntar "existe este elemento?" en tiempo constante. Luego recorremos `a` en orden: si el elemento esta en `b` y aun no lo agregamos, lo agregamos. El segundo mapa `yaAgregados` evita duplicados en el resultado cuando `a` trae repetidos. Sin los mapas, tendríamos que comparar cada elemento de `a` contra cada elemento de `b`, un algoritmo cuadrático $O(n*m)$ que se vuelve lento con listas grandes. Con mapas es $O(n+m)$: lineal. Esta diferencia entre "funciona" y "funciona rapido" es exactamente el tipo de criterio que evalúan en entrevistas técnicas, y nace de elegir la estructura de datos correcta, que es el tema de esta unidad.

Ejercicio 8: Rotar un slice a la derecha

Escribe una función que rote los elementos de un slice `k` posiciones a la derecha, en el lugar. Por ejemplo, `[1, 2, 3, 4, 5]` rotado 2 posiciones queda `[4, 5, 1, 2, 3]`. Si `k` es mayor que la longitud, debe funcionar igual (rotar 7 posiciones un slice de 5 equivale a rotar 2).

Pista: hay un truco elegante con tres inversiones: invierte todo el slice, invierte los primeros `k`, invierte el resto. Comprueba con papel y lapiz por que funciona.

Solucion:

```
package main

import "fmt"

func invertir(s []int) {
    for i, j := 0, len(s)-1; i < j; i, j = i+1, j-1 {
        s[i], s[j] = s[j], s[i]
    }
}

func rotarDerecha(s []int, k int) {
    n := len(s)
    if n == 0 {
        return
    }
    k = k % n // normaliza: rotar n es no rotar
    if k == 0 {
        return
    }
    invertir(s)      // paso 1: invierte todo
    invertir(s[:k])  // paso 2: invierte los primeros k
    invertir(s[k:])  // paso 3: invierte el resto
}

func main() {
    s := []int{1, 2, 3, 4, 5}
    rotarDerecha(s, 2)
    fmt.Println(s) // [4 5 1 2 3]

    s2 := []int{1, 2, 3, 4, 5}
    rotarDerecha(s2, 7) // 7 % 5 == 2
    fmt.Println(s2) // [4 5 1 2 3]
}
```

Explicacion: el truco de las tres inversiones es uno de esos algoritmos que parecen magia hasta que los entiendes. Al invertir todo el slice, los `k` elementos que deben quedar al frente terminan al principio pero en orden inverso; invertir esos `k` los deja en su orden correcto; e invertir el resto hace lo mismo con los demas. El resultado es una rotacion perfecta en $O(n)$ de tiempo y $O(1)$ de memoria extra. La normalizacion `k = k % n` maneja rotaciones mayores que

la longitud, y los casos borde (slice vacio, `k == 0`) se atienden al inicio para no dividir entre cero ni trabajar de mas. Reutilizamos la funcion `invertir` del ejercicio 1: en Go es comun construir soluciones componiendo funciones pequenas que ya probaste, en lugar de reescribir la logica. Cada funcion pequena y probada es un ladrillo confiable para la siguiente.

Resumen de la unidad

Llegamos al final de la unidad. Aqui tienes todo lo esencial condensado para repaso rapido.

Arreglos. Secuencia de tamano fijo donde el tamano forma parte del tipo: `[5]int` y `[6]int` son tipos distintos. Se declaran con `var a [5]int` o con literales como `[5]int{1, 2, 3, 4, 5}`; `[...]int{...}` deja que el compilador cuente. Los indices empiezan en cero y salirse del rango detiene el programa. Se pasan a funciones por valor (se copian). Usalos cuando el tamano es fijo por naturaleza: dias de la semana, coordenadas, cabeceras de protocolo.

Slices. La estructura de trabajo diaria en Go: una vista sobre un arreglo subyacente descrita por puntero, longitud y capacidad. Se crean con literales (`[]int{...}`), con `make([]int, len)` o `make([]int, len, cap)`, o rebanando (`s[i:j]`, `s[i:j:k]`). `len` es lo que hay, `cap` es lo que cabe. Rebanar no copia: comparte el arreglo subyacente.

append. Agrega elementos y devuelve el slice resultante: siempre asigna el resultado (`s = append(s, x)`). Si hay capacidad, escribe en el arreglo existente; si no, pide memoria nueva, copia todo y devuelve un slice que apunta al bloque nuevo. `append(s, otros...)` agrega un slice completo. Para clonar: `append([]T(nil), s...)`.

copy. `copy(destino, origen)` copia elementos y devuelve cuantos copio; no crece el destino. Después de copiar, los slices son independientes.

La trampa del arreglo compartido. Hacer `append` sobre un slice rebanado puede modificar el slice original si habia capacidad disponible. Protegete con la expresion de rebanado completa `s[i:j:j]` (capacidad limitada) o copiando antes de modificar. Nunca asumas que un slice rebanado es independiente.

nil vs vacio. `var s []int` es nil; `[]int{}` y `make([]int, 0)` son vacios. Casi todo funciona igual en los tres (`len`, `range`, `append`). Prefiere nil por simplicidad, salvo que necesites distinguir en JSON (`null` vs `[]`).

Strings y runes. Un string es un slice de bytes de solo lectura. `len(s)` cuenta bytes, no caracteres. Los caracteres en espanol (acentos, ñ, ç, ÿ) ocupan 2 o mas bytes en UTF-8. Recorre con `range` para obtener runes completos, usa `utf8.RuneCountInString` para contar caracteres, y convierte a `[]rune` para modificar letras individuales.

Mapas. Pares clave-valor con busqueda rapida. Se crean con `make(map[K]V)` o literales; un mapa nil permite leer pero no escribir. Insertar y actualizar: `m[k] = v`. Leer con coma-ok: `v, ok := m[k]`. Borrar: `delete(m, k)`. Tamano: `len(m)`. Recorrer con `range`, pero el orden es aleatorio a proposito: si necesitas orden, extrae las claves, ordenalas con `sort` y recorre. Las claves deben ser comparables (no slices).

Idiomas que debes dominar. Contar con `m[k]++`. Agrupar con `m[k] = append(m[k], v)`. Conjunto de vistos con `map[T]bool`. Funciones que devuelven `(valor, ok)`.

Cuando usar cada una. Arreglo: tamano fijo conocido en compilacion. Slice: secuencias de tamano variable, el caso general. Mapa: busqueda por clave, conteos, agrupaciones, indices. Combinalas: `map[string][]string`, `[]map[string]int`.

sort. `sort.Ints`, `sort.Strings`, `sort.Float64s` ordenan en el lugar. `sort.Slice(s, func(i, j int) bool {...})` ordena con criterio propio; `sort.SliceStable` conserva el orden previo en empates.

En la siguiente unidad subiremos un nivel de abstraccion: los structs, que nos permiten agrupar datos de distintos tipos bajo un mismo nombre, y los metodos, que nos permiten darles comportamiento. Con arreglos, slices, mapas y structs ya tendras el kit completo de estructuras de datos de Go.

Hoja de referencia rapida: los idiomas que debes saber de memoria

```

// --- Slices ---
var s []int // slice nil, listo para append
s = append(s, 1, 2, 3) // agregar (siempre asignar el resultado)
s = append(s, otro...) // agregar un slice completo
t := make([]int, 5) // 5 ceros
t = make([]int, 0, 100) // vacio, capacidad para 100
sub := s[1:4] // rebanada: indices 1,2,3 (comparte arreglo)
seg := s[1:4:4] // rebanada con capacidad limitada (segura)
dst := make([]int, len(s)) // destino para copiar
copy(dst, s) // copia independiente
clon := append([]int(nil), s...) // clon en una linea
s = append(s[:i], s[i+1:]...) // borrar el elemento i

// --- Strings / runes ---
n := len(texto) // bytes, NO caracteres
n = utf8.RuneCountInString(texto) // caracteres reales
for _, r := range texto { } // recorrido correcto por runes
runas := []rune(texto) // para modificar letras
deNuevo := string(runas) // de vuelta a string

// --- Mapas ---
m := make(map[string]int) // mapa listo para escribir
m["ana"] = 25 // insertar / actualizar
v := m["ana"] // leer (0 si no existe)
v, ok := m["ana"] // leer con coma-ok
delete(m, "ana") // borrar (no falla si no existe)
n := len(m) // numero de entradas
for k, v := range m { } // recorrer (orden aleatorio)
m[k]++ // contar ocurrencias
m[k] = append(m[k], v) // agrupar valores por clave

// --- Ordenar ---
sort.Ints(nums) // ordena en el lugar
sort.Strings(nombres) // ordena en el lugar
sort.Slice(datos, func(i, j int) bool {
    return datos[i] < datos[j] // criterio propio
})

```

Copia esta hoja en tus notas. Cuando lleves unas semanas programando en Go, la miraras cada vez menos, hasta que un día descubras que ya la sabes de memoria. Ese día, esta unidad habra cumplido su mision.

UNIDAD 6: Structs y métodos

En la unidad anterior aprendimos a agrupar muchos valores del mismo tipo: listas de numeros, diccionarios de nombres. Pero el mundo real no esta hecho solo de listas homogeneas. Piensa en un jugador de videojuegos: tiene un nombre (texto), un nivel (numero), una cantidad de vida (numero), y una lista de objetos en su inventario (slice). O en una cuenta bancaria: tiene un titular, un numero de cuenta, un saldo, y un historial de movimientos. Estos datos son de tipos distintos, pero pertenecen a la misma entidad. Necesitamos una forma de empaquetarlos juntos, con nombre y apellido, para tratarlos como una sola cosa.

Esa forma es el struct. Un struct es un tipo compuesto que agrupa campos con nombre, cada uno con su propio tipo. Es el bloque de construccion fundamental para modelar el mundo real en Go: usuarios, productos, pedidos, configuraciones, respuestas de un servidor. Si los slices y mapas de la unidad anterior eran colecciones de valores sueltos, los structs son los contenedores con etiqueta que les dan significado.

Pero un struct solo con datos es como un coche sin motor: guarda cosas, pero no hace nada. Por eso esta unidad tiene dos mitades inseparables: los structs (los datos) y los metodos (el comportamiento). Un metodo es una funcion atada a un tipo, que le da capacidades: una cuenta bancaria puede depositar y retirar, un jugador puede subir de nivel, un rectangulo puede calcular su area. La combinacion de structs con metodos es como Go hace programacion orientada a objetos, aunque Go deliberadamente evita esa etiqueta y hace las cosas a su manera: sin clases, sin herencia, sin constructores magicos. Solo tipos, funciones y una filosofia clara.

Esa filosofia se resume en una frase que escucharas en toda la comunidad Go: "acepta interfaces, devuelve structs". Significa que tus funciones deben ser flexibles con lo que reciben (cualquier cosa que cumpla un comportamiento) y concretas con lo que entregan (un tipo especifico y documentado). La profundizaremos al final de la teoria, porque es la idea que une todo lo que veremos: composicion sobre herencia, tipos pequenos y claros, comportamiento explicito.

Esta unidad sigue la misma estructura que la anterior: explicacion detallada con analogias y errores comunes, ejemplos paso a paso con codigo compilable (incluyendo el ejemplo completo de la CuentaBancaria que pide el temario), ejercicios con soluciones, y un resumen final. Al terminar, sabras modelar cualquier entidad del mundo real en Go y darle comportamiento de forma idiomática. Es el paso que convierte tus programas de "scripts que procesan listas" en "sistemas que representan cosas".

Un aviso: si vienes de lenguajes como Java, Python o JavaScript, tendras que desaprender un poco. Go no tiene clases, y eso no es una carencia: es una decision de diseno. Los structs con metodos cubren el 95% de lo que las clases hacen, con menos magia y menos sorpresas. Los que intentan escribir "Java con sintaxis de Go" terminan peleando con el lenguaje; los que aceptan la filosofia de Go terminan escribiendo codigo mas simple y mas facil de mantener. Mantente abierto y veras que el enfoque minimalista tiene su encanto.

Explicacion detallada

Structs: la ficha tecnica de una entidad

Imagina la ficha de un jugador en un videojuego de rol: nombre, clase, nivel, puntos de vida, fuerza. Cada dato tiene su tipo, pero todos describen al mismo personaje. Un struct es exactamente eso: una ficha tecnica donde cada campo tiene nombre y tipo.

```
type Jugador struct {  
    Nombre string  
    Nivel   int  
    Vida    int  
}
```

La palabra `type` crea un tipo nuevo llamado `Jugador`, y `struct{...}` dice que esta compuesto por esos campos. Por convencion, los nombres de tipos empiezan con mayuscula, y los campos que empiezan con mayuscula son exportados (visibles desde otros paquetes), mientras que los que empiezan con minuscula son privados del paquete. Esta regla de mayusculas y minusculas es el sistema de visibilidad de Go: simple y sin palabras clave extra.

Para crear un valor del struct (una instancia, en vocabulario de otros lenguajes) hay varias formas. La mas comun es el literal:

```
j := Jugador{  
    Nombre: "Aria",  
    Nivel:  5,  
    Vida:   100,  
}
```

Nombrar los campos en el literal es la forma recomendada: el codigo se lee solo y no se rompe si mas adelante agregas o reordenas campos. Existe la forma corta sin nombres (`Jugador{"Aria", 5, 100}`), pero es fragil: si alguien agrega un campo en medio, todos los

literales cortos se rompen o, peor, asignan valores a campos equivocados sin que el compilador se queje (si los tipos coinciden). En código profesional veras casi siempre la forma con nombres.

También puedes declarar una variable del tipo y asignar campo por campo:

```
var j Jugador      // todos los campos valen cero: "", 0, 0
j.Nombre = "Aria"
j.Nivel = 5
j.Vida = 100
```

Y puedes crear un puntero al struct con `new`:

```
p := new(Jugador) // p es *Jugador, apunta a un Jugador en ceros
p.Nombre = "Aria" // Go permite p.Nombre en lugar de (*p).Nombre
```

`new(T)` reserva memoria para un valor de tipo T en ceros y te devuelve su dirección. Es equivalente a `&Jugador{}`. La última línea muestra una comodidad de Go: si `p` es un puntero a struct, puedes escribir `p.Nombre` y el compilador entiende `(\*p).Nombre` automáticamente. No necesitas desreferenciar a mano.

Acceder a los campos es con el punto, igual que en casi todos los lenguajes: `j.Nombre`, `j.Vida`. Y puedes copiar structs con una simple asignación: `copia := j`. Pero atención: si el struct contiene un slice o un mapa, la copia comparte esos datos internos (recuerda la unidad 5: los slices son vistas sobre un arreglo compartido). Copiar un struct copia los campos, no lo que apuntan sus campos.

Campos anónimos y embedding: composición desde el día uno

Go no tiene herencia, pero tiene algo más flexible: el embedding. Puedes incluir un tipo dentro de un struct sin darle nombre de campo, y sus campos y métodos se "promocionan" al struct exterior, como si fueran propios.

```

type Persona struct {
    Nombre string
    Edad    int
}

type Empleado struct {
    Persona      // campo anonimo: embedding
    Sueldo float64
}

```

Ahora un `Empleado` tiene `Nombre` y `Edad` directamente accesibles:

```

e := Empleado{
    Persona: Persona{Nombre: "Luis", Edad: 30},
    Sueldo:  15000,
}
fmt.Println(e.Nombre) // "Luis": campo promocionado
fmt.Println(e.Persona.Edad) // también funciona la ruta completa

```

Esto no es herencia: `Empleado` no "es un" `Persona` en el sentido de subtipos. Es composicion con azucar sintactica: el `Empleado` "tiene un" `Persona` cuyos miembros se promocionan para comodidad. La diferencia importa cuando hablamos de interfaces (un `Empleado` satisface las interfaces que `Persona` satisface, gracias a la promocion de metodos), pero no hay jerarquias de clases ni polimorfismo por herencia. Si vienes de Java o C++, piensa en el embedding como "herencia de implementacion sin herencia de tipos": reutilizas codigo sin crear jerarquias rigidas.

El embedding también funciona con tipos que no son structs, incluyendo interfaces, pero eso lo veremos cuando hablemos de interfaces en otra unidad. Por ahora quedate con la idea: en Go se comparte comportamiento componiendo, no heredando.

Metodos: funciones con dueño

Un metodo es una funcion que pertenece a un tipo. Se declara con un receptor entre la palabra `func` y el nombre:

```

func (j Jugador) Saludar() string {
    return "Hola, soy " + j.Nombre + " nivel " + string(rune(j.Nivel))
}

```

`(j Jugador)` es el receptor: dice que `Saludar` es un metodo del tipo `Jugador`, y dentro del metodo `j` se refiere al valor sobre el que se llamo. Se usa asi:

```
j := Jugador{Nombre: "Aria", Nivel: 5, Vida: 100}
fmt.Println(j.Saludar())
```

La diferencia entre una funcion y un metodo es solo el receptor, pero conceptualmente es enorme: los metodos agrupan el comportamiento junto a los datos. En lugar de tener una funcion suelta `calcularArea(rectangulo)`, tienes `rectangulo.Area()`. El codigo se lee como una narracion: el sujeto hace la accion.

Receptor por valor vs receptor por puntero: la decision mas importante

Aqui esta el tema que mas preguntas genera. El receptor puede ser por valor `(j Jugador)` o por puntero `(j \*Jugador)`. La diferencia es la misma que entre pasar un argumento por valor o por referencia:

- Receptor por valor: el metodo recibe una copia del struct. Si modifica campos, el original no cambia. Es seguro pero no sirve para metodos que deben modificar el estado.
- Receptor por puntero: el metodo recibe la direccion del struct original. Puede modificarlo, y evita copiar structs grandes.

```
// Por valor: no modifica el original
func (j Jugador) SubirNivelMal() {
    j.Nivel++ // modifica la copia, se pierde al salir
}

// Por puntero: si modifica el original
func (j *Jugador) SubirNivel() {
    j.Nivel++ // modifica el struct real
}
```

Si llamas `j.SubirNivelMal()`, el nivel de `j` no cambia: el metodo trabajo sobre una copia desechable. Con `j.SubirNivel()`, el cambio persiste. Go te ayuda con la sintaxis: aunque `SubirNivel` tiene receptor de puntero, puedes llamarlo sobre una variable normal `j` (no necesitas `(&j).SubirNivel()`); el compilador toma la direccion automaticamente siempre que la variable sea direccionable.

La regla practica que usa la comunidad Go es sencilla. Usa receptor por puntero cuando:

1. El metodo necesita modificar el receptor (Depositar, Retirar, SubirNivel).

2. El struct es grande y copiarlo en cada llamada seria costoso.
3. El tipo contiene un campo que no debe copiarse, como un ``sync.Mutex`` (los veras en concurrencia).

Usa receptor por valor cuando:

1. El metodo no modifica nada y el struct es pequeno (un Punto con X, Y; un Rectangulo).
2. Quieres garantizar que el metodo no puede modificar el original.
3. El tipo es inmutable por diseno.

Y la regla de oro de consistencia: si algun metodo de un tipo usa receptor por puntero, todos deberian usarlo. Mezclar receptores por valor y por puntero en el mismo tipo es confuso y puede causar errores sutiles (por ejemplo, al guardar valores en interfaces, que veremos después). Elige uno y mantenlo en todo el tipo.

Un error clasico del principiante: definir ``Depositar`` con receptor por valor y luego no entender por que el saldo no cambia. Si tu metodo modifica estado y "no funciona", lo primero que debes revisar es el receptor. El compilador no te avisa porque tecnicamente el codigo es valido: simplemente modifica una copia.

Constructores por convencion: la funcion New

Go no tiene constructores como los de otros lenguajes (no hay palabra ``new`` magica ni metodos especiales). En su lugar, la convencion es escribir una funcion ``NewTipo`` que crea y devuelve el valor inicializado:

```
type CuentaBancaria struct {
    titular string
    saldo   float64
}

func NuevaCuenta(titular string, depositoInicial float64) *CuentaBancaria {
    return &CuentaBancaria{
        titular: titular,
        saldo:   depositoInicial,
    }
}
```

Observa varias convenciones en accion. La funcion se llama ``NuevaCuenta`` (en codigo en ingles seria ``NewAccount``); devuelve un puntero porque los metodos de la cuenta usaran receptor por puntero. Los campos son privados (minuscula): nadie desde fuera del paquete

puede tocar ``saldo`` directamente, lo cual nos obliga a pasar por los metodos ``Depositar`` y ``Retirar`` que validan las operaciones. Esto es encapsulamiento al estilo Go: no con palabras clave ``private``, sino con mayusculas, minusculas y constructores que establecen invariantes.

La funcion ``New`` también es el lugar para validar: si el deposito inicial es negativo, puedes devolver un error o ajustarlo a cero. Y para valores por defecto: si el struct tiene diez campos pero el usuario normal solo configura dos, ``New`` pone los otros ocho en valores sensatos. Un buen constructor convierte la creacion de un valor valido en algo de una linea, imposible de hacer mal.

Structs comparables: cuando `==` funciona

Los structs se pueden comparar con ``==`` siempre que todos sus campos sean comparables. Dos structs son iguales si todos sus campos son iguales:

```
type Punto struct {
    X, Y int
}

p1 := Punto{1, 2}
p2 := Punto{1, 2}
fmt.Println(p1 == p2) // true
```

Esto es util para pruebas y para usar structs como claves de mapa. Pero si el struct contiene un slice, un mapa o una funcion, deja de ser comparable y el ``==`` causa un error de compilacion (o un panico en ejecucion si la comparacion ocurre dentro de una interfaz). Regla practica: si quieres comparar structs o usarlos como claves, manten sus campos en tipos comparables (numeros, strings, booleanos, arreglos de comparables, otros structs comparables).

Para structs no comparables, escribe tu propio metodo ``Igual``:

```
func (j Jugador) Igual(otro Jugador) bool {
    if j.Nombre != otro.Nombre || j.Nivel != otro.Nivel || j.Vida != otro.Vida {
        return false
    }
    return slices.Equal(j.Inventario, otro.Inventario)
}
```

Explicito y sin sorpresas: otra muestra de la filosofia Go.

Etiquetas de struct: metadatos con los tags

Los tags son anotaciones entre comillas invertidas que se pegan a los campos de un struct. No afectan al código directamente: son metadatos que otros paquetes leen por reflexión. El uso más común es con JSON:

```
type Usuario struct {  
    Nombre    string `json:"nombre"`  
    Edad      int     `json:"edad"`  
    Password  string `json:"-"`  
}
```

Cuando conviertes este struct a JSON con ``encoding/json``, los campos se llaman ``nombre`` y ``edad`` (en minúsculas, como piden las APIs), y ``Password`` se omite por completo gracias al ``-``. Sin tags, el JSON usaría los nombres de los campos tal cual (``Nombre``, ``Edad``), lo cual rompe las convenciones de casi todas las APIs.

```
u := Usuario{Nombre: "Ana", Edad: 28, Password: "secreto123"}  
datos, _ := json.Marshal(u)  
fmt.Println(string(datos)) // {"nombre":"Ana","edad":28}
```

La contraseña no aparece: el tag la protege. Los tags también se usan para bases de datos (``db:"nombre_columna"``), validación (``validate:"required"``), y configuración. Son texto libre con formato ``clave:"valor"``, y cada paquete define los suyos. Detalles importantes: van entre acentos graves (backticks), no comillas normales; y si un campo no tiene tag, se usan valores por defecto razonables.

Un error común es escribir el tag con comillas dobles en lugar de backticks: `` `json:"nombre"`` es correcto, `` `json:"nombre""`` no compila como tag (sería un string normal y el paquete json lo ignoraría silenciosamente, produciendo JSON con los nombres originales). Si tu JSON sale con mayúsculas inesperadas, revisa los backticks primero.

Structs anidados: structs dentro de structs

Un campo de un struct puede ser otro struct. Esto modela relaciones de "tiene un": una persona tiene una dirección, un pedido tiene un cliente, un curso tiene un profesor.


```

type Direccion struct {
    Calle string
    Ciudad string
    CP     string
}

type Persona struct {
    Nombre string
    Edad   int
    Direccion Direccion // struct anidado con nombre de campo
}

```

Se usa con la ruta completa:

```

p := Persona{
    Nombre: "Maria",
    Edad:   34,
    Direccion: Direccion{
        Calle: "Av. Juarez 123",
        Ciudad: "CDMX",
        CP:     "06600",
    },
}
fmt.Println(p.Direccion.Ciudad) // CDMX

```

La diferencia con el embedding es el nombre del campo: aquí el campo se llama `Direccion` y accedes con `p.Direccion.Ciudad`. Con embedding (`Persona` sin nombre dentro de `Empleado`), los campos se promocionan y accedes directo con `e.Nombre`. Usa struct anidado con nombre cuando la relación deba ser explícita ("la dirección de la persona"), y embedding cuando quieras reutilizar comportamiento de forma transparente ("el empleado actúa como persona").

Los structs anidados también aparecen al leer JSON de APIs reales: una respuesta con objetos dentro de objetos se modela con structs dentro de structs, cada uno con sus tags. Es el pan de cada día del desarrollo backend en Go.

Composicion vs herencia: la filosofía de Go

En lenguajes con clases, la reutilización se hace con herencia: "un Gerente es un Empleado, que es una Persona". Las jerarquías crecen, se enredan, y terminas con el problema del "gorila con la banana": pediste una banana (un método útil) y te dieron el gorila (toda la jerarquía) con todo y jungla.

Go propone otra cosa: composicion. En lugar de decir "X es un Y", dices "X tiene un Y" o "X sabe hacer Z". La reutilizacion viene de dos mecanismos:

1. Embedding de structs: reutilizas datos y metodos concretos sin crear jerarquias de tipos.
2. Interfaces: defines comportamientos ("sabe guardarse", "sabe dibujarse") y cualquier tipo que implemente esos metodos sirve, sin declarar parentesco.

Esto lleva a la frase que mencionamos al inicio: "acepta interfaces, devuelve structs". Cuando escribes una funcion, pide como parametros lo mas general que te sirva (una interfaz con los metodos que realmente usas), pero devuelve algo concreto (un struct o puntero a struct) para que quien te llama sepa exactamente que recibe. Asi tus funciones son flexibles por dentro y predecibles por fuera.

Un ejemplo de la libreria estandar: ``io.Copy`` acepta cualquier cosa que sepa leer y cualquier cosa que sepa escribir (interfaces), pero ``os.Create`` devuelve un ``*os.File`` concreto. Tu codigo puede copiar de un archivo a un buffer de red sin que ``io.Copy`` sepa que existen los archivos o la red. Esa es la potencia de disenar por comportamiento en lugar de por jerarquia.

En la practica, esto significa: structs pequenos, cada uno con una responsabilidad clara; metodos que operan sobre esos structs; embedding cuando necesites reutilizar implementacion; e interfaces (que veremos en la siguiente unidad) cuando necesites polimorfismo. Sin arboles de herencia, sin clases abstractas, sin constructores magicos. Menos conceptos, mas claridad.

Metodos o funciones sueltas: cuando usar cada uno

No todo debe ser un metodo. La pregunta para decidir es: esta operacion es parte del comportamiento natural del tipo, o es un algoritmo que casualmente usa el tipo? ``cuenta.Depositar(500)`` es comportamiento de la cuenta: metodo. ``generarReporteAnual(cuentas)`` es un proceso que opera sobre muchas cuentas: funcion suelta que recibe un slice.

Una senal practica: si la operacion necesita acceder a campos privados del struct, casi siempre debe ser un metodo (las funciones de otro paquete no pueden verlos). Si solo usa la parte publica, puede ser funcion. Otra senal: los metodos aparecen en la documentacion del tipo y tu editor los sugiere al escribir ``cuenta.``; las funciones sueltas hay que recordarlas. Pon como metodos lo que un usuario del tipo hara todos los dias; deja como funciones lo ocasional o lo que combina varios tipos.

Y un matiz importante: en Go los metodos no pueden definirse sobre tipos de otros paquetes. Si quieres darle comportamiento a un ``time.Time`` o a un ``[]string`` ajeno, no puedes agregarle

metodos directamente: defines tu propio tipo basado en el (como hicimos con `Celsius`) o escribes funciones sueltas. Esta restriccion evita que una libreria "contamine" los tipos de otra, y es otra expresion de la filosofia de mantener las cosas explicitas.

El struct vacio: el tipo que no pesa nada

`struct{}` es un tipo sin campos que ocupa cero bytes de memoria. Parece inutil, pero tiene dos usos elegantes. Como valor en un mapa cuando solo te importa la existencia de la clave (un conjunto matematico):

```
vistos := make(map[string]struct{})
vistos["ana"] = struct{}{}
if _, ok := vistos["ana"]; ok {
    fmt.Println("ana ya estaba")
}
```

Ocupa menos memoria que `map[string]bool` porque el valor no pesa nada. Y como senal en canales de concurrencia (tema de otra unidad): enviar un `struct{}` dice "paso algo" sin gastar memoria en el mensaje. No lo usaras todos los dias, pero cuando lo veas en codigo ajeno sabras que significa: "aqui solo importa la presencia, no el valor".

Convenciones de nombres que delatan al profesional

Los detalles pequenos separan el codigo Go idiomatico del traducido de otro lenguaje. El receptor de un metodo se nombra con una o dos letras del tipo: `(j Jugador)`, `(c \*CuentaBancaria)`, `(p Pedido)`. No se usa `this` ni `self` (no existen en Go) ni nombres largos como `(jugador Jugador)`. La razon es legibilidad: el receptor aparece en cada metodo del tipo, y un nombre corto reduce el ruido visual.

Los constructores se llaman `New` + nombre del tipo en el mismo paquete (`NuevaCuenta` en codigo en espanol, `NewAccount` en ingles). Si el constructor esta en otro paquete, se usa como `banco.NuevaCuenta(...)`: el nombre del paquete ya da contexto, por eso no se repite ("banco.NuevaCuentaBancaria" seria redundante).

Los metodos que devuelven un valor y un error nombran el error `err` al revisarlo, y las funciones que solo consultan empiezan con el nombre de lo que devuelven (`Saldo()`, `Total()`, `EstaVivo()`), no con `Get` (`GetSaldo()` es estilo Java, no Go). Los booleanos se nombran como preguntas: `EstaVivo()`, `Vacio()`, `EsAdmin`. Adoptar estas convenciones hace que tu codigo se lea como el de la libreria estandar, y eso es un cumplido en la comunidad Go.

Errores comunes con structs y metodos

1. Metodo que modifica con receptor por valor: el cambio se pierde en la copia. El error numero uno de esta unidad.
2. Mezclar receptores por valor y por puntero en el mismo tipo: funciona, pero confunde y rompe la consistencia.
3. Olvidar el `&` en `json.Unmarshal`: sin puntero, el struct no se llena y no hay error visible.
4. Tags con comillas dobles en lugar de backticks: el paquete los ignora en silencio.
5. Literal corto `Jugador{"Aria", 5, 100}` que se rompe (o asigna mal) cuando alguien agrega un campo.
6. Copiar un struct con slices o mapas internos creyendo que es copia profunda.
7. Campos privados que impiden crear el struct con literal desde otro paquete: si los campos son privados y no hay constructor exportado, el tipo es inutilizable desde fuera. Disena la visibilidad a proposito.
8. Comparar structs con `==` cuando tienen un slice: no compila. Usa un metodo `Igual` o `reflect.DeepEqual` en pruebas.

Si algo "deberia funcionar" y no funciona, esta lista es tu primer lugar de busqueda.

Ejemplos paso a paso

Ejemplo 1: Punto geometrico con metodos por valor

Empezamos con el caso mas simple: un struct pequeno e inmutable con metodos que no modifican nada. Un punto 2D con un metodo de distancia es el ejemplo canonico de receptor por valor.

```

package main

import (
    "fmt"
    "math"
)

type Punto struct {
    X, Y float64
}

// Distancia calcula la distancia euclidiana al origen.
// Receptor por valor: no modifica el punto, y el struct es pequeno.
func (p Punto) Distancia() float64 {
    return math.Sqrt(p.X*p.X + p.Y*p.Y)
}

// Mover devuelve un punto nuevo desplazado: no modifica el original.
func (p Punto) Mover(dx, dy float64) Punto {
    return Punto{X: p.X + dx, Y: p.Y + dy}
}

func main() {
    p := Punto{X: 3, Y: 4}
    fmt.Printf("Distancia al origen: %.2f\n", p.Distancia()) // 5.00

    q := p.Mover(1, 1)
    fmt.Println("Original:", p) // {3 4} sin cambios
    fmt.Println("Movido: ", q) // {4 5}
}

```

Observa el estilo: `Mover` no modifica el receptor, devuelve un punto nuevo. Esto hace al tipo inmutable en la practica: cada operacion produce un valor nuevo en lugar de cambiar el existente. Para tipos pequenos de tipo "valor" (puntos, colores, fechas), este estilo es limpio y evita toda una categoria de errores. Y `fmt.Println(p)` muestra `{3 4}` porque Go imprime los structs con sus campos entre llaves por defecto.

Ejemplo 2: Rectangulo con area y perimetro

Un rectangulo con metodos de calculo. Seguimos con receptor por valor porque los metodos solo leen.

```

package main

import "fmt"

type Rectangulo struct {
    Ancho, Alto float64
}

func (r Rectangulo) Area() float64 {
    return r.Ancho * r.Alto
}

func (r Rectangulo) Perimetro() float64 {
    return 2 * (r.Ancho + r.Alto)
}

// Escalar devuelve un rectangulo nuevo con las dimensiones multiplicadas.
func (r Rectangulo) Escalar(factor float64) Rectangulo {
    return Rectangulo{Ancho: r.Ancho * factor, Alto: r.Alto * factor}
}

func main() {
    r := Rectangulo{Ancho: 10, Alto: 5}
    fmt.Println("Area:", r.Area())           // 50
    fmt.Println("Perimetro:", r.Perimetro()) // 30

    grande := r.Escalar(2)
    fmt.Println("Original:", r)              // {10 5}
    fmt.Println("Escalado:", grande)         // {20 10}
}

```

Nota como los metodos hacen que el codigo cliente se lea como espanol: `r.Area()`, `r.Escalar(2)`. Compara con la alternativa procedural `Area(r)`, `Escalar(r, 2)`: funciona igual, pero los metodos agrupan el comportamiento con el tipo y tu editor puede autocompletarlos al escribir `r.`. Esa es la ganancia ergonomica de los metodos.

Ejemplo 3: CuentaBancaria completa (el ejemplo central)

Aqui esta el ejemplo completo que pide el temario: un struct `CuentaBancaria` con constructor por convencion, metodos con receptor por puntero que modifican el saldo, validaciones, y el metodo `String()` que implementa la interfaz `fmt.Stringer`.

```

package main

```

```

import "fmt"

// CuentaBancaria modela una cuenta simple.
// Los campos son privados: solo se accede a ellos mediante metodos.
type CuentaBancaria struct {
    titular string
    saldo   float64
}

// NuevaCuenta es el constructor por convencion.
// Devuelve un puntero porque los metodos usan receptor por puntero.
func NuevaCuenta(titular string, depositoInicial float64) *CuentaBancaria {
    if depositoInicial < 0 {
        depositoInicial = 0 // invariante: el saldo nunca empieza negativo
    }
    return &CuentaBancaria{titular: titular, saldo: depositoInicial}
}

// Depositar agrega fondos. Rechaza cantidades no positivas.
func (c *CuentaBancaria) Depositar(cantidad float64) error {
    if cantidad <= 0 {
        return fmt.Errorf("cantidad a depositar debe ser positiva: %.2f", cantidad)
    }
    c.saldo += cantidad
    return nil
}

// Retirar quita fondos si hay saldo suficiente.
func (c *CuentaBancaria) Retirar(cantidad float64) error {
    if cantidad <= 0 {
        return fmt.Errorf("cantidad a retirar debe ser positiva: %.2f", cantidad)
    }
    if cantidad > c.saldo {
        return fmt.Errorf("fondos insuficientes: saldo %.2f, retiro %.2f", c.saldo,
            cantidad)
    }
    c.saldo -= cantidad
    return nil
}

// Saldo devuelve el saldo actual (metodo de lectura).
func (c *CuentaBancaria) Saldo() float64 {
    return c.saldo
}

```

```

// Titular devuelve el nombre del titular.
func (c *CuentaBancaria) Titular() string {
    return c.titular
}

// String implementa fmt.Stringer: controla como se imprime la cuenta.
func (c *CuentaBancaria) String() string {
    return fmt.Sprintf("Cuenta de %s | Saldo: $%.2f", c.titular, c.saldo)
}

func main() {
    cuenta := NuevaCuenta("Ana", 1000)

    // Gracias a String(), Println muestra algo legible en lugar de &{Ana 1000}
    fmt.Println(cuenta)

    if err := cuenta.Depositar(500); err != nil {
        fmt.Println("Error:", err)
    }
    if err := cuenta.Retirar(200); err != nil {
        fmt.Println("Error:", err)
    }
    fmt.Println(cuenta) // Cuenta de Ana | Saldo: $1300.00

    // Intentos invalidos: el struct se protege solo
    if err := cuenta.Retirar(5000); err != nil {
        fmt.Println("Error:", err) // fondos insuficientes
    }
    if err := cuenta.Depositar(-100); err != nil {
        fmt.Println("Error:", err) // cantidad debe ser positiva
    }
    fmt.Println("Saldo final:", cuenta.Saldo())
}

```

Hay mucho que estudiar aquí. Primero, el encapsulamiento: los campos son privados, así que nadie puede hacer `cuenta.saldo = -9999` desde otro paquete; toda modificación pasa por `Depositar` y `Retirar`, que validan. Segundo, los métodos devuelven `error` en lugar de detonar el programa: en Go los errores son valores que se revisan, no excepciones. Tercero, el método `String()`: cuando un tipo tiene un método `String() string`, el paquete `fmt` lo usa automáticamente al imprimir. Sin el, `fmt.Println(cuenta)` mostraría `&{Ana 1300}`; con el, muestra algo diseñado para humanos. Implementar `fmt.Stringer` es una de las primeras interfaces que todo programador Go implementa, casi sin darse cuenta.

Fijate también en que `NuevaCuenta` devuelve `\*CuentaBancaria` (puntero): como los metodos modifican el saldo, trabajar con punteros evita copias y garantiza que todos los metodos operen sobre la misma cuenta. Y la validacion en el constructor establece la invariante desde el nacimiento: ninguna cuenta nace con saldo negativo.

Ejemplo 4: Jugador con inventario y metodos que modifican estado

Un ejemplo mas ludico que muestra un struct con un slice adentro, metodos por puntero que cambian el estado, y el cuidado con las copias.

```
package main

import "fmt"

type Jugador struct {
    Nombre      string
    Nivel       int
    Vida        int
    VidaMaxima  int
    Inventario  []string
}

func NuevoJugador(nombre string) *Jugador {
    return &Jugador{
        Nombre:      nombre,
        Nivel:       1,
        Vida:        100,
        VidaMaxima:  100,
        Inventario:  []string{},
    }
}

// SubirNivel aumenta el nivel y restaura la vida: modifica el receptor.
func (j *Jugador) SubirNivel() {
    j.Nivel++
    j.VidaMaxima += 20
    j.Vida = j.VidaMaxima
    fmt.Printf("%s subio al nivel %d\n", j.Nombre, j.Nivel)
}

// RecibirDanio resta vida sin bajar de cero.
func (j *Jugador) RecibirDanio(danio int) {
    j.Vida -= danio
    if j.Vida < 0 {
```

```

    j.Vida = 0
}
}

// EstaVivo reporta si al jugador le queda vida.
func (j *Jugador) EstaVivo() bool {
    return j.Vida > 0
}

// Recoger agrega un objeto al inventario.
func (j *Jugador) Recoger(objeto string) {
    j.Inventario = append(j.Inventario, objeto)
}

func main() {
    j := NuevoJugador("Aria")
    j.Recoger("espada")
    j.Recoger("pocion")
    j.RecibirDanio(30)
    fmt.Printf("%s: vida %d/%d, inventario %v\n",
        j.Nombre, j.Vida, j.VidaMaxima, j.Inventario)

    j.SubirNivel()
    fmt.Printf("Tras subir: vida %d/%d\n", j.Vida, j.VidaMaxima)

    // Cuidado: copiar el struct comparte el inventario
    clon := *j
    clon.Recoger("escudo")
    fmt.Println("Original:", j.Inventario) // incluye "escudo": slice compartido
}

```

La ultima parte es la leccion: `clon := \*j` copia el struct, pero el campo `Inventario` es un slice, y los slices copiados comparten el arreglo subyacente. Al agregar "escudo" al clon con `append`, como habia capacidad... bueno, en este caso el append puede o no reubicar, pero el punto conceptual es valido: copiar un struct no hace copia profunda de sus slices y mapas. Si necesitas un clon independiente, debes clonar también los campos referencia a mano. Este es el tipo de detalle que muerde en programas grandes, así que mejor aprenderlo aqui con un ejemplo pequeno.

Ejemplo 5: Embedding en accion (Empleado y Persona)

Veamos como el embedding promociona campos y metodos, y en que se diferencia de un struct anidado con nombre.

```

package main

import "fmt"

type Persona struct {
    Nombre string
    Edad   int
}

func (p Persona) Presentarse() string {
    return fmt.Sprintf("Me llamo %s y tengo %d años", p.Nombre, p.Edad)
}

type Empleado struct {
    Persona          // embedding: campos y metodos promocionados
    Puesto   string
    Sueldo   float64
}

type Cliente struct {
    Persona          // embedding también
    Direccion Direccion // anidado con nombre: acceso explicito
}

type Direccion struct {
    Calle   string
    Ciudad string
}

func main() {
    e := Empleado{
        Persona: Persona{Nombre: "Luis", Edad: 30},
        Puesto:   "Desarrollador",
        Sueldo:   25000,
    }
    // Campos y metodos promocionados: se usan como si fueran de Empleado
    fmt.Println(e.Nombre)
    fmt.Println(e.Presentarse())

    c := Cliente{
        Persona:   Persona{Nombre: "Maria", Edad: 28},
        Direccion: Direccion{Calle: "Av. Juarez 123", Ciudad: "CDMX"},
    }
    fmt.Println(c.Presentarse())           // promocionado
    fmt.Println(c.Direccion.Ciudad)       // anidado: ruta explicita
}

```

```
}
```

`Empleado` y `Cliente` reutilizan `Persona` sin herencia: ambos "tienen" los campos y el metodo `Presentarse`, accesibles directamente. Pero `Direccion` esta anidada con nombre, asi que se accede con la ruta completa `c.Direccion.Ciudad`. La regla para elegir: embedding cuando quieres que el exterior se comporte como el interior (promocion transparente), campo con nombre cuando la relacion debe leerse explicita en el codigo.

Ejemplo 6: Tags JSON en una API simulada

Los tags brillan al trabajar con JSON. Este ejemplo simula recibir el JSON de una API, convertirlo a struct, modificarlo y devolverlo, con nombres de campo en minusculas y un campo sensible excluido.

```

package main

import (
    "encoding/json"
    "fmt"
)

type Usuario struct {
    ID      int    `json:"id"`
    Nombre  string `json:"nombre"`
    Email   string `json:"email"`
    Password string `json:"- "` // nunca sale en el JSON
    Admin   bool   `json:"es_admin"`
}

func main() {
    // Simulamos el JSON que llega de una API externa
    entrada := `{"id": 7, "nombre": "Ana", "email": "ana@example.com", "es_admin": true}`

    var u Usuario
    if err := json.Unmarshal([]byte(entrada), &u); err != nil {
        fmt.Println("Error:", err)
        return
    }
    fmt.Printf("Leido: %+v\n", u) // %+v muestra los nombres de los campos

    u.Password = "super-secreto" // lo guardamos internamente...

    salida, err := json.Marshal(u)
    if err != nil {
        fmt.Println("Error:", err)
        return
    }
    fmt.Println("Enviado:", string(salida))
    // {"id":7,"nombre":"Ana","email":"ana@example.com","es_admin":true}
    // Password no aparece gracias al tag "-"
}

```

Tres detalles a notar. Primero, `json.Unmarshal`` necesita un puntero (`&u``) porque debe modificar el struct. Segundo, `%+v`` en `Printf`` imprime el struct con los nombres de los campos, útil para depurar. Tercero, el tag `json:"- "`` excluye el password de la salida: un error común de seguridad es olvidar ese tag y filtrar contraseñas en las respuestas de una API. Los tags son metadatos, pero tienen consecuencias reales.

Ejemplo 7: Structs anidados para un pedido

Un pedido de tienda en línea modelado con structs anidados: el pedido tiene un cliente, una lista de artículos (slice de structs), y métodos que calculan el total.

```
package main

import "fmt"

type Cliente struct {
    Nombre string
    Email  string
}

type Artículo struct {
    Producto string
    Precio   float64
    Cantidad int
}

type Pedido struct {
    Folio    int
    Cliente  Cliente
    Articulos []Artículo
}

// Total suma precio*cantidad de cada artículo.
func (p Pedido) Total() float64 {
    total := 0.0
    for _, a := range p.Articulos {
        total += a.Precio * float64(a.Cantidad)
    }
    return total
}

// AgregarArtículo anade un artículo al pedido.
func (p *Pedido) AgregarArtículo(a Artículo) {
    p.Articulos = append(p.Articulos, a)
}

func main() {
    pedido := Pedido{
        Folio:    1001,
        Cliente:  Cliente{Nombre: "Pedro", Email: "pedro@example.com"},
    }
    pedido.AgregarArtículo(Artículo{Producto: "Teclado", Precio: 899.50, Cantidad: 1})
}
```

```
pedido.AgregarArticulo(Articulo{Producto: "Mouse", Precio: 349.00, Cantidad: 2})

fmt.Printf("Pedido %d para %s: total $%.2f\n",
    pedido.Folio, pedido.Cliente.Nombre, pedido.Total())
}
```

Este ejemplo muestra structs en tres niveles: `Pedido` contiene un `Cliente` y un slice de `Articulo`. El metodo `Total` usa receptor por valor (solo lee), mientras que `AgregarArticulo` usa puntero (modifica el slice). Es el patron mixto correcto: cada metodo usa el receptor que su comportamiento requiere, con la salvedad de la regla de consistencia. Aqui hay una tension real: `Total` por valor y `AgregarArticulo` por puntero mezclan receptores en el mismo tipo. En codigo profesional, para evitar confusiones, se suele poner todo por puntero cuando algun metodo lo necesita. Lo mostramos mixto aqui con proposito didactico, pero en tu codigo sigue la regla de consistencia: si uno es puntero, todos puntero.

Ejemplo 8: Struct comparable como clave de mapa

Para cerrar, un caso practico de structs comparables: usar un struct como clave de un mapa para contar combinaciones. Contemos cuantas veces aparece cada par (producto, sucursal) en una lista de ventas.

```

package main

import "fmt"

// Venta es comparable: todos sus campos son strings.
// Por eso puede ser clave de mapa.
type Venta struct {
    Producto string
    Sucursal string
}

func main() {
    ventas := []Venta{
        {"tortillas", "norte"},
        {"huevos", "sur"},
        {"tortillas", "norte"},
        {"tortillas", "sur"},
        {"huevos", "sur"},
    }

    conteo := make(map[Venta]int)
    for _, v := range ventas {
        conteo[v]++
    }

    for venta, veces := range conteo {
        fmt.Printf("%s en %s: %d ventas\n", venta.Producto, venta.Sucursal, veces)
    }
}

```

Un struct con puros campos comparables funciona como clave de mapa sin ningun esfuerzo extra: Go compara campo por campo. Esto es mas limpio que concatenar strings como `"tortillas|norte"` para fabricar una clave, porque mantiene los datos estructurados y evita colisiones del separador. Si el struct tuviera un slice, no compilaria como clave; en ese caso, la alternativa es mantener los campos comparables o usar otro diseno. Diseñar tus tipos pensando en como se comparan y se usan como claves es parte de pensar en Go.

Ejemplo 9: Inventario de tienda (structs + mapas juntos)

Las unidades 5 y 6 se encuentran aqui: un mapa cuya clave es el codigo del producto y cuyo valor es un struct con los datos. Es el diseno tipico de un catalogo, un registro de usuarios o cualquier indice por clave.


```

package main

import "fmt"

type Producto struct {
    Codigo string
    Nombre string
    Precio float64
    Stock int
}

type Tienda struct {
    nombre string
    productos map[string]Producto
}

func NuevaTienda(nombre string) *Tienda {
    return &Tienda{
        nombre: nombre,
        productos: make(map[string]Producto),
    }
}

// Registrar agrega un producto al catalogo.
func (t *Tienda) Registrar(p Producto) {
    t.productos[p.Codigo] = p
}

// Vender descuenta stock si hay suficiente. Devuelve error si no.
func (t *Tienda) Vender(codigo string, cantidad int) error {
    p, ok := t.productos[codigo]
    if !ok {
        return fmt.Errorf("producto %q no existe", codigo)
    }
    if p.Stock < cantidad {
        return fmt.Errorf("stock insuficiente de %q: hay %d", p.Nombre, p.Stock)
    }
    p.Stock -= cantidad
    t.productos[codigo] = p // hay que guardar el struct modificado
    return nil
}

// ValorInventario suma precio*stock de todo el catalogo.
func (t *Tienda) ValorInventario() float64 {
    total := 0.0

```

```

for _, p := range t.productos {
    total += p.Precio * float64(p.Stock)
}
return total
}

func main() {
    tienda := NuevaTienda("La Esquina")
    tienda.Registrar(Producto{Codigo: "A1", Nombre: "Tortillas", Precio: 18.50, Stock: 100})
    tienda.Registrar(Producto{Codigo: "B2", Nombre: "Huevos", Precio: 45.00, Stock: 60})

    if err := tienda.Vender("A1", 10); err != nil {
        fmt.Println("Error:", err)
    }
    if err := tienda.Vender("B2", 999); err != nil {
        fmt.Println("Error:", err) // stock insuficiente
    }
    fmt.Printf("Valor del inventario: %.2f\n", tienda.ValorInventario())
}

```

La línea más instructiva es `t.productos[codigo] = p`` dentro de `Vender``. Cuando lees `p``, `ok := t.productos[codigo]`, obtienes una copia del struct; modificar `p.Stock`` no toca el mapa. Hay que guardar la copia modificada de vuelta. Este es el error más común al combinar mapas con structs, y la regla para recordarlo es: "los valores de un mapa no son direccionables, así que se leen por copia y se escriben completos". La alternativa es guardar punteros en el mapa (`map[string]*Producto``), donde la copia es del puntero y las modificaciones sí se reflejan; cada diseño tiene sus implicaciones y vale la pena conocer ambos.

Ejemplo 10: Constructor con validación y valores por defecto

Un constructor bien diseñado hace imposible crear un valor inválido. Este ejemplo modela la configuración de un servidor: si el usuario no especifica puerto, se usa el 8080; si el número de conexiones es absurdo, se corrige; y todo queda validado en un solo lugar.

```

package main

import "fmt"

type Servidor struct {
    Host          string
    Puerto        int
    MaxConexiones int
}

```

```

func NuevoServidor(host string, puerto, maxConexiones int) *Servidor {
    s := &Servidor{Host: host}

    // Valores por defecto
    if puerto == 0 {
        puerto = 8080
    }
    s.Puerto = puerto

    // Validacion con limites sensatos
    if maxConexiones <= 0 {
        maxConexiones = 100
    }
    if maxConexiones > 10000 {
        maxConexiones = 10000
    }
    s.MaxConexiones = maxConexiones

    return s
}

func (s *Servidor) Direccion() string {
    return fmt.Sprintf("%s:%d", s.Host, s.Puerto)
}

func (s *Servidor) String() string {
    return fmt.Sprintf("Servidor en %s (max %d conexiones)",
        s.Direccion(), s.MaxConexiones)
}

func main() {
    s1 := NuevoServidor("localhost", 0, 0) // todo por defecto
    s2 := NuevoServidor("api.midominio.com", 9090, 50000) // conexiones recortadas
    s3 := NuevoServidor("127.0.0.1", 3000, 200)

    fmt.Println(s1) // Servidor en localhost:8080 (max 100 conexiones)
    fmt.Println(s2) // Servidor en api.midominio.com:9090 (max 10000 conexiones)
    fmt.Println(s3)
}

```

Observa el patron: el constructor recibe lo que el usuario quiere configurar y se encarga del resto. Quien usa `NuevoServidor` no necesita saber cuales son los valores por defecto ni los limites; el tipo se protege solo. Este es el mismo principio que vimos en `NuevaCuenta` (saldo nunca negativo) aplicado a otro dominio. Un buen constructor es documentacion ejecutable: leyendo sus validaciones entiendes las reglas del tipo sin buscar en otro lado.

Practica

Los structs se aprenden disenandolos. Estos ejercicios te piden modelar entidades reales y darles comportamiento con metodos. Como siempre: intenta primero, atora te un rato, y solo después mira la solucion. Y cuando la mires, reescribela desde cero.

Ejercicio 1: Libro de biblioteca

Disena un struct `Libro` con titulo, autor y un campo que indique si esta prestado. Agrega un constructor `NuevoLibro`, y metodos `Prestar()` y `Devolver()` que cambien el estado y devuelvan un error si la operacion no tiene sentido (prestar un libro ya prestado, devolver uno que no lo esta). También implementa `String()` para mostrar la ficha.

Solucion:

```
package main

import "fmt"

type Libro struct {
    Titulo    string
    Autor     string
    Prestado  bool
}

func NuevoLibro(titulo, autor string) *Libro {
    return &Libro{Titulo: titulo, Autor: autor}
}

func (l *Libro) Prestar() error {
    if l.Prestado {
        return fmt.Errorf("el libro %q ya esta prestado", l.Titulo)
    }
    l.Prestado = true
    return nil
}
```

```

func (l *Libro) Devolver() error {
    if !l.Prestado {
        return fmt.Errorf("el libro %q no estaba prestado", l.Titulo)
    }
    l.Prestado = false
    return nil
}

func (l *Libro) String() string {
    estado := "disponible"
    if l.Prestado {
        estado = "prestado"
    }
    return fmt.Sprintf("%q de %s [%s]", l.Titulo, l.Autor, estado)
}

func main() {
    libro := NuevoLibro("Cien años de soledad", "Gabriel Garcia Marquez")
    fmt.Println(libro)

    if err := libro.Prestar(); err != nil {
        fmt.Println("Error:", err)
    }
    fmt.Println(libro)
    if err := libro.Prestar(); err != nil {
        fmt.Println("Error:", err) // ya esta prestado
    }
    libro.Devolver()
    fmt.Println(libro)
}

```

Explicacion: el campo `Prestado` es el estado que los metodos protegen. Cada metodo valida antes de cambiar: prestar dos veces es un error logico, y devolver sin prestar también. Devolver `error` en lugar de un booleano permite mensajes descriptivos. Los receptores son por puntero porque modifican el estado, y `String()` implementa `fmt.Stringer` para que `Println` muestre la ficha legible. Nota el uso de `%q` para entrecomillar el titulo automaticamente. Este ejercicio es un mini-modelo de como se disenian tipos con invariantes en Go: el estado solo cambia por caminos validados.

Ejercicio 2: Temperatura con conversiones

Crea un tipo `Celsius` (basado en `float64`) con metodos `AFahrenheit()` y `AKelvin()` que devuelvan la conversion. Los metodos pueden definirse sobre tipos propios aunque no sean structs: cualquier tipo declarado con `type` puede tener metodos.

```
package main

import "fmt"

type Celsius float64

func (c Celsius) AFahrenheit() float64 {
    return float64(c)*9.0/5.0 + 32.0
}

func (c Celsius) AKelvin() float64 {
    return float64(c) + 273.15
}

func (c Celsius) String() string {
    return fmt.Sprintf("%.1f C", float64(c))
}

func main() {
    var t Celsius = 25
    fmt.Println(t) // 25.0 C
    fmt.Printf("%.1f F\n", t.AFahrenheit()) // 77.0 F
    fmt.Printf("%.2f K\n", t.AKelvin()) // 298.15 K
}
```

Explicacion: la leccion clave es que los metodos no son exclusivos de los structs: cualquier tipo nombrado puede tenerlos, incluyendo tipos basados en primitivos. Esto permite crear "tipos con significado": un `Celsius` no es un `float64` generico, es una temperatura, y el compilador te impide mezclarla accidentalmente con otras cantidades. Los receptores son por valor porque el tipo es pequeno y los metodos no modifican nada. Este patron (tipos pequenos con metodos de conversion o validacion) aparece mucho en codigo Go real: duraciones, dinero, identificadores.

Ejercicio 3: Carrito de compras

Disena un `Carrito` que guarde articulos (nombre y precio) en un slice, con metodos `Agregar`, `Total`, `Vaciar` y `Cantidad`. El metodo `Total` debe usar receptor por valor o por puntero según corresponda; justifica tu eleccion en un comentario.

Solucion:

```
package main

import "fmt"

type ArticuloCarrito struct {
    Nombre string
    Precio float64
}

type Carrito struct {
    articulos []ArticuloCarrito
}

func NuevoCarrito() *Carrito {
    return &Carrito{articulos: []ArticuloCarrito{}}
}

func (c *Carrito) Agregar(nombre string, precio float64) {
    c.articulos = append(c.articulos, ArticuloCarrito{nombre, precio})
}

// Total usa receptor por valor: solo lee, no modifica.
func (c Carrito) Total() float64 {
    total := 0.0
    for _, a := range c.articulos {
        total += a.Precio
    }
    return total
}

func (c Carrito) Cantidad() int {
    return len(c.articulos)
}

func (c *Carrito) Vaciar() {
    c.articulos = []ArticuloCarrito{}
}
```

```
func main() {
    carrito := NuevoCarrito()
    carrito.Agregar("Teclado", 899.50)
    carrito.Agregar("Mouse", 349.00)
    fmt.Printf("Articulos: %d, Total: $%.2f\n", carrito.Cantidad(), carrito.Total())
    carrito.Vaciar()
    fmt.Printf("Tras vaciar: %d articulos\n", carrito.Cantidad())
}
```

Explicacion: el slice de articulos es privado, así que el carrito controla como se agregan elementos (podríamos validar precios negativos en `Agregar`, por ejemplo). `Agregar` y `Vaciar` usan puntero porque modifican; `Total` y `Cantidad` usan valor porque solo leen. Aquí aplicamos a propósito la mezcla didáctica para practicar el criterio, pero recuerda la regla de consistencia para código profesional: si un método necesita puntero, considera ponerlos todos por puntero. `Vaciar` reasigna un slice nuevo en lugar de hacer `c.articulos = c.articulos[:0]`: la primera forma libera la memoria del contenido anterior (el recolector puede reclamarla), la segunda la conserva para reutilizar. Para un carrito que se vacía y se vuelve a llenar muchas veces, la segunda sería más eficiente; para el caso simple, la primera es más clara.

Ejercicio 4: Empleado con embedding y bono

Usando embedding, crea un struct `Empleado` que incluya `Persona` (nombre, edad) y agregue puesto y sueldo mensual. Implementa un método `SalarioAnual()` que devuelva el sueldo por 12 más un bono del 10% si la edad es mayor a 40. También implementa `Presentarse()` en `Persona` y usalo desde un `Empleado`.

Solucion:


```

package main

import "fmt"

type Persona struct {
    Nombre string
    Edad    int
}

func (p Persona) Presentarse() string {
    return fmt.Sprintf("Me llamo %s y tengo %d años", p.Nombre, p.Edad)
}

type Empleado struct {
    Persona
    Puesto string
    Sueldo float64 // mensual
}

func (e Empleado) SalarioAnual() float64 {
    anual := e.Sueldo * 12
    if e.Edad > 40 {
        anual *= 1.10 // bono del 10%
    }
    return anual
}

func main() {
    e1 := Empleado{
        Persona: Persona{Nombre: "Rosa", Edad: 45},
        Puesto:   "Gerente",
        Sueldo:   30000,
    }
    e2 := Empleado{
        Persona: Persona{Nombre: "Juan", Edad: 28},
        Puesto:   "Analista",
        Sueldo:   18000,
    }
    fmt.Println(e1.Presentarse()) // metodo promocionado de Persona
    fmt.Printf("Salario anual de %s: $%.2f\n", e1.Nombre, e1.SalarioAnual())
    fmt.Printf("Salario anual de %s: $%.2f\n", e2.Nombre, e2.SalarioAnual())
}

```

Explicacion: el embedding promociona `Nombre`, `Edad` y el metodo `Presentarse` al `Empleado`, que los usa como propios. `SalarioAnual` accede a `e.Edad` directamente gracias a la promocion. El bono por edad es logica de negocio encapsulada en el metodo: quien usa el `Empleado` no necesita saber la regla, solo llamar al metodo. Si la regla cambiara (bono desde los 50, o del 15%), se cambia en un solo lugar. Esa es la ganancia de poner comportamiento en metodos en lugar de esparcirlo en funciones sueltas por todo el programa.

Ejercicio 5: Producto con JSON de ida y vuelta

Define un struct `Producto` con ID, nombre, precio y stock, con tags JSON en minusculas y guion bajo (`en\_stock`). Escribe un programa que cree un producto, lo convierta a JSON, imprima el JSON, luego lo convierta de vuelta a struct y verifique que los datos sobrevivieron el viaje.

Solucion:

```

package main

import (
    "encoding/json"
    "fmt"
    "reflect"
)

type Producto struct {
    ID      int      `json:"id"`
    Nombre  string   `json:"nombre"`
    Precio  float64  `json:"precio"`
    EnStock bool     `json:"en_stock"`
}

func main() {
    original := Producto{ID: 101, Nombre: "Teclado mecanico", Precio: 899.50, EnStock: true}

    datos, err := json.Marshal(original)
    if err != nil {
        fmt.Println("Error al convertir:", err)
        return
    }
    fmt.Println("JSON:", string(datos))

    var recuperado Producto
    if err := json.Unmarshal(datos, &recuperado); err != nil {
        fmt.Println("Error al leer:", err)
        return
    }

    if reflect.DeepEqual(original, recuperado) {
        fmt.Println("Viaje redondo exitoso: los datos son identicos")
    } else {
        fmt.Println("Algo se perdio en el camino")
    }
}

```

Explicacion: el "viaje redondo" (marshal seguido de unmarshal) es la prueba basica de que tus tags estan bien: si el JSON usa los nombres que esperas y los datos regresan intactos, el struct esta listo para hablar con una API. `reflect.DeepEqual` compara structs a fondo, incluyendo todos los campos; es util en pruebas, aunque en codigo de produccion se prefiere comparar campo por campo para mensajes de error mas claros. Nota que `EnStock` se

convierte en `en\_stock` gracias al tag: sin el, el JSON diria `EnStock` y la API del otro lado (probablemente esperando snake_case) no lo reconoceria. Un detalle de una linea que decide si tu integracion funciona o no.

Ejercicio 6: Pila (stack) con metodos

Implementa una pila de enteros (estructura LIFO: ultimo en entrar, primero en salir) como un struct con un slice interno. Metodos: `Push`, `Pop` (devuelve el valor y un error si esta vacia), `Vacía` y `Tamano`. Usa receptor por puntero donde corresponda.

Solucion:

```
package main

import (
    "errors"
    "fmt"
)

type Pila struct {
    elementos []int
}

func NuevaPila() *Pila {
    return &Pila{elementos: []int{}}
}

func (p *Pila) Push(v int) {
    p.elementos = append(p.elementos, v)
}

func (p *Pila) Pop() (int, error) {
    if len(p.elementos) == 0 {
        return 0, errors.New("la pila esta vacia")
    }
    ultimo := len(p.elementos) - 1
    v := p.elementos[ultimo]
    p.elementos = p.elementos[:ultimo] // recorta sin borrar el dato
    return v, nil
}

func (p *Pila) Vacía() bool {
    return len(p.elementos) == 0
}
```

```

func (p *Pila) Tamano() int {
    return len(p.elementos)
}

func main() {
    pila := NuevaPila()
    pila.Push(10)
    pila.Push(20)
    pila.Push(30)

    for !pila.Vacia() {
        v, err := pila.Pop()
        if err != nil {
            fmt.Println("Error:", err)
            break
        }
        fmt.Println("Salio:", v) // 30, 20, 10: orden LIFO
    }

    if _, err := pila.Pop(); err != nil {
        fmt.Println("Error:", err) // la pila esta vacia
    }
}

```

Explicacion: la pila es el ejemplo clasico de tipo abstracto de datos, y en Go se implementa de forma natural con un struct que envuelve un slice. `Push` agrega al final con `append` (recuerda asignar el resultado al campo, no a una variable local). `Pop` toma el ultimo elemento y recorta el slice; el dato "borrado" sigue en el arreglo subyacente mas alla de la longitud, pero es inaccesible y se sobrescribira con el proximo `Push`, asi que no hay fuga. Devolver `(int, error)` sigue el idioma Go para operaciones que pueden fallar. `Vacia` y `Tamano` podrian usar receptor por valor (solo leen), pero por la regla de consistencia los dejamos por puntero junto con `Push` y `Pop`. Este ejercicio conecta las dos unidades: el struct le da forma a la pila, y el slice de la unidad 5 le da el almacenamiento.

Ejercicio 7: Agenda de contactos

Disena una `Agenda` que guarde contactos (nombre, telefono, email) en un mapa indexado por nombre. Implementa `Agregar`, `Buscar` (que devuelva el contacto y un booleano como la coma-ok), `Eliminar` y `Total`. El constructor debe inicializar el mapa para que nunca sea nil.

Solucion:

```

package main

```

```

import "fmt"

type Contacto struct {
    Nombre    string
    Telefono  string
    Email     string
}

type Agenda struct {
    contactos map[string]Contacto
}

func NuevaAgenda() *Agenda {
    return &Agenda{contactos: make(map[string]Contacto)}
}

func (a *Agenda) Agregar(c Contacto) {
    a.contactos[c.Nombre] = c
}

// Buscar devuelve el contacto y si existe: el patron coma-ok como metodo.
func (a *Agenda) Buscar(nombre string) (Contacto, bool) {
    c, ok := a.contactos[nombre]
    return c, ok
}

func (a *Agenda) Eliminar(nombre string) bool {
    if _, ok := a.contactos[nombre]; !ok {
        return false
    }
    delete(a.contactos, nombre)
    return true
}

func (a *Agenda) Total() int {
    return len(a.contactos)
}

func main() {
    agenda := NuevaAgenda()
    agenda.Agregar(Contacto{Nombre: "Ana", Telefono: "555-1234", Email:
"ana@example.com"})
    agenda.Agregar(Contacto{Nombre: "Luis", Telefono: "555-5678", Email:
"luis@example.com"})
}

```

```

if c, ok := agenda.Buscar("Ana"); ok {
    fmt.Printf("Encontrado: %s, tel %s\n", c.Nombre, c.Telefono)
}
if _, ok := agenda.Buscar("Pedro"); !ok {
    fmt.Println("Pedro no esta en la agenda")
}

fmt.Println("Eliminado Luis:", agenda.Eliminar("Luis"))
fmt.Println("Total:", agenda.Total())
}

```

Explicacion: este ejercicio es la coma-ok de la unidad 5 elevada a metodo: `Buscar` devuelve `(Contacto, bool)` igual que un acceso a mapa con dos variables. El llamador decide que hacer con cada caso, y el codigo se lee casi como espanol: "si buscar Ana, ok". El constructor inicializa el mapa porque un mapa nil acepta lecturas pero revienta al escribir; inicializar en `NuevaAgenda` elimina esa trampa de raiz. `Eliminar` devuelve bool para informar si habia algo que borrar, siguiendo el mismo espiritu de "valor + ok". Nota que `Agregar` sobrescribe si el nombre ya existe: es el comportamiento de los mapas, y si quisieras prohibir duplicados tendrias que validarlo explicitamente. Diseñar es decidir, y cada decision asi debe ser consciente.

Ejercicio 8: Figuras geometricas con embedding

Crea un struct `Figura` con campo `Color` y metodo `Describir()` que devuelva una frase con el color. Luego crea `Circulo` (con `Radio`) y `Rectangulo` (con `Base`, `Altura`) que embeban `Figura`. Cada uno debe tener su metodo `Area()`. Demuestra que `Describir` funciona en ambos gracias a la promocion.

Solucion:

```

package main

import (
    "fmt"
    "math"
)

type Figura struct {
    Color string
}

func (f Figura) Describir() string {
    return fmt.Sprintf("Figura de color %s", f.Color)
}

```

```

type Circulo struct {
    Figura // embedding
    Radio float64
}

func (c Circulo) Area() float64 {
    return math.Pi * c.Radio * c.Radio
}

type Rectangulo struct {
    Figura // embedding
    Base float64
    Altura float64
}

func (r Rectangulo) Area() float64 {
    return r.Base * r.Altura
}

func main() {
    c := Circulo{Figura: Figura{Color: "rojo"}, Radio: 5}
    r := Rectangulo{Figura: Figura{Color: "azul"}, Base: 4, Altura: 6}

    fmt.Println(c.Describir(), "| area:", c.Area())
    fmt.Println(r.Describir(), "| area:", r.Area())

    // El campo promocionado también se puede modificar directo
    c.Color = "verde"
    fmt.Println(c.Describir())
}

```

Explicacion: `Circulo` y `Rectangulo` reutilizan `Color` y `Describir` sin repetir codigo y sin herencia: cada uno "tiene una" figura base cuyos miembros se promocionan. Si mañana agregas un metodo `Mover()` a `Figura`, ambas figuras lo obtienen gratis. Pero nota los limites del embedding: `Area` esta definida por separado en cada tipo porque cada figura calcula distinto; el embedding reutiliza lo comun, no lo especifico. Y aunque `c.Describir()` funciona, `c` no "es" una `Figura` para efectos de asignacion directa de tipos: la promocion es de miembros, no de identidad. Este es el equilibrio de Go: reutilizacion sin jerarquias, cada tipo sigue siendo el mismo de siempre.

Resumen de la unidad

Todo lo esencial de structs y metodos, condensado para repaso.

Structs. Tipos compuestos que agrupan campos con nombre: ``type Jugador struct { Nombre string; Nivel int }`. Los campos con mayuscula son exportados (publicos), con minuscula son privados del paquete. Se instancian con literales nombrados (``Jugador{Nombre: "Aria", Nivel: 5}`), la forma recomendada), por asignacion campo por campo, o con ``new(T)` / `&T{} para obtener un puntero. Copiar un struct copia sus campos, pero los slices y mapas internos se siguen compartiendo.`

Embedding. Incluir un tipo sin nombre de campo promociona sus campos y metodos al exterior. Es composicion con azucar sintactica, no herencia: reutilizas implementacion sin crear jerarquias de tipos.

Metodos. Funciones con receptor: ``func (j Jugador) Saludar() string``. Agrupan el comportamiento con los datos. También existen sobre tipos no-struct (como ``type Celsius float64``).

Receptor por valor vs por puntero. Por valor: recibe una copia, no puede modificar el original; ideal para tipos pequenos inmutables. Por puntero: modifica el original y evita copias costosas; obligatorio cuando el metodo cambia el estado. Regla de consistencia: si un metodo usa puntero, todos los del tipo usan puntero. Si un metodo "no modifica nada" y deberia, revisa el receptor primero.

Constructores. No hay constructores magicos; la convencion es ``func NuevaCuenta(...)*CuentaBancaria`` que valida, pone valores por defecto y devuelve el valor listo para usar. Es donde se establecen las invariantes del tipo.

Comparabilidad. Un struct es comparable con ``==`` si todos sus campos lo son. Si tiene slices o mapas, escribe tu propio metodo ``Igual``. Los structs comparables sirven como claves de mapa.

Tags. Metadatos entre backticks: ``` `json:"nombre" ``` controla la serializacion, ``` `json:"- " ``` excluye el campo. Van con acentos graves, no con comillas dobles. Se usan para JSON, bases de datos y validacion.

Structs anidados. Campos que son otros structs, con ruta de acceso explicita (``p.Direccion.Ciudad``). Ideales para modelar relaciones y respuestas de APIs con objetos anidados.

Composicion vs herencia. Go no tiene herencia: se reutiliza componiendo (embedding para implementacion, structs anidados para relaciones explicitas). La filosofia se resume en "acepta interfaces, devuelve structs": parametros flexibles por comportamiento, resultados concretos y predecibles.

fmt.Stringer. Si tu tipo tiene `String() string`, `fmt` lo usa al imprimir. Implementalo siempre que el tipo aparezca en logs o salida para humanos.

Mapa de decisiones: que elegir y cuando

La teoria esta clara, pero en la practica te enfrentaras a decisiones de diseno una y otra vez. Aqui tienes las preguntas que debes hacerte, con respuestas directas.

Valor o puntero como receptor? Preguntate: el metodo modifica el struct? Si si, puntero, sin discusion. Si no, preguntate: el struct es grande (mas de unos cuantos campos, o contiene arreglos grandes)? Si si, puntero por eficiencia. Si no, valor. Y después aplica la regla de consistencia: mira los demas metodos del tipo; si alguno usa puntero, usa puntero en todos. En caso de duda entre valor y puntero para un tipo nuevo, elige puntero: es la eleccion mas comun en codigo Go real y te evita tener que cambiar todo cuando mas adelante agregues un metodo que si modifica.

Struct anidado con nombre o embedding? Preguntate: quiero que quien lea el codigo vea la relacion explicita? Un `Pedido` con campo `Cliente` se lee mejor como `pedido.Cliente.Nombre`: la relacion es parte del dominio. En cambio, si el interior es un detalle de implementacion que solo existe para reutilizar codigo (una `Persona` dentro de `Empleado`, un `mutex` dentro de un contador), el embedding lo esconde elegantemente y promociona lo util. Otra prueba: si el tipo interior tiene metodos que quieres heredar tal cual, embedding; si solo quieres sus datos agrupados, campo con nombre.

Metodo o funcion suelta? Preguntate: la operacion necesita campos privados? Si si, metodo. Es algo que haras a diario con ese tipo (`Depositar`, `Agregar`, `Total`)? Metodo. Es un algoritmo que combina varios tipos o que se usa una vez al ano (`generarReporteAnual`, `migrarBaseDeDatos`)? Funcion suelta. Y recuerda la restriccion: no puedes definir metodos sobre tipos de otros paquetes; ahi la funcion suelta es tu unica opcion (o defines tu propio tipo basado en el ajeno).

Campos exportados o privados? Preguntate: quiero que otros paquetes lean o modifiquen este campo directamente? Solo exporta lo que forme parte del contrato publico del tipo. Todo lo demas, privado, con metodos que controlen el acceso. Un struct con todos los campos exportados es una estructura de datos pasiva (como `Punto` o `Producto`); uno con campos privados y metodos es un objeto con comportamiento protegido (como `CuentaBancaria`).

Ambas son validas; la primera para datos simples, la segunda cuando hay invariantes que proteger. El error comun es exportar todo "por si acaso": cada campo exportado es una promesa de compatibilidad que tendras que mantener.

Constructor ``New`` o literal directo? Si el tipo tiene invariantes (saldo no negativo, mapa inicializado, valores por defecto), escribe un constructor y documenta que es la forma oficial de crear el tipo. Si el tipo es una simple bolsa de datos sin reglas (``Punto``, ``Articulo``, ``Usuario`` de ejemplo), el literal directo es perfecto y un constructor seria burocracia. Senal de alerta: si te encuentras repitiendo la misma inicializacion con validaciones en tres lugares distintos, es hora de un constructor.

``String()`` o no? Implementa ``fmt.Stringer`` en todo tipo que aparezca en logs, mensajes de error o salida para el usuario. El costo es un metodo de tres lineas; el beneficio es no ver ``&{Ana 1300}`` en tus logs a las 3 de la manana cuando depuras un problema en produccion. Para tipos puramente internos que nadie imprime, puedes omitirlo.

Estas seis preguntas cubren el 90% de las decisiones de diseno con structs. No hay respuestas magicas, pero hay respuestas consistentes: y la consistencia, en un equipo o en un proyecto que crece, vale mas que la brillantez ocasional.

Hoja de referencia rapida

```
// --- Definir e instanciar ---
type Jugador struct {
    Nombre string
    Nivel  int
}
j := Jugador{Nombre: "Aria", Nivel: 5} // literal nombrado (recomendado)
p := new(Jugador)                      // *Jugador en ceros
p = &Jugador{}                        // equivalente a new

// --- Metodos ---
func (j Jugador) Saludar() string { return "Hola " + j.Nombre } // por valor
func (j *Jugador) Subir()      { j.Nivel++ }                  // por puntero

// --- Constructor por convencion ---
func NuevoJugador(nombre string) *Jugador {
    return &Jugador{Nombre: nombre, Nivel: 1}
}

// --- Embedding ---
type Empleado struct {
    Persona // promociona campos y metodos
    Sueldo  float64
}

// --- Tags JSON ---
type Usuario struct {
    Nombre  string `json:"nombre"`
    Password string `json:"- "`
}

// --- Stringer ---
func (j Jugador) String() string {
    return fmt.Sprintf("%s (nivel %d)", j.Nombre, j.Nivel)
}
```

Con esta unidad cierras el modelado de datos en Go: slices y mapas para colecciones (unidad 5), structs y metodos para entidades con comportamiento (unidad 6). Lo que sigue en tu camino es darles a esos tipos la capacidad de colaborar mediante interfaces, y luego ponerlos a trabajar en paralelo con goroutines. Pero esa es otra historia, de otras unidades. Por ahora, practica: toma cualquier cosa de tu vida (tu lista de peliculas, tus gastos del mes, tus contactos) y modelala con structs, metodos y un constructor. Si puedes representar tu mundo en tipos de Go y operar sobre ellos sin que el compilador se queje, ya piensas como programador Go.

Glosario de la unidad

Struct. Tipo compuesto que agrupa campos con nombre, cada uno con su tipo. La unidad básica para modelar entidades del mundo real en Go.

Campo. Cada dato con nombre dentro de un struct. Con mayúscula inicial es exportado (visible desde otros paquetes); con minúscula es privado.

Literal de struct. Forma de crear un valor nombrando los campos: `Jugador{Nombre: "Aria", Nivel: 5}`. La forma recomendada por legibilidad y resistencia a cambios.

Receptor. El parámetro especial entre `func`` y el nombre del método, que ata el método a un tipo: `func (j Jugador) Saludar()`.

Receptor por valor. El método recibe una copia del struct; no puede modificar el original. Para tipos pequeños que no cambian.

Receptor por puntero. El método recibe la dirección del struct; puede modificarlo y evita copiar. Obligatorio cuando el método cambia el estado.

Método. Función atada a un tipo mediante un receptor. Agrupa el comportamiento con los datos.

Constructor (por convención). Función `NewTipo`` que crea un valor válido del struct, con validaciones y valores por defecto. Go no tiene constructores mágicos; esta es la convención.

Embedding. Incluir un tipo dentro de un struct sin nombre de campo. Sus campos y métodos se promocionan al exterior. Composición, no herencia.

Campo promocionado. Campo o método del tipo embebido que se puede usar directamente desde el exterior, como si fuera propio.

Struct anidado. Campo cuyo tipo es otro struct, con nombre de campo explícito. Acceso por ruta completa: `p.Direccion.Ciudad``.

Tag. Metadato entre backticks pegado a un campo: ``json:"nombre"```. Lo leen otros paquetes por reflexión; no afecta al código directamente.

fmt.Stringer. Interfaz con un solo método, `String() string``. Si tu tipo la implementa, `fmt`` la usa automáticamente al imprimir.

Invariante. Regla que un tipo siempre cumple (el saldo nunca es negativo, el mapa nunca es nil). Se establece en el constructor y se protege con métodos.

Composicion. Reutilizar comportamiento combinando tipos (embedding, campos anidados) en lugar de heredarlos. La filosofía de Go frente a la herencia de clases.

"Acepta interfaces, devuelve structs". Principio de diseño Go: funciones flexibles en lo que reciben (comportamiento) y concretas en lo que entregan (tipos específicos).

Guarda este glosario junto a la hoja de referencia. Cuando en la siguiente unidad hablemos de interfaces, volverás a él y notarás algo satisfactorio: ya conoces la mitad del vocabulario necesario, porque las interfaces en Go existen para darle superpoderes a los structs y métodos que acabas de dominar.

UNIDAD 7: Interfaces y composición

Imagina que acabas de comprar una lámpara nueva. La sacas de la caja, la llevas a tu cuarto, buscas el contacto de la pared y la conectas. Funciona a la primera. No tuviste que llamar al electricista, no tuviste que cambiar el contacto de tu casa, no tuviste que preguntarle a la lámpara de qué marca es el cableado de tu pared. Eso pasa porque existe un acuerdo silencioso entre todos los fabricantes de aparatos eléctricos y todas las constructoras: el enchufe tiene una forma estándar. A la pared no le importa si conectas una lámpara, un cargador de celular, una televisión o una licuadora. Lo único que le importa es que el enchufe cumpla con la forma acordada. Si cumple, pasa corriente. Si no cumple, no entra. Así de simple, así de poderoso.

Las interfaces en Go son exactamente eso: el enchufe estándar de tu programa. Son un acuerdo sobre la forma que debe tener un tipo para poder "conectarse" con cierta parte de tu código. A una función no le tiene que importar si le pasas un Perro, un Gato, un Robot o un Extraterrestre; lo único que le importa es que el valor que reciba sepa hacer lo que ella necesita. Si el tipo sabe hacerlo, entra. Si no sabe, el compilador ni siquiera te deja conectarlo. Y lo mejor de todo: el tipo nunca tuvo que pedir permiso, nunca tuvo que registrarse en una lista, nunca tuvo que declarar "yo implemento esta interfaz". Simplemente sabe hacerlo, y eso basta.

Piensa en el problema que resuelven. Supón que estás escribiendo un programa sin interfaces y necesitas una función que calcule el área de figuras geométricas. Empiezas con el rectángulo y escribes una función `AreaRectangulo`. Luego necesitas el círculo y escribes `AreaCirculo`. Después el triángulo y escribes `AreaTriangulo`. Cada figura nueva te obliga a escribir una función nueva, y cada lugar de tu código donde quieras trabajar con "una figura cualquiera" se convierte en un enredo de condicionales preguntando qué tipo de figura es. Tu código se vuelve rígido como una tabla: cada cambio duele, cada figura nueva te obliga a tocar diez lugares distintos. Eso es programar contra tipos concretos, y es una receta para el dolor de cabeza.

Con interfaces el enfoque se voltea por completo. En lugar de preguntar "¿qué eres?", tu código pregunta "¿qué sabes hacer?". Defines una interfaz `Figura` que dice: "para ser figura hay que saber calcular área y perímetro". Y a partir de ese momento escribes funciones que reciben una `Figura`, sin importar cuál. El rectángulo, el círculo y el triángulo cumplen el acuerdo por su cuenta, cada uno a su manera. Cuando mañana inventes el hexágono, no tocas ni una línea del código existente: simplemente le enseñas al hexágono a calcular su área y su perímetro, y automáticamente entra al sistema. Eso es polimorfismo, y en Go se logra sin ceremonias, sin palabras clave raras y sin jerarquías de herencia complicadas.

Aquí viene la primera gran diferencia con otros lenguajes. Si vienes de Java o de C#, estás acostumbrado a declarar explícitamente que una clase implementa una interfaz, con una palabra como `implements`. En Go eso no existe. No hay `implements`, no hay registro, no hay declaración de intenciones. La relación entre un tipo y una interfaz es implícita: si el tipo tiene todos los métodos que la interfaz pide, con las firmas exactas, entonces el tipo satisface la interfaz. Punto. El compilador lo descubre solo. Esto tiene una consecuencia hermosa: puedes definir una interfaz hoy para tipos que se escribieron hace años, incluso tipos de bibliotecas de terceros que jamás oyeron hablar de tu interfaz. Si esos tipos ya saben hacer lo que pides, ya son compatibles. Es como descubrir que tu vieja lámpara de la abuela también entra en el contacto nuevo: nadie tuvo que modificar la lámpara.

Esta idea tiene nombre en el mundo de la programación: `duck typing`, que viene del dicho "si camina como pato, nada como pato y grazna como pato, entonces es un pato". Go lo lleva al nivel del compilador: el pato no necesita credencial de pato, basta con que grazne. Pero ojo, no es el `duck typing` desordenado de los lenguajes dinámicos, donde te enteras del error hasta que el programa truena en producción. En Go el compilador verifica todo antes de que tu programa corra. Tienes la flexibilidad del pato con la seguridad del acero.

La segunda gran idea de esta unidad es la composición. Muchos lenguajes resuelven la reutilización de código con herencia: creas una clase `Animal` y de ella heredan `Perro` y `Gato`, y todo lo que está en `Animal` baja automáticamente a los hijos. Go no tiene herencia. Punto. No hay `extends`, no hay clases padre, no hay jerarquías. En su lugar, Go te da la composición: armas tipos nuevos metiendo structs dentro de structs, como piezas de Lego. Un `Empleado` tiene una `Persona` adentro; un `Auto` tiene un `Motor` adentro. Y gracias al `embedding`, los métodos del struct interno se "promueven" y parece que pertenecen al struct externo. Es herencia sin sus problemas: sin el temido "problema del diamante", sin jerarquías frágiles donde cambiar al abuelo rompe a los nietos. La filosofía de Go es clara: prefiere componer piezas pequeñas y claras antes que heredar de monstruos gigantes.

¿Por qué te debe importar todo esto? Porque las interfaces son el pegamento de los programas grandes en Go. La biblioteca estándar está construida sobre ellas: cuando lees un archivo, escribes en la red o imprimes un valor con `fmt.Println`, por debajo hay interfaces trabajando. El famoso tipo error no es más que una interfaz con un solo método. Cuando entiendas interfaces, vas a entender por qué el código Go profesional se ve como se ve: funciones pequeñas que reciben interfaces pequeñas, tipos concretos que ni se enteran de las interfaces que satisfacen, y programas que crecen agregando piezas en lugar de modificando las existentes. Es la diferencia entre construir con plastilina y construir con Lego: con Lego agregas bloques sin deformar los que ya pusiste.

En esta unidad vamos a recorrer el tema paso a paso. Primero veremos qué es exactamente una interfaz y cómo se declara, con su sintaxis y sus reglas. Luego entenderemos a fondo la implementación implícita, que es el corazón del asunto. Conoceremos a la interfaz vacía, conocida como `any`, que acepta literalmente cualquier valor, y aprenderemos a recuperar el valor original con `type assertions` y `type switches` sin que el programa truene. Después visitaremos las interfaces más famosas de la biblioteca estándar: `error`, `fmt.Stringer`, `io.Reader` e `io.Writer`, porque verlas en acción te va a enseñar más que diez definiciones. Luego entraremos a la composición con structs embebidos y la promoción de métodos. Hablaremos del principio de las interfaces pequeñas, de los errores más comunes que cometen hasta los programadores con experiencia, y de las buenas prácticas que separan el código decente del código elegante. Cerraremos con ejemplos paso a paso, ejercicios con solución y un mini sistema completo de notificaciones para que veas todo junto y funcionando.

Mira este primer adelanto para que vayas calentando motores. Define una interfaz con un solo método, dos tipos que la cumplen sin saberlo, y una función que trabaja con la interfaz:

```

package main

import "fmt"

// Hablador es una interfaz: cualquier tipo con el metodo Hablar
// automaticamente la satisface, sin declararlo.
type Hablador interface {
    Hablar() string
}

type Perro struct {
    Nombre string
}

func (p Perro) Hablar() string {
    return "Guau, soy " + p.Nombre
}

type Persona struct {
    Nombre string
}

func (p Persona) Hablar() string {
    return "Hola, soy " + p.Nombre
}

// Presentar no conoce a Perro ni a Persona.
// Solo conoce la interfaz Hablador.
func Presentar(h Hablador) {
    fmt.Println(h.Hablar())
}

func main() {
    firulais := Perro{Nombre: "Firulais"}
    ana := Persona{Nombre: "Ana"}
    Presentar(firulais)
    Presentar(ana)
}

```

Fíjate en lo que NO aparece en ese código: en ningún lado dice que Perro implementa Hablador. Ni Persona. Y sin embargo Presentar los acepta a los dos, porque ambos tienen el método Hablar con la firma exacta. Si mañana creas un tipo Robot con su método Hablar, Presentar lo va a aceptar sin que modifiques ni una coma de Presentar. Esa es la magia: el código que usa la interfaz queda abierto a tipos que todavía no existen. Cuando un

programador de otro lenguaje ve esto por primera vez, suele quedarse callado un momento. Es normal. Respira, sigue leyendo, y al final de la unidad esto te va a parecer lo más natural del mundo.

Un último aviso antes de entrar en materia: las interfaces son una herramienta de diseño, no un adorno. Un error clásico de principiante es crear interfaces para todo, desde el primer día, "por si acaso". La comunidad Go tiene un consejo sabio al respecto: no diseñes con interfaces desde el inicio; escribe primero el código concreto, y cuando veas que dos o más tipos necesitan tratarse igual, ahí extraes la interfaz. Las interfaces se descubren, no se inventan. Con eso en mente, empecemos.

Explicación detallada

Qué es una interfaz y cómo se declara

Una interfaz en Go es un tipo que describe un conjunto de firmas de métodos. No dice cómo se hace nada, solo dice qué se debe saber hacer. Es un contrato de comportamiento puro: sin datos, sin implementación, solo promesas. La sintaxis es directa: usas la palabra clave `type`, le pones un nombre y luego la palabra `interface` seguida de las firmas de los métodos entre llaves.

```
type Encendible interface {  
    Encender()  
    Apagar()  
}
```

Esa interfaz dice: "cualquiera que quiera ser `Encendible` debe tener un método `Encender` sin parámetros y sin valor de retorno, y un método `Apagar` igual". Fíjate en que los métodos se escriben solo con su firma: nombre, parámetros y tipos de retorno. Nada de cuerpos, nada de llaves con código adentro. La interfaz no sabe ni le interesa cómo enciendes o apagas; eso es problema de cada tipo.

Por convención, los nombres de interfaces en Go suelen terminar en "er" cuando describen una sola acción, tomando el verbo del método: `Reader` (el que lee), `Writer` (el que escribe), `Stringer` (el que se convierte a texto), `Error` (bueno, esa es la excepción famosa). No es una regla del compilador, es una convención de la comunidad que hace el código más legible. Si tu interfaz tiene un método `Hablar`, llamarla `Hablador` se siente natural. Si tiene varios métodos relacionados, usa un nombre que describa el rol: `Figura`, `Notificador`, `Almacenable`.

Un detalle importante: las interfaces son tipos de verdad. Puedes declarar variables de tipo interfaz, recibirlas como parámetros, regresarlas desde funciones y guardarlas en estructuras de datos. Una variable de tipo interfaz puede contener cualquier valor cuyo tipo satisfaga la interfaz. Eso es lo que permite el polimorfismo: una sola variable, muchos tipos posibles adentro.

Implementación implícita: no existe "implements"

Este es el concepto central de toda la unidad, así que vamos a masticarlo bien. En Java escribes `class Perro implements Hablador` y el compilador registra la relación. En Go no hay nada parecido. Tú simplemente le das a tu tipo los métodos necesarios, y en el momento en que el conjunto de métodos de tu tipo cubre todo lo que la interfaz pide, tu tipo satisface la interfaz automáticamente. Nadie lo declaró, nadie lo firmó, simplemente es.

Esto tiene tres consecuencias prácticas enormes. Primera: desacoplamiento total. El tipo no necesita conocer la interfaz. Puedes escribir un tipo hoy y dentro de dos años alguien define una interfaz que tu tipo ya satisface sin que tengas que tocar tu código. Segunda: puedes crear interfaces para código que no es tuyo. Si una biblioteca de terceros tiene un tipo con los métodos que necesitas, defines tu interfaz pequeña y ese tipo extranjero entra a tu sistema sin modificaciones. Tercera: tus funciones pueden pedir exactamente lo que necesitan, ni más ni menos, definiendo la interfaz en el lugar donde se usa.

Esa tercera consecuencia merece un ejemplo mental. Supón que escribes una función que guarda datos y necesita algo que sepa escribir bytes. En lugar de recibir un tipo concreto como `*os.File`, defines en tu paquete una interfaz mínima:

```
type escritor interface {
    Write(p []byte) (n int, err error)
}

func guardar(e escritor, datos []byte) error {
    _, err := e.Write(datos)
    return err
}
```

Ahora `guardar` acepta un archivo, un buffer de memoria, una conexión de red o cualquier cosa que sepa escribir bytes. Y nota algo sutil pero poderoso: la interfaz está en minúsculas, es privada del paquete. No necesitas exportarla para que funcione. Las interfaces pequeñas y locales son una marca del buen código Go.

¿Cómo verifica el compilador la satisfacción? Compara firmas método por método: mismo nombre, mismos tipos de parámetros en el mismo orden, mismos tipos de retorno. Si falta un método o una firma no coincide exactamente, el tipo no satisface la interfaz y el compilador te lo dice con un error claro en la línea donde intentas usarlo como tal. No hay sorpresas en tiempo de ejecución: todo se decide al compilar.

La interfaz vacía: any

¿Qué pasa si defines una interfaz sin ningún método? Obtienes la interfaz vacía, que se escribe `interface{}` en el Go clásico y `any` desde Go 1.18 (`any` es simplemente un alias, un nombre bonito para lo mismo). Una interfaz sin métodos no pide nada, así que todos los tipos la satisfacen automáticamente. Literalmente cualquier valor de Go puede guardarse en una variable de tipo `any`: enteros, textos, structs, slices, funciones, lo que sea.

```
var cosa any
cosa = 42
cosa = "hola"
cosa = true
cosa = []int{1, 2, 3}
```

¿Para qué sirve algo tan permisivo? Para los casos donde de verdad no sabes qué tipo va a llegar: la función `fmt.Println` recibe `any` porque puede imprimir lo que sea; los formatos de datos genéricos como JSON se decodifican en estructuras `any` porque un documento JSON puede traer cualquier forma; los contenedores genéricos guardan `any` cuando quieren aceptar de todo. Pero ojo: la interfaz vacía es como una caja sin etiqueta. Sabes que hay algo adentro, pero no sabes qué. Para usar el valor necesitas "abrir la caja" con una `type assertion`, que veremos enseguida. Por eso la regla de oro es: usa `any` solo cuando de verdad lo necesites, no como atajo para evitar pensar en los tipos. Un programa lleno de `any` es un programa donde el compilador ya no te puede ayudar, y eso es desperdiciar la mejor herramienta de Go.

Type assertions y la coma-ok

Cuando tienes un valor guardado en una interfaz y necesitas recuperar su tipo concreto, usas una `type assertion`. La sintaxis es `valor.(Tipo)`: le preguntas a la interfaz "¿lo que traes adentro es de este tipo?".

```
var cosa any = "hola mundo"
texto := cosa.(string)
fmt.Println(texto)
```

Eso funciona, pero tiene un filo peligroso: si el valor no es del tipo que afirmas, el programa entra en pánico y se muere ahí mismo. Si cosa trajera un entero y tú afirmas string, truena. Para evitarlo existe la forma segura con coma-ok, que regresa dos valores: el valor convertido y un booleano que dice si la afirmación fue correcta.

```
var cosa any = "hola mundo"
texto, ok := cosa.(string)
if ok {
    fmt.Println("Es texto:", texto)
} else {
    fmt.Println("No era texto")
}
```

La coma-ok es tu cinturón de seguridad. Úsala siempre que no estés absolutamente seguro del tipo que viene adentro, que en la práctica es casi siempre. El patrón `if valor, ok := x.(Tipo); ok` es tan común en Go que lo vas a ver en cada esquina del código profesional. Graba esto en tu memoria: la type assertion de un solo valor es para cuando estás seguro; la de dos valores es para cuando eres prudente. Y en programación, la prudencia paga.

Type switches

Cuando necesitas distinguir entre varios tipos posibles, encadenar ifs con assertions se vuelve tedioso. Para eso existe el type switch: un switch especial que examina el tipo dinámico guardado en una interfaz.

```
func describir(cosa any) {
    switch v := cosa.(type) {
    case int:
        fmt.Println("Es un entero:", v*2)
    case string:
        fmt.Println("Es un texto:", v)
    case bool:
        fmt.Println("Es un booleano:", v)
    default:
        fmt.Println("No se que tipo es")
    }
}
```

La sintaxis `cosa.(type)` solo está permitida dentro de un switch, y la variable `v` toma automáticamente el tipo concreto de cada caso. Dentro del `case int`, `v` es un `int` de verdad y puedes multiplicarlo; dentro del `case string`, `v` es un `string`. Es la forma idiomática de trabajar

con valores any cuando hay varios tipos posibles, como al procesar datos JSON o al escribir funciones genéricas de utilería. El caso default atrapa todo lo demás, igual que en un switch normal.

Las interfaces de la biblioteca estándar: error

La interfaz más famosa de Go tiene un solo método y la usas todos los días aunque no te des cuenta:

```
type error interface {  
    Error() string  
}
```

Eso es todo lo que es un error en Go: cualquier tipo con un método Error que regrese un string. Cuando una función regresa un error no nil, te está entregando un valor que satisface esa interfaz. Y aquí viene lo bonito: puedes crear tus propios tipos de error para dar información más rica. En lugar de regresar errors.New con un texto plano, defines un struct con los detalles del problema y le agregas el método Error.

```
type ErrorDivision struct {  
    Dividendo float64  
    Divisor   float64  
}  
  
func (e ErrorDivision) Error() string {  
    return "division no valida"  
}
```

Ahora quien reciba ese error puede hacer una type assertion para recuperar el struct original y saber exactamente qué dividendo y qué divisor causaron el problema. Los errores en Go no son códigos misteriosos: son valores con comportamiento, y entender que error es una interfaz te abre la puerta a manejarlos con elegancia en lugar de solo compararlos con nil.

fmt.Stringer: controla cómo se imprime tu tipo

Cuando pasas un valor a fmt.Println, la biblioteca usa la interfaz fmt.Stringer para decidir cómo mostrarlo:

```
type Stringer interface {  
    String() string  
}
```

Si tu tipo tiene un método `String` que regresa un `string`, `fmt` lo usa automáticamente para imprimirlo. Sin ese método, `fmt` muestra el struct con sus campos crudos, que a veces es ilegible. Con el método, tú decides la representación. Es un ejemplo perfecto del poder de las interfaces pequeñas: con un solo método, toda la maquinaria de impresión de Go se vuelve extensible a tus tipos sin que tengas que modificar la biblioteca estándar.

io.Reader e io.Writer: el dúo dinámico

Dos de las interfaces más importantes de Go son estas:

```
type Reader interface {  
    Read(p []byte) (n int, err error)  
}  
  
type Writer interface {  
    Write(p []byte) (n int, err error)  
}
```

Un `Reader` es cualquier cosa de la que se pueden leer bytes: un archivo, una conexión de red, un buffer en memoria, incluso un generador de datos inventado por ti. Un `Writer` es cualquier cosa a la que se le pueden escribir bytes: lo mismo, del otro lado. La genialidad está en las funciones que trabajan con ellas, como `io.Copy`, que copia todo lo que lee de un `Reader` a un `Writer` sin saber ni qué lee ni a dónde escribe:

```
func Copy(dst Writer, src Reader) (written int64, err error)
```

Con esa sola función puedes copiar de un archivo a otro, de la red a un archivo, de memoria a la red, o de tu `Reader` inventado a tu `Writer` inventado. Nadie tuvo que enseñarle a `io.Copy` sobre tus tipos. Esa es la interoperabilidad que las interfaces regalan: piezas de distintos autores encajan porque hablan el mismo protocolo mínimo.

Composición con structs embebidos

Go no tiene herencia, y eso es una decisión de diseño, no una carencia. En su lugar ofrece composición mediante structs embebidos: metes un struct dentro de otro sin darle nombre de campo, solo poniendo el tipo.


```

type Persona struct {
    Nombre string
    Edad    int
}

func (p Persona) Saludar() string {
    return "Hola, soy " + p.Nombre
}

type Empleado struct {
    Persona
    Puesto  string
    Sueldo  float64
}

```

Al embeber `Persona` dentro de `Empleado`, los campos y métodos de `Persona` se promueven: puedes escribir `empleado.Nombre` y `empleado.Saludar()` como si fueran directamente del `Empleado`. Por dentro, Go los busca en la `Persona` embebida. Esto se parece a la herencia, pero con una diferencia filosófica crucial: no hay una relación "es un", hay una relación "tiene un". Un `Empleado` tiene una `Persona` adentro, no es una versión especializada de `Persona`. Y si `Empleado` define su propio método `Saludar`, ese método sombrea al de `Persona` para el `Empleado`, sin afectar a `Persona`. Cada tipo sigue siendo independiente; no hay jerarquías frágiles donde un cambio en la clase abuela rompe a toda la descendencia.

El embedding también funciona con interfaces: puedes embeber una interfaz dentro de otra para componer interfaces grandes a partir de pequeñas, como hace `io.ReadWriter` que combina `Reader` y `Writer`. Es el mismo principio en todos los niveles: piezas pequeñas que se ensamblan.

Interfaces pequeñas: menos es más

Hay un principio famoso en la comunidad Go que dice: "cuanto más grande la interfaz, más débil la abstracción". Piénsalo con la analogía del enchufe: un enchufe estándar tiene dos o tres patitas y le sirve a millones de aparatos. Si el enchufe tuviera cincuenta patitas con formas rarísimas, casi ningún aparato podría conectarse. Igual pasa con las interfaces: una interfaz de un solo método (como `error`, `Stringer`, `Reader`) la pueden satisfacer miles de tipos distintos. Una interfaz de veinte métodos casi nadie la puede cumplir, y cada vez que le agregas un método rompes a todos los tipos que la satisfacían.

La práctica recomendada es diseñar interfaces mínimas, de uno a tres métodos, que describan exactamente lo que el consumidor necesita y nada más. Si una función solo necesita leer, que

pida un Reader, no un ReadWriter. Si solo necesita el método Error, que pida un error. Las interfaces grandes suelen ser señal de que estás modelando "lo que el tipo es" en lugar de "lo que el consumidor necesita", y eso invierte la filosofía de Go. Recuerda el consejo: primero escribe código concreto, y cuando detectes que varias partes necesitan tratar tipos distintos de la misma forma, extrae la interfaz mínima que capture ese uso. Las interfaces se descubren en el código que ya existe, no se decretan desde un diagrama.

http.Handler: la interfaz que mueve la web en Go

Si alguna vez te preguntaste cómo funciona un servidor web en Go, la respuesta cabe en una interfaz diminuta del paquete net/http:

```
type Handler interface {  
    ServeHTTP(ResponseWriter, *Request)  
}
```

Un solo método. Cualquier tipo que sepa responder a una petición HTTP (recibiendo un escritor de respuesta y la petición) es un Handler. El servidor de Go no conoce tus tipos, no le importa si tu handler es un struct, una función adaptada o un sistema completo: si cumple el contrato, atiende peticiones. Cuando escribes `http.Handle("/ruta", miHandler)`, le estás entregando al servidor una interfaz, y por dentro el servidor guarda handlers de todos los sabores en las mismas estructuras. Millones de servidores en producción funcionan así: una interfaz de un método sosteniendo toda la web de Go. Veremos un ejemplo completo en la sección de ejemplos, pero graba la lección: las interfaces más influyentes son las más pequeñas.

Mocks: probar tu código sin depender de lo real

Aquí está la razón práctica número uno por la que las interfaces dominan el código profesional: las pruebas. Supón que tu lógica de negocio guarda usuarios en una base de datos. Probar esa lógica contra la base real es lento, frágil y complicado. La solución es definir una interfaz pequeña para el almacenamiento:

```
type Guardador interface {  
    Guardar(usuario Usuario) error  
    Buscar(id string) (Usuario, error)  
}
```

Tu lógica recibe un Guardador, no la base de datos concreta. En producción le pasas la implementación real que habla con PostgreSQL. En las pruebas le pasas un mock: un tipo

falso, de mentira, que guarda los usuarios en un mapa en memoria y responde al instante. Tu lógica no nota la diferencia porque solo conoce la interfaz. Esto se llama inyección de dependencias y es pan de cada día en Go: las interfaces son las costuras por donde tu código se puede abrir para probarse por partes. Sin interfaces, probar es un martirio; con interfaces pequeñas, cada pieza se prueba aislada, rápido y sin sorpresas. Cuando veas un código Go difícil de probar, casi siempre es porque en algún lugar se ató a un tipo concreto donde debía pedir una interfaz.

El conjunto de métodos y las interfaces embebidas

Ya vimos que el conjunto de métodos de `T` incluye solo métodos con receptor de valor, y el de `*T` incluye todos. Esa distinción decide qué satisface cada interfaz, así que vale la pena tenerla siempre presente al diseñar tipos. Pero hay más: las interfaces también se pueden componer embebiendo unas dentro de otras, igual que los structs.

```
type Lector interface {
    Leer() string
}

type Escritor interface {
    Escribir(texto string)
}

// LectorEscritor combina ambas interfaces embebiendolas.
type LectorEscritor interface {
    Lector
    Escritor
}
```

Un tipo satisface `LectorEscritor` solo si tiene los tres métodos (los dos heredados más los suyos propios, aunque aquí no agrega ninguno nuevo). La biblioteca estándar hace esto todo el tiempo: `io.ReadWriter` combina `io.Reader` e `io.Writer`, y `http.ResponseWriter` embebe `io.Writer` para garantizar que toda respuesta sepa escribir bytes. Es composición aplicada a los contratos: interfaces pequeñas que se ensamblan en contratos mayores solo cuando hace falta, sin duplicar firmas y sin jerarquías rígidas.

Errores comunes

El primer error clásico es el del receptor de puntero contra receptor de valor. En Go, el conjunto de métodos de un tipo `T` incluye solo los métodos con receptor de valor, mientras que el conjunto de métodos de `*T` (el puntero) incluye tanto los de valor como los de puntero. En la

práctica esto significa: si defines el método con receptor de puntero, solo el puntero satisface la interfaz, no el valor. Mira la trampa:

```
type Hablador interface {
    Hablar() string
}

type Perro struct{ Nombre string }

func (p *Perro) Hablar() string {
    return "Guau"
}

var h Hablador = Perro{} // ERROR: Perro no tiene el metodo Hablar
var h2 Hablador = &Perro{} // OK: *Perro si lo tiene
```

El compilador te avisa, así que no es un error silencioso, pero confunde a todos al principio. La regla práctica: si alguno de tus métodos necesita receptor de puntero (porque modifica el struct), define todos los métodos de ese tipo con receptor de puntero y usa siempre el puntero. Consistencia te ahorra dolores de cabeza.

El segundo error clásico es confundir una interfaz nil con una interfaz que guarda un puntero nil. Una interfaz en Go tiene dos partes por dentro: el tipo concreto guardado y el valor. La interfaz es nil solo cuando ambas partes están vacías. Pero si guardas un puntero nil dentro de la interfaz, el tipo ya no está vacío, así que la interfaz NO es nil aunque el valor sí lo sea.

```
func crear() *Perro {
    return nil
}

var h Hablador = crear() // h guarda (*Perro)(nil)
if h == nil {
    // Esto NO se ejecuta: h no es nil
}
```

Este es uno de los bugs más famosos de Go y ha tumbado sistemas en producción. La lección: cuando una función regresa una interfaz (como error), regresa nil directamente, no un puntero nil disfrazado. Y cuando compares interfaces con nil, recuerda que estás preguntando por las dos partes internas, no solo por el valor.

Cuándo NO usar interfaces

Las interfaces son tan elegantes que dan ganas de usarlas para todo, y ese es precisamente el peligro. Un error muy común en programadores que recién las descubren es el "interfacismo": crear una interfaz por cada struct "por si acaso algún día hace falta". El resultado es código más difícil de leer, donde para entender qué hace una función tienes que cazar qué tipo concreto se esconde detrás de cada interfaz, sin ningún beneficio a cambio.

Hay señales claras de que una interfaz sobra. Primera: existe una sola implementación y no hay ninguna razón real para que exista otra. Si tu programa solo va a guardar usuarios en PostgreSQL y jamás en otro lado, una interfaz Guardador con una sola implementación no te da flexibilidad, te da indirección gratuita. Segunda: la interfaz duplica método por método a un único tipo concreto, como una sombra. Eso no es una abstracción, es burocracia. Tercera: la interfaz se definió antes de que existiera el código que la usa, basada en adivinanzas sobre el futuro. El futuro casi nunca llega como lo imaginaste.

La regla de oro de la comunidad Go es directa: no diseñes con interfaces desde el inicio. Escribe primero el código concreto, el más simple que funcione. Cuando aparezca la segunda implementación, o cuando necesites un mock para probar, o cuando dos partes del código necesiten tratar tipos distintos de la misma forma, ahí extraes la interfaz mínima. Las interfaces prematuras son como comprar un traje tres tallas más grande "por si creces": la mayoría del tiempo solo estorba. Y recuerda la otra cara: tampoco caigas en el any para todo. Entre el concreto y el genérico hay un punto justo, y ese punto se encuentra mirando el código real, no imaginando el futuro.

Receta práctica: diseñar con interfaces en 5 pasos

Para que todo lo de esta unidad no se quede en teoría, aquí tienes una receta concreta que puedes aplicar cada vez que sospeches que una interfaz te serviría. Paso uno: escribe el código concreto primero, sin interfaces. Haz que funcione con tipos reales, aunque se repita un poco. Paso dos: detecta el comportamiento común. ¿Hay dos o más tipos que se usan de la misma forma? ¿Una función que hace casi lo mismo con tipos distintos? ¿Una parte que quieres probar con un falso? Ahí hay una interfaz escondida. Paso tres: extrae la interfaz mínima en el lugar donde se usa. Pregúntate "¿qué métodos necesita exactamente este consumidor?" y define solo esos, ni uno más. Paso cuatro: ponle un nombre que describa lo que hace, con la terminación "er" cuando sea natural: Validador, Notificador, Guardador. Paso cinco: verifica que compila, corre las pruebas y documenta el contrato: qué debe hacer cada método y qué pasa con valores límite. Si en el paso dos no encuentras comportamiento común real, no hay interfaz: vuelve al código concreto sin culpa. Esta receta te protege de los dos extremos: el código rígido que nunca abstrae y el interfacismo que abstrae de más.

Buenas prácticas

Cierra esta sección con las costumbres que distinguen al programador Go maduro. Primera: acepta interfaces y regresa tipos concretos. Tus funciones deben pedir lo mínimo que necesitan (una interfaz pequeña) pero regresar algo específico y útil (un struct o puntero concreto). Así el que llama tiene toda la información y tú mantienes la flexibilidad. Segunda: define la interfaz donde se usa, no donde se implementa. El consumidor sabe lo que necesita; el productor no debería adivinarlo. Tercera: verifica la satisfacción en tiempo de compilación cuando te importe, con la declaración `var _ Hablador = (*Perro)(nil)`, que no hace nada en ejecución pero truena al compilar si `Perro` deja de cumplir. Cuarta: no abuses de `any`; cada `any` es un lugar donde el compilador deja de cuidarte. Quinta: mantén las interfaces pequeñas y dales nombres claros. Y sexta: documenta qué debe hacer cada método, incluyendo cómo se comporta con valores cero o nil, porque una interfaz es un contrato y los contratos buenos están por escrito.

Ejemplos paso a paso

Ejemplo 1: Figuras geométricas y polimorfismo

El ejemplo clásico para entender interfaces: varias figuras que saben calcular su área y su perímetro, cada una a su manera, y una sola función que trabaja con todas.

```
package main

import (
    "fmt"
    "math"
)

// Figura es la interfaz: todo lo que sepa dar area y perimetro
// puede usarse como Figura.
type Figura interface {
    Area() float64
    Perimetro() float64
}

type Rectangulo struct {
    Base, Altura float64
}

func (r Rectangulo) Area() float64 {
    return r.Base * r.Altura
```

```

}

func (r Rectangulo) Perimetro() float64 {
    return 2*r.Base + 2*r.Altura
}

type Circulo struct {
    Radio float64
}

func (c Circulo) Area() float64 {
    return math.Pi * c.Radio * c.Radio
}

func (c Circulo) Perimetro() float64 {
    return 2 * math.Pi * c.Radio
}

// Mostrar recibe la interfaz, no un tipo concreto.
func Mostrar(f Figura) {
    fmt.Printf("Area: %.2f, Perimetro: %.2f\n", f.Area(), f.Perimetro())
}

func main() {
    r := Rectangulo{Base: 4, Altura: 5}
    c := Circulo{Radio: 3}
    Mostrar(r)
    Mostrar(c)

    // Un slice de la interfaz guarda figuras distintas.
    figuras := []Figura{r, c}
    var total float64
    for _, f := range figuras {
        total += f.Area()
    }
    fmt.Printf("Area total: %.2f\n", total)
}

```

Observa tres cosas. Primera: Rectangulo y Circulo nunca mencionan a Figura, y aun así Mostrar los acepta. Segunda: el slice []Figura guarda tipos mezclados, algo imposible con un slice de un tipo concreto. Tercera: si mañana agregas un Triangulo con sus dos métodos, todo este código lo acepta sin cambios. Ese es el polimorfismo en Go: un solo código, muchos comportamientos.

Ejemplo 2: Un error personalizado con información útil

En lugar de regresar un texto plano, creamos un tipo de error que guarda los datos del problema para que quien lo reciba pueda inspeccionarlos.

```
package main

import "fmt"

// ErrorSaldo es un tipo de error con datos: el saldo y el monto.
type ErrorSaldo struct {
    Saldo float64
    Monto float64
}

// Al tener el metodo Error, ErrorSaldo satisface la interfaz error.
func (e ErrorSaldo) Error() string {
    return fmt.Sprintf("saldo insuficiente: tienes %.2f y quieres retirar %.2f",
        e.Saldo, e.Monto)
}

func retirar(saldo, monto float64) (float64, error) {
    if monto > saldo {
        return saldo, ErrorSaldo{Saldo: saldo, Monto: monto}
    }
    return saldo - monto, nil
}

func main() {
    _, err := retirar(100, 250)
    if err != nil {
        fmt.Println("Fallo:", err)
        // Recuperamos el tipo concreto para ver los datos.
        if detalle, ok := err.(ErrorSaldo); ok {
            fmt.Printf("Te faltaron %.2f\n", detalle.Monto-detalle.Saldo)
        }
    }
}
```

La función `retirar` regresa la interfaz `error`, pero por dentro viaja nuestro `struct` con toda la información. Quien recibe el error puede usarlo como un error normal (imprimirlo) o abrirlo con una `type assertion` para tomar decisiones con los datos. Así se construyen errores profesionales: mensaje legible para humanos y datos estructurados para el programa.

Ejemplo 3: `fmt.Stringer` para imprimir bonito

Sin `String`, `fmt` imprime tu struct con sus campos crudos. Con `String`, tú controlas la presentación.

```
package main

import "fmt"

type Punto struct {
    X, Y int
}

// String hace que Punto satisfaga fmt.Stringer.
func (p Punto) String() string {
    return fmt.Sprintf("(%d, %d)", p.X, p.Y)
}

func main() {
    p := Punto{X: 3, Y: 4}
    fmt.Println("El punto es", p)
    // Sin el metodo String imprimiria: El punto es {3 4}
    // Con el metodo imprime: El punto es (3, 4)
}
```

Este ejemplo muestra el poder de las interfaces de la biblioteca estándar: con un solo método, tu tipo se integra con toda la maquinaria de impresión de Go. No modificaste `fmt`, no pediste permiso, simplemente cumpliste el contrato y todo funcionó.

Ejemplo 4: Un `io.Reader` inventado por ti

Vamos a crear un `Reader` que genera la misma letra una y otra vez, y a usarlo con `io.Copy` para demostrar que encaja con la biblioteca estándar sin haberla tocado.

```

package main

import (
    "io"
    "os"
)

// Repetidor genera la letra indicada, N veces, como un Reader.
type Repetidor struct {
    Letra    byte
    Restantes int
}

func (r *Repetidor) Read(p []byte) (int, error) {
    if r.Restantes <= 0 {
        return 0, io.EOF
    }
    n := len(p)
    if n > r.Restantes {
        n = r.Restantes
    }
    for i := 0; i < n; i++ {
        p[i] = r.Letra
    }
    r.Restantes -= n
    return n, nil
}

func main() {
    rep := &Repetidor{Letra: 'A', Restantes: 10}
    // io.Copy solo conoce Reader y Writer: nuestro tipo encaja.
    io.Copy(os.Stdout, rep)
}

```

Al correrlo imprime AAAAAAAAAA. Nota que Read usa receptor de puntero porque modifica Restantes, así que solo *Repetidor satisface io.Reader y por eso creamos el valor con &. Y lo más importante: io.Copy no sabe nada de Repetidor; solo sabe que recibe un Reader. Tu tipo inventado trabaja codo a codo con código escrito hace años por el equipo de Go.

Ejemplo 5: Embedding y promoción de métodos

Veamos cómo un Empleado reutiliza todo lo de Persona sin herencia, y cómo puede personalizar comportamientos.

```

package main

import "fmt"

type Persona struct {
    Nombre string
    Edad   int
}

func (p Persona) Saludar() string {
    return "Hola, soy " + p.Nombre
}

func (p Persona) CumplirAños() {
    p.Edad++
}

type Empleado struct {
    Persona // struct embebido: sus campos y metodos se promueven
    Puesto  string
}

// Empleado sombrea Saludar con su propia version.
func (e Empleado) Saludar() string {
    return "Buenos días, soy " + e.Nombre + ", " + e.Puesto
}

func main() {
    emp := Empleado{
        Persona: Persona{Nombre: "Luis", Edad: 30},
        Puesto:   "Programador",
    }
    fmt.Println(emp.Saludar()) // version de Empleado
    fmt.Println(emp.Persona.Saludar()) // version original de Persona
    fmt.Println(emp.Nombre)           // campo promovido: no hace falta
    emp.Persona.Nombre
    fmt.Println(emp.Edad)             // campo promovido
}

```

El embedding te da reutilización sin ataduras: Empleado aprovecha los campos y métodos de Persona, pero puede sombrear los que quiera sin afectar a Persona ni a nadie más que embeba Persona. Compáralo con la herencia clásica, donde cambiar un método del padre puede romper silenciosamente a todos los hijos. Aquí cada tipo es dueño de su comportamiento.

Ejemplo 6: Type switch para procesar datos mixtos

Una función que recibe any y se comporta distinto según el tipo real que llegue, usando type switch.

```
package main

import "fmt"

func procesar(valor any) {
    switch v := valor.(type) {
    case int:
        fmt.Println("Entero al doble:", v*2)
    case float64:
        fmt.Println("Decimal al triple:", v*3)
    case string:
        fmt.Println("Texto en mayusculas:", v)
    case []int:
        suma := 0
        for _, n := range v {
            suma += n
        }
        fmt.Println("Suma del slice:", suma)
    default:
        fmt.Printf("Tipo no soportado: %T\n", v)
    }
}

func main() {
    procesar(21)
    procesar(2.5)
    procesar("hola")
    procesar([]int{1, 2, 3, 4})
    procesar(true)
}
```

El verbo %T en el caso default imprime el tipo dinámico del valor, útil para depurar. Este patrón aparece constantemente al trabajar con JSON decodificado, configuraciones genéricas o sistemas de plugins donde los datos llegan en una interfaz vacía y cada tipo merece un tratamiento distinto.

Ejemplo 7: Mini sistema de notificaciones con Email y SMS

El ejemplo completo que une todo: una interfaz Notificador, dos implementaciones distintas y un sistema que envía mensajes sin saber por qué canal salen.

```
package main

import "fmt"

// Notificador es la interfaz minima: solo pide saber notificar.
type Notificador interface {
    Notificar(destino, mensaje string) error
}

// Email envia por correo electronico.
type Email struct {
    Remitente string
}

func (e Email) Notificar(destino, mensaje string) error {
    fmt.Printf("[EMAIL] De: %s Para: %s | %s\n", e.Remitente, destino, mensaje)
    return nil
}

// SMS envia por mensaje de texto.
type SMS struct {
    Proveedor string
}

func (s SMS) Notificar(destino, mensaje string) error {
    fmt.Printf("[SMS] Via %s Para: %s | %s\n", s.Proveedor, destino, mensaje)
    return nil
}

// Sistema envia el mismo aviso por todos los canales registrados.
type Sistema struct {
    canales []Notificador
}

func (s *Sistema) Registrar(n Notificador) {
    s.canales = append(s.canales, n)
}

func (s *Sistema) Avisar(destino, mensaje string) {
    for _, c := range s.canales {
        if err := c.Notificar(destino, mensaje); err != nil {
```

```

        fmt.Println("Fallo un canal:", err)
    }
}

func main() {
    sistema := &Sistema{}
    sistema.Registrar(Email{Remitente: "alertas@academia.com"})
    sistema.Registrar(SMS{Proveedor: "Telcel"})

    sistema.Avisar("ana@correo.com", "Tu curso nuevo ya esta disponible")
}

```

Fíjate en el diseño: Sistema guarda un slice de la interfaz Notificador y no conoce a Email ni a SMS. Si mañana agregas WhatsApp o Push, solo creas el tipo con su método Notificar y lo registras; Sistema no cambia ni una línea. Además la interfaz es mínima (un solo método), siguiendo el principio de interfaces pequeñas. Este es el patrón que vas a repetir en sistemas reales: define lo mínimo que necesitas, deja que los tipos concretos cumplan el contrato por su cuenta, y tu código queda abierto a crecer sin romperse.

Ejemplo 8: Un servidor web con http.Handler

La prueba de fuego de las interfaces pequeñas: dos handlers distintos atendiendo rutas distintas, sin que el servidor sepa nada de ellos.

```

package main

import (
    "fmt"
    "net/http"
)

// Pagina es un handler: cualquier valor Pagina satisface http.Handler
// porque tiene el metodo ServeHTTP con la firma exacta.
type Pagina struct {
    Titulo    string
    Contenido string
}

func (p Pagina) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "=== "+p.Titulo+" ===")
    fmt.Fprintln(w, p.Contenido)
}

func main() {
    http.Handle("/bienvenida", Pagina{
        Titulo:    "Bienvenida",
        Contenido: "Hola, esta pagina la sirve un struct comun y corriente.",
    })
    http.Handle("/curso", Pagina{
        Titulo:    "Curso Go",
        Contenido: "Unidad 7: interfaces y composicion.",
    })
    fmt.Println("Servidor escuchando en http://localhost:8080")
    http.ListenAndServe(":8080", nil)
}

```

Corre el programa y visita las dos rutas en tu navegador: cada una responde con su contenido. El servidor, escrito hace años por el equipo de Go, atiende tus structs sin conocerlos porque cumplen el contrato de un solo método. Si quisieras una ruta que muestre la hora actual, crearías otro tipo con `ServeHTTP` y lo registrarías; el servidor no cambiaría ni una línea.

Ejemplo 9: Ordenar con `sort.Interface`

El paquete `sort` ordena cualquier colección cuyo tipo tenga tres métodos: `Len`, `Less` y `Swap`. Otro triunfo de la interfaz pequeña.

```

package main

import (
    "fmt"
    "sort"
)

type Persona struct {
    Nombre string
    Edad    int
}

// PorEdad es un slice de Persona que sabe ordenarse por edad.
type PorEdad []Persona

func (p PorEdad) Len() int           { return len(p) }
func (p PorEdad) Less(i, j int) bool { return p[i].Edad < p[j].Edad }
func (p PorEdad) Swap(i, j int)      { p[i], p[j] = p[j], p[i] }

func main() {
    gente := PorEdad{
        {Nombre: "Ana", Edad: 32},
        {Nombre: "Luis", Edad: 25},
        {Nombre: "Marta", Edad: 41},
    }
    sort.Sort(gente)
    for _, p := range gente {
        fmt.Printf("%s (%d)\n", p.Nombre, p.Edad)
    }
}

```

sort.Sort recibe la interfaz sort.Interface y ordena sin saber que está ordenando personas. Cambia Less para comparar por Nombre y obtienes orden alfabético sin tocar el algoritmo de ordenamiento. El algoritmo se escribió una vez; los criterios de orden los pones tú con tres métodos.

Ejemplo 10: Un mock para probar sin base de datos

La técnica profesional: tu lógica depende de una interfaz, y en la prueba le inyectas un falso.

```

package main

import (
    "errors"

```



```

    "fmt"
}

type Usuario struct {
    ID      string
    Nombre string
}

// Almacen es todo lo que la logica necesita: guardar y buscar.
type Almacen interface {
    Guardar(u Usuario) error
    Buscar(id string) (Usuario, error)
}

// Servicio es la logica de negocio: solo conoce la interfaz.
type Servicio struct {
    db Almacen
}

func (s Servicio) Registrar(u Usuario) error {
    if u.Nombre == "" {
        return errors.New("el nombre no puede estar vacio")
    }
    return s.db.Guardar(u)
}

// MockAlmacen es el falso para pruebas: un mapa en memoria.
type MockAlmacen struct {
    datos map[string]Usuario
}

func (m *MockAlmacen) Guardar(u Usuario) error {
    m.datos[u.ID] = u
    return nil
}

func (m *MockAlmacen) Buscar(id string) (Usuario, error) {
    u, ok := m.datos[id]
    if !ok {
        return Usuario{}, errors.New("no encontrado")
    }
    return u, nil
}

func main() {

```

```

// En produccion aqui iria la base real; en la prueba, el mock.
mock := &MockAlmacen{datos: make(map[string]Usuario)}
servicio := Servicio{db: mock}

err := servicio.Registrar(Usuario{ID: "1", Nombre: "Ana"})
fmt.Println("Registro:", err)

err = servicio.Registrar(Usuario{ID: "2", Nombre: ""})
fmt.Println("Registro vacio:", err)

u, _ := mock.Buscar("1")
fmt.Println("Encontrado:", u.Nombre)
}

```

Servicio jamás toca una base de datos real y aun así su lógica queda probada: el registro válido se guarda, el nombre vacío se rechaza. En un programa real, este main sería una función de prueba y el mock viviría en el archivo de tests. Así se prueba el 90 por ciento del código Go profesional: interfaces pequeñas como costuras, mocks como dobles de riesgo.

Ejemplo 11: Validadores encadenables

Una lista de reglas donde cada regla es un tipo distinto, todas detrás de una interfaz mínima. Agregar una regla nueva no toca el motor de validación.

```

package main

import (
    "errors"
    "fmt"
    "strings"
)

// Validador es todo lo que el motor necesita: algo que revise un texto.
type Validador interface {
    Validar(texto string) error
}

type NoVacio struct{}

func (NoVacio) Validar(texto string) error {
    if strings.TrimSpace(texto) == "" {
        return errors.New("el texto no puede estar vacio")
    }
    return nil
}

```

```

type LongitudMinima struct {
    Min int
}

func (l LongitudMinima) Validar(texto string) error {
    if len(texto) < l.Min {
        return fmt.Errorf("el texto debe tener al menos %d caracteres", l.Min)
    }
    return nil
}

// ValidarTodo aplica todas las reglas y se detiene en la primera falla.
func ValidarTodo(texto string, reglas []Validador) error {
    for _, r := range reglas {
        if err := r.Validar(texto); err != nil {
            return err
        }
    }
    return nil
}

func main() {
    reglas := []Validador{NoVacio{}, LongitudMinima{Min: 5}}
    for _, texto := range []string{"", "hola", "hola mundo"} {
        if err := ValidarTodo(texto, reglas); err != nil {
            fmt.Printf("%q -> invalido: %s\n", texto, err)
        } else {
            fmt.Printf("%q -> valido\n", texto)
        }
    }
}

```

El motor ValidarTodo no sabe cuántas reglas hay ni qué revisa cada una. Si mañana necesitas una regla SinNumeros, creas el tipo con su método Validar y lo agregas al slice. Este patrón de "lista de estrategias" aparece en validaciones, middlewares, filtros y pipelines de procesamiento: la interfaz es el enchufe, cada regla es un aparato distinto.

Ejemplo 12: Ordenar figuras por área (interfaces combinadas)

Cerramos los ejemplos uniendo dos ideas de la unidad: la interfaz Figura y la interfaz sort.Interface. Ordenamos figuras mezcladas por su área.

```

package main

```

```

import (
    "fmt"
    "math"
    "sort"
)

type Figura interface {
    Area() float64
    Nombre() string
}

type Rectangulo struct {
    Base, Altura float64
}

func (r Rectangulo) Area() float64 { return r.Base * r.Altura }
func (r Rectangulo) Nombre() string { return "rectangulo" }

type Circulo struct {
    Radio float64
}

func (c Circulo) Area() float64 { return math.Pi * c.Radio * c.Radio }
func (c Circulo) Nombre() string { return "circulo" }

// PorArea es un slice de Figura que sabe ordenarse por area.
type PorArea []Figura

func (p PorArea) Len() int { return len(p) }
func (p PorArea) Less(i, j int) bool { return p[i].Area() < p[j].Area() }
func (p PorArea) Swap(i, j int) { p[i], p[j] = p[j], p[i] }

func main() {
    figuras := PorArea{
        Circulo{Radio: 3},
        Rectangulo{Base: 4, Altura: 5},
        Circulo{Radio: 1},
    }
    sort.Sort(figuras)
    for _, f := range figuras {
        fmt.Printf("%s: area %.2f\n", f.Nombre(), f.Area())
    }
}

```

Fíjate en las capas: `Figura` abstrae las formas, `sort.Interface` abstrae el ordenamiento, y `PorArea` las une. Ninguna de las tres capas conoce los detalles de las otras. Este es el diseño Go en su máxima expresión: piezas mínimas que se combinan para lograr algo sofisticado sin que ninguna pieza sea complicada.

Práctica

Ejercicio 1: La interfaz Sonido

Define una interfaz `Sonido` con un método `Emitir() string`. Crea los tipos `Perro` y `Gato` que la satisfagan, y escribe una función `Coro` que reciba un slice de `Sonido` e imprima el sonido de cada uno.

Solución explicada: la interfaz pide un solo método, así que cualquier tipo con `Emitir() string` entra automáticamente. El slice `[]Sonido` permite mezclar perros y gatos, y la función `Coro` no necesita saber qué tipos hay adentro.

```

package main

import "fmt"

type Sonido interface {
    Emitir() string
}

type Perro struct{ Nombre string }

func (p Perro) Emitir() string { return "Guau" }

type Gato struct{ Nombre string }

func (g Gato) Emitir() string { return "Miau" }

func Coro(sonidos []Sonido) {
    for _, s := range sonidos {
        fmt.Println(s.Emitir())
    }
}

func main() {
    Coro([]Sonido{Perro{Nombre: "Firulais"}, Gato{Nombre: "Mishi"}})
}

```

Ejercicio 2: Libro que se imprime bonito

Crea un struct Libro con campos Titulo, Autor y Paginas. Implementa el método String() string para que satisfaga fmt.Stringer e imprima algo como: "Cien años de soledad" por Gabriel García Márquez (417 páginas).

Solución explicada: basta agregar el método String con la firma exacta. En cuanto existe, fmt.Println lo usa solo. Nota que en el código el texto va sin acentos porque el código debe ser ASCII puro; en un programa real podrías escribir los acentos en los datos sin problema, pero aquí mantenemos la regla del libro.

```

package main

import "fmt"

type Libro struct {
    Titulo  string
    Autor   string
    Paginas int
}

func (l Libro) String() string {
    return fmt.Sprintf("%q por %s (%d paginas)", l.Titulo, l.Autor, l.Paginas)
}

func main() {
    libro := Libro{Titulo: "El principito", Autor: "Antoine de Saint-Exupery",
Paginas: 96}
    fmt.Println(libro)
}

```

El verbo %q pone el título entre comillas dobles automáticamente. Al correrlo verás la ficha formateada en lugar del struct crudo {El principito Antoine de Saint-Exupery 96}.

Ejercicio 3: Error de temperatura

Define un tipo ErrorTemperatura con los campos Grados y Limite. Implementa el método Error() string para que satisfaga la interfaz error con un mensaje descriptivo. Escribe una función verificar que regrese ese error si los grados superan el límite.

Solución explicada: el método Error convierte tu struct en un error válido. La función verificar regresa la interfaz error, así que quien la llame trabaja con el error de forma estándar, pero puede abrirlo con una assertion para leer los grados exactos.

```

package main

import "fmt"

type ErrorTemperatura struct {
    Grados float64
    Limite float64
}

func (e ErrorTemperatura) Error() string {
    return fmt.Sprintf("temperatura %.1f supera el limite de %.1f",
        e.Grados, e.Limite)
}

func verificar(grados, limite float64) error {
    if grados > limite {
        return ErrorTemperatura{Grados: grados, Limite: limite}
    }
    return nil
}

func main() {
    if err := verificar(39.5, 37.5); err != nil {
        fmt.Println("Alerta:", err)
        if det, ok := err.(ErrorTemperatura); ok {
            fmt.Printf("Exceso de %.1f grados\n", det.Grados-det.Limite)
        }
    }
}

```

Ejercicio 4: Vehículo eléctrico con embedding

Crea un struct Vehiculo con campo Marca y método Describir() string que diga la marca. Crea un struct AutoElectrico que embeba Vehiculo y agregue el campo AutonomiaKm. Sombrea el método Describir en AutoElectrico para que incluya la autonomía. Demuestra la promoción llamando al método original mediante el campo embebido.

Solución explicada: al embeber Vehiculo, sus campos y métodos se promueven a AutoElectrico. Al definir Describir también en AutoElectrico, esa versión sombrea a la promovida, pero la original sigue accesible con auto.Vehiculo.Describir().


```

package main

import "fmt"

type Vehiculo struct {
    Marca string
}

func (v Vehiculo) Describir() string {
    return "Vehiculo marca " + v.Marca
}

type AutoElectrico struct {
    Vehiculo
    AutonomiaKm int
}

func (a AutoElectrico) Describir() string {
    return fmt.Sprintf("Auto electrico %s con %d km de autonomia",
        a.Marca, a.AutonomiaKm)
}

func main() {
    auto := AutoElectrico{Vehiculo: Vehiculo{Marca: "Tesla"}, AutonomiaKm: 500}
    fmt.Println(auto.Describir())
    fmt.Println(auto.Vehiculo.Describir())
    fmt.Println("Marca promovida:", auto.Marca)
}

```

Ejercicio 5: Clasificador con type switch

Escribe una función clasificar que reciba un any y, usando un type switch, imprima: para int si es par o impar, para string su longitud, para bool su negación, y para cualquier otro tipo un mensaje con el tipo usando %T.

Solución explicada: el type switch con valor.(type) nos da en cada caso la variable ya con el tipo concreto, lista para operar. El default con %T cubre lo inesperado sin que el programa truene.

```

package main

import "fmt"

func clasificar(valor any) {
    switch v := valor.(type) {
    case int:
        if v%2 == 0 {
            fmt.Println(v, "es par")
        } else {
            fmt.Println(v, "es impar")
        }
    case string:
        fmt.Printf("%q tiene %d letras\n", v, len(v))
    case bool:
        fmt.Println("Su negacion es", !v)
    default:
        fmt.Printf("Tipo %T no clasificado\n", v)
    }
}

func main() {
    clasificar(42)
    clasificar(7)
    clasificar("hola")
    clasificar(true)
    clasificar(3.14)
}

```

Ejercicio 6: Sistema de pagos (reto integrador)

Define una interfaz Pagador con un método Pagar(monto float64) error. Implementa Efectivo (siempre funciona e imprime el cambio si se paga de más) y Tarjeta (falla si el monto supera su Limite). Escribe una función Cobrar que reciba un Pagador y un monto, intente el pago y reporte el resultado. Prueba ambos.

Solución explicada: este ejercicio junta interfaz mínima, tipos concretos que la satisfacen sin declararlo, y manejo de errores mediante la interfaz error. Cobrar no conoce ni a Efectivo ni a Tarjeta: solo sabe que le dieron algo que sabe pagar.

```

package main

import (
    "errors"
    "fmt"
)

type Pagador interface {
    Pagar(monto float64) error
}

type Efectivo struct{}

func (Efectivo) Pagar(monto float64) error {
    fmt.Printf("Pago en efectivo: %.2f recibido\n", monto)
    return nil
}

type Tarjeta struct {
    Limite float64
}

func (t Tarjeta) Pagar(monto float64) error {
    if monto > t.Limite {
        return errors.New("monto supera el limite de la tarjeta")
    }
    fmt.Printf("Pago con tarjeta por %.2f aprobado\n", monto)
    return nil
}

func Cobrar(p Pagador, monto float64) {
    if err := p.Pagar(monto); err != nil {
        fmt.Println("Cobro fallido:", err)
        return
    }
    fmt.Println("Cobro exitoso")
}

func main() {
    Cobrar(Efectivo{}, 150)
    Cobrar(Tarjeta{Limite: 500}, 300)
    Cobrar(Tarjeta{Limite: 500}, 800)
}

```

Si este ejercicio te salió sin mirar la solución, ya piensas con interfaces: diseñas el contrato mínimo, los tipos cumplen por su cuenta y el código que cobra queda abierto a cualquier medio de pago futuro.

Ejercicio 7: Figuras que se dibujan

Define una interfaz Dibujable con un método Dibujar() string. Crea los tipos Circulo (con campo Radio) y Cuadrado (con campo Lado) que la satisfagan, cada uno regresando una descripción de su dibujo. Escribe una función Galeria que reciba un slice de Dibujable e imprima el dibujo de cada figura numerado.

Solución explicada: otra vuelta al polimorfismo, ahora con dibujo en lugar de área. La función Galeria itera la interfaz y numera; no sabe ni le importa cuántos tipos de figura existen.

```

package main

import "fmt"

type Dibujable interface {
    Dibujar() string
}

type Circulo struct {
    Radio float64
}

func (c Circulo) Dibujar() string {
    return fmt.Sprintf("Circulo de radio %.1f", c.Radio)
}

type Cuadrado struct {
    Lado float64
}

func (c Cuadrado) Dibujar() string {
    return fmt.Sprintf("Cuadrado de lado %.1f", c.Lado)
}

func Galeria(figuras []Dibujable) {
    for i, f := range figuras {
        fmt.Printf("%d. %s\n", i+1, f.Dibujar())
    }
}

func main() {
    Galeria([]Dibujable{
        Circulo{Radio: 5},
        Cuadrado{Lado: 4},
        Circulo{Radio: 2.5},
    })
}

```

Ejercicio 8: Registrador con dos destinos

Define una interfaz Registrador con un método Registrar(mensaje string). Implementa Consola (imprime el mensaje con prefijo "[LOG]") y Memoria (guarda los mensajes en un slice interno, con un método Ver() []string para consultarlos). Escribe una función Procesar que reciba un

Registrador y registre tres pasos de un proceso. Pruébala con ambos y verifica que Memoria guardó los tres mensajes.

Solución explicada: Procesar solo conoce la interfaz, así que el mismo código sirve para imprimir en pantalla o para acumular en memoria (útil en pruebas, donde quieres verificar qué se registró sin leer la consola). Memoria usa receptor de puntero porque modifica su slice.

```

package main

import "fmt"

type Registrador interface {
    Registrar(mensaje string)
}

type Consola struct{}

func (Consola) Registrar(mensaje string) {
    fmt.Println("[LOG]", mensaje)
}

type Memoria struct {
    mensajes []string
}

func (m *Memoria) Registrar(mensaje string) {
    m.mensajes = append(m.mensajes, mensaje)
}

func (m *Memoria) Ver() []string {
    return m.mensajes
}

func Procesar(r Registrador) {
    r.Registrar("Iniciando proceso")
    r.Registrar("Paso intermedio completado")
    r.Registrar("Proceso terminado")
}

func main() {
    fmt.Println("--- A consola ---")
    Procesar(Consola{})

    fmt.Println("--- A memoria ---")
    mem := &Memoria{}
    Procesar(mem)
    fmt.Println("Mensajes guardados:", len(mem.Ver()))
    for _, m := range mem.Ver() {
        fmt.Println(" *", m)
    }
}

```

Nota que Memoria se pasa como &Memoria{} porque su método usa receptor de puntero: solo el puntero satisface la interfaz. Si intentaras pasar Memoria{} por valor, el compilador te detendría, que es exactamente el error común que estudiamos.

Ejercicio 9: Descuentos intercambiables

Define una interfaz Descuento con un método Aplicar(precio float64) float64 que regrese el precio con el descuento aplicado. Implementa DescuentoFijo (resta una cantidad) y DescuentoPorcentaje (resta un porcentaje). Escribe una función PrecioFinal que reciba el precio y un Descuento, y prueba ambas implementaciones con un precio de 1000.

Solución explicada: la interfaz captura "saber aplicar un descuento" sin importar la estrategia. PrecioFinal trabaja con la abstracción; las estrategias concretas son intercambiables. Es el mismo patrón de los validadores del Ejemplo 11, aplicado a otro dominio.


```

package main

import "fmt"

type Descuento interface {
    Aplicar(precio float64) float64
}

type DescuentoFijo struct {
    Monto float64
}

func (d DescuentoFijo) Aplicar(precio float64) float64 {
    final := precio - d.Monto
    if final < 0 {
        return 0
    }
    return final
}

type DescuentoPorcentaje struct {
    Porc float64
}

func (d DescuentoPorcentaje) Aplicar(precio float64) float64 {
    return precio * (1 - d.Porc/100)
}

func PrecioFinal(precio float64, d Descuento) float64 {
    return d.Aplicar(precio)
}

func main() {
    fmt.Println("Fijo de 200:", PrecioFinal(1000, DescuentoFijo{Monto: 200}))
    fmt.Println("20 porciento:", PrecioFinal(1000, DescuentoPorcentaje{Porc: 20}))
}

```

Ejercicio 10: Sistema de archivos virtual (reto integrador)

Define una interfaz `Nodo` con métodos `Nombre() string` y `Tamano() int`. Implementa `Archivo` (con nombre y bytes) y `Directorio` (con nombre y un slice de `Nodo` hijos). El tamaño de un `Directorio` es la suma recursiva de los tamaños de sus hijos. Construye una estructura con un directorio raíz que contenga dos archivos y un subdirectorio con otro archivo, e imprime el

tamaño total. Como reto extra, escribe una función Imprimir que muestre el árbol con indentación.

Solución explicada: este es el patrón Composite, una joya del diseño con interfaces. Directorio guarda []Nodo, y como Directorio también es un Nodo, los directorios pueden anidarse sin límite. El método Tamano se llama a sí mismo a través de la interfaz: cada nodo resuelve su parte y el total emerge solo. Nadie necesita saber si un hijo es archivo o directorio.

```
package main

import (
    "fmt"
    "strings"
)

type Nodo interface {
    Nombre() string
    Tamano() int
}

type Archivo struct {
    nombre string
    bytes  int
}

func (a Archivo) Nombre() string { return a.nombre }
func (a Archivo) Tamano() int    { return a.bytes }

type Directorio struct {
    nombre string
    hijos  []Nodo
}

func (d Directorio) Nombre() string { return d.nombre }

func (d Directorio) Tamano() int {
    total := 0
    for _, h := range d.hijos {
        total += h.Tamano()
    }
    return total
}

// Imprimir muestra el arbol con indentacion según la profundidad.
func Imprimir(n Nodo, profundidad int) {
```

```

    fmt.Printf("%s%s (%d bytes)\n",
        strings.Repeat(" ", profundidad), n.Nombre(), n.Tamano())
    if dir, ok := n.(Directorio); ok {
        for _, h := range dir.hijos {
            Imprimir(h, profundidad+1)
        }
    }
}

func main() {
    raiz := Directorio{
        nombre: "raiz",
        hijos: []Nodo{
            Archivo{nombre: "notas.txt", bytes: 120},
            Archivo{nombre: "foto.jpg", bytes: 2048},
            Directorio{
                nombre: "docs",
                hijos: []Nodo{
                    Archivo{nombre: "manual.pdf", bytes: 5120},
                },
            },
        },
    }
    Imprimir(raiz, 0)
    fmt.Println("Total:", raiz.Tamano(), "bytes")
}

```

Observa el type assertion en Imprimir: necesitamos saber si el nodo es un Directorio para recorrer sus hijos, porque la interfaz Nodo no expone los hijos (a propósito: la interfaz es mínima). El total debe dar 7288 bytes. Si llegaste hasta aquí resolviendo los ejercicios, ya no solo entiendes interfaces: las diseñas.

Ejercicio 11: Biblioteca prestable

Define una interfaz Prestable con métodos Titulo() string, Prestar() error, Devolver() y Prestado() bool. Implementa el tipo Libro con receptor de puntero (porque prestar modifica el estado). Escribe una función ProbarPrestamo que reciba un Prestable, intente prestarlo dos veces (la segunda debe fallar), lo devuelva y confirme que quedó disponible.

Solución explicada: el estado (prestado o no) obliga al receptor de puntero, y por tanto solo *Libro satisface la interfaz. La función ProbarPrestamo ejercita el ciclo de vida completo usando solo la abstracción: podría recibir cualquier cosa prestable, no solo libros.

```
package main
```

```

import (
    "errors"
    "fmt"
)

type Prestable interface {
    Titulo() string
    Prestar() error
    Devolver()
    Prestado() bool
}

type Libro struct {
    titulo    string
    prestado  bool
}

func NuevoLibro(titulo string) *Libro {
    return &Libro{titulo: titulo}
}

func (l *Libro) Titulo() string { return l.titulo }

func (l *Libro) Prestar() error {
    if l.prestado {
        return errors.New("ya esta prestado")
    }
    l.prestado = true
    return nil
}

func (l *Libro) Devolver()      { l.prestado = false }
func (l *Libro) Prestado() bool { return l.prestado }

func ProbarPrestamo(p Prestable) {
    fmt.Println("Solicitando:", p.Titulo())
    if err := p.Prestar(); err != nil {
        fmt.Println("Fallo:", err)
        return
    }
    fmt.Println("Prestado con exito")
    if err := p.Prestar(); err != nil {
        fmt.Println("Segundo intento:", err)
    }
}

```

```

    p.Devolver()
    fmt.Println("Devuelto. Disponible:", !p.Prestado())
}

func main() {
    ProbarPrestamo(NuevoLibro("El llano en llamas"))
}

```

Si intentaras pasar `Libro{}` por valor en lugar de `NuevoLibro`, el compilador se quejaría: otra aparición del error clásico del receptor de puntero, ahora en un caso real con estado mutable. Las interfaces no solo sirven para comportamientos sin estado; también modelan objetos con ciclo de vida, siempre que respetes las reglas del receptor.

Ejercicio 12: Funciones como handlers (el adaptador)

En Go hasta las funciones pueden tener métodos. Define un tipo `ManejadorFunc` basado en `func(http.ResponseWriter, *http.Request)` y dale el método `ServeHTTP` que simplemente se llame a sí mismo. Registra dos rutas usando funciones anónimas convertidas a `ManejadorFunc`. Este es exactamente el patrón de `http.HandlerFunc` de la biblioteca estándar.

Solución explicada: el adaptador convierte una función común en un valor que satisface la interfaz `http.Handler`. Es la prueba definitiva de que las interfaces describen comportamiento, no jerarquías: hasta una función, que no es un struct, puede "implementar" una interfaz con un método.

```

package main

import (
    "fmt"
    "net/http"
    "time"
)

// ManejadorFunc es un tipo funcion adaptado a http.Handler.
type ManejadorFunc func(w http.ResponseWriter, r *http.Request)

func (f ManejadorFunc) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    f(w, r)
}

func main() {
    http.Handle("/hora", ManejadorFunc(func(w http.ResponseWriter, r *http.Request) {
        {
            fmt.Fprintln(w, "Son las", time.Now().Format("15:04:05"))
        })
    })
    http.Handle("/saludo", ManejadorFunc(func(w http.ResponseWriter, r
    *http.Request) {
        fmt.Fprintln(w, "Hola desde una funcion adaptada")
    })
    fmt.Println("Servidor en http://localhost:8080")
    http.ListenAndServe(":8080", nil)
}

```

Este patrón aparece en todo el ecosistema Go cada vez que una interfaz de un solo método necesita una implementación rápida: en lugar de crear un struct solo para tener el método, adaptas una función. Cuando veas tipos cuyo único propósito es envolver una función para satisfacer una interfaz, estás viendo este adaptador en acción.

Resumen de la unidad

En esta unidad aprendiste que una interfaz en Go es un contrato de comportamiento: un conjunto de firmas de métodos sin implementación. Viste que los tipos satisfacen las interfaces de forma implícita, sin declarar nada parecido a implements, y que el compilador verifica la compatibilidad comparando firmas método por método. Conociste la interfaz vacía any, que acepta cualquier valor, y las herramientas para trabajar con ella de forma segura: las type assertions con coma-ok y los type switches.

Exploraste las interfaces más importantes de la biblioteca estándar: `error`, que convierte cualquier tipo con un método `Error` en un error válido; `fmt.Stringer`, que te da control sobre cómo se imprimen tus tipos; e `io.Reader` e `io.Writer`, el dúo que permite que piezas de distintos autores trabajen juntas, como demostramos con `io.Copy` y nuestro `Reader` inventado. Aprendiste composición con structs embebidos: meter un struct dentro de otro para reutilizar campos y métodos mediante promoción, sombreando cuando necesites personalizar, sin las trampas de la herencia.

También interiorizaste principios de diseño: las interfaces pequeñas son abstracciones fuertes, se descubren en el código existente en lugar de inventarse desde un diagrama, las funciones aceptan interfaces y regresan tipos concretos, y la interfaz se define donde se usa. Te advertimos de los errores clásicos: el receptor de puntero que solo satisface la interfaz con el puntero, y la interfaz que guarda un puntero nil pero no es nil. Y cerraste con un sistema de notificaciones completo donde `Email` y `SMS` conviven detrás de una interfaz mínima, demostrando el polimorfismo real: código que acepta tipos que aún no existen.

La idea que debes llevarte grabada es esta: en Go no preguntas "¿qué eres?", preguntas "¿qué sabes hacer?". Diseña contratos pequeños, deja que los tipos los cumplan por su cuenta y construye sistemas que crecen agregando piezas en lugar de modificando las existentes. En la siguiente unidad cambiarás de tercio por completo: le pondrás velocidad a tus programas con goroutines y canales, y descubrirás que la concurrencia en Go es tan natural como pedir tacos para toda la mesa al mismo tiempo.

UNIDAD 8: Goroutines y canales: concurrencia

Entra a una taquería de barrio a la hora de la comida. Hay una fila de veinte personas, el taquero tiene las manos llenas y sin embargo nadie espera más de cinco minutos. ¿Cómo le hace? No es que el taquero sea más rápido que tú calentando una tortilla en tu casa. El secreto es que no hace las cosas en fila, una tras otra, como las harías tú solo en tu cocina. Mientras la carne se asa en la plancha, él ya está picando cebolla con la otra mano, calentando tortillas en el comal de al lado y echándole ojo al trompo. No termina una tarea para empezar la siguiente: avanza varias a la vez, aprovechando los tiempos muertos de cada una. Cuando la carne necesita dos minutos en el fuego, esos dos minutos no se desperdician: se usan para otra cosa. Eso, mi estimado, es concurrencia. Y es exactamente lo que las goroutines y los canales te permiten hacer en tus programas.

Ahora imagina lo contrario: tú solo en tu cocina preparando cena para veinte personas. Pones la carne al fuego y te quedas parado mirándola dos minutos. Luego picas la cebolla. Luego calientas las tortillas, una por una. Cada paso espera a que termine el anterior. Eso es un programa secuencial tradicional: una sola línea de ejecución haciendo una cosa a la vez, desperdiciando todo el tiempo en que podría estar avanzando en otra cosa. Funciona, pero es lento y aburrido, como ver la carne asarse.

Aquí hay que aclarar desde el inicio una confusión muy común: concurrencia no es lo mismo que paralelismo. La concurrencia es la organización: estructurar tu programa para manejar varias tareas que avanzan de forma independiente, aunque se turnen para usar los recursos. El paralelismo es la ejecución física: hacer dos cosas literalmente al mismo tiempo, lo cual requiere más de un procesador o núcleo. El taquero con dos manos es concurrencia; dos taqueros trabajando lado a lado es paralelismo. Go te da concurrencia con las goroutines, y si tu máquina tiene varios núcleos, el runtime de Go reparte esas goroutines entre los núcleos disponibles y obtienes paralelismo gratis, sin que tengas que pedirlo. Tú diseñas el programa pensando en tareas independientes; Go se encarga de exprimir el hardware.

¿Por qué esto importa tanto? Porque el mundo real es concurrente. Un servidor web atiende a cientos de usuarios al mismo tiempo. Una aplicación lee archivos del disco, espera respuestas de internet y responde al usuario, todo a la vez. Si tu programa hace todo en secuencia, cada operación lenta (leer un archivo, esperar una respuesta de red, consultar una base de datos) congela todo lo demás. El usuario ve una aplicación trabada. Con concurrencia, mientras una tarea espera, las demás avanzan. El programa se siente vivo, rápido, profesional.

Otros lenguajes también tienen concurrencia, claro, pero casi siempre es complicada: hilos del sistema operativo que pesan mucho, que hay que crear y destruir con cuidado, y que se

comunican compartiendo memoria con candados por todos lados, lo cual es una fábrica de errores difícilísimos de encontrar. Go tomó otro camino, inspirado en un modelo elegante llamado CSP (Communicating Sequential Processes, procesos secuenciales que se comunican). La filosofía se resume en un proverbio famoso de la comunidad Go: "no te comuniques compartiendo memoria; comparte memoria comunicándote". En lugar de que dos tareas se peleen por la misma variable con candados, se mandan mensajes por canales. Cada tarea es dueña de sus datos y cuando necesita pasarle algo a otra, se lo envía por un canal, como quien pasa una nota por un tubo. Es más simple de razonar, más difícil de romper y, cuando algo sale mal, mucho más fácil de depurar.

Las dos piezas del rompecabezas son las goroutines y los canales. Una goroutine es una función ejecutándose de forma concurrente con el resto del programa. Crear una es ridículamente barato: escribes la palabra `go` delante de una llamada a función y listo, ya tienes una tarea corriendo en paralelo con tu código. No pesan casi nada (empiezan con unos pocos kilobytes de pila), así que puedes tener miles, incluso cientos de miles, sin que tu computadora sude. Compáralo con los hilos tradicionales, donde cada uno se come un megabyte o más y el sistema operativo se queja si creas demasiados. En Go, lanzar cien mil goroutines es un martes cualquiera.

Los canales son los tubos por donde las goroutines se hablan. Un canal transporta valores de un tipo específico: un canal de enteros solo lleva enteros, un canal de textos solo lleva textos. Una goroutine mete un valor por un extremo y otra lo saca por el otro. Y aquí está el detalle genial: si nadie está listo para recibir, el que envía se espera; si nadie ha enviado, el que quiere recibir se espera. Esa sincronización automática elimina una montaña de problemas. No necesitas decirle a nadie "avísame cuando termines": el canal lo hace solo.

En esta unidad vamos a construir tu intuición sobre la concurrencia desde cero. Empezaremos distinguiendo concurrencia de paralelismo con analogías que no se te van a olvidar. Luego crearemos nuestras primeras goroutines y descubriremos la primera trampa clásica: el programa principal no espera a las goroutines, así que hay que coordinarlas con `sync.WaitGroup`. Después construiremos canales, aprenderemos a enviar y recibir valores, y entenderemos la diferencia entre canales sin buffer (donde el apretón de manos es directo) y canales con buffer (donde hay una fila de espera). Veremos cómo cerrar canales, cómo recorrerlos con `for-range` y cómo detectar con la coma-ok si un canal ya se cerró. Luego conoceremos a `select`, que es como un semáforo que espera en varios canales a la vez, incluyendo timeouts con `time.After` y el modo no bloqueante con `default`. Después estudiaremos los patrones clásicos que todo programador Go debe conocer: el pipeline (línea de ensamble), el fan-out/fan-in (repartir trabajo y juntar resultados) y el worker pool (cuadrilla de trabajadores). También veremos los mutex para los casos donde compartir memoria con

candado es lo correcto, aprenderemos a decidir entre canales y mutex, y usaremos el detector de carreras de Go para cazar errores. Cerraremos con los errores comunes que hacen llorar a los principiantes (deadlocks, goroutines olvidadas, copiar un mutex) y cómo evitarlos.

Mira este adelanto para que veas lo simple que es empezar. Dos tareas corriendo a la vez, coordinadas con un WaitGroup:

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func cocinar(platillo string, wg *sync.WaitGroup) {
    defer wg.Done()
    fmt.Println("Cocinando", platillo, "...")
    time.Sleep(time.Second)
    fmt.Println(platillo, "listo!")
}

func main() {
    var wg sync.WaitGroup
    wg.Add(2)
    go cocinar("tacos", &wg)
    go cocinar("quesadillas", &wg)
    wg.Wait()
    fmt.Println("Todo servido, buen provecho.")
}
```

Con la palabra go delante de cada llamada, las dos cocciones avanzan al mismo tiempo en lugar de una tras otra. El WaitGroup es el mesero que no sirve la mesa hasta que la cocina termina los dos platillos: Add cuenta cuántas tareas hay, Done marca cada tarea terminada y Wait se queda esperando hasta que todas terminan. Sin el WaitGroup, el programa principal terminaría antes y las goroutines morirían a la mitad, como un restaurante que cierra la cocina con los tacos a medio hacer. No te preocupes si todavía no entiendes cada línea: al final de esta unidad vas a leer ese programa como lees el menú de tu taquería favorita.

Una advertencia honesta antes de empezar: la concurrencia es poderosa pero tiene sus trampas, y esta unidad no te las va a esconder. Vas a ver programas que se quedan congelados para siempre (deadlocks), goroutines que se quedan esperando un mensaje que nunca llega (fugas), y carreras de datos donde dos goroutines pisan la misma variable. Todo

eso tiene solución y la vamos a ver. La meta no es que le tengas miedo a la concurrencia, sino que la uses con respeto y con las herramientas correctas. Como el cuchillo del taquero: en manos entrenadas es velocidad pura; en manos descuidadas, un accidente. Empecemos el entrenamiento.

Explicación detallada

Concurrencia no es paralelismo

Vamos a fijar esta distinción porque te va a perseguir toda tu carrera. La concurrencia es una propiedad del diseño de tu programa: significa que está estructurado como un conjunto de tareas independientes que pueden avanzar sin esperarse entre sí. El paralelismo es una propiedad de la ejecución: significa que dos tareas corren literalmente al mismo tiempo en hardware distinto. Todo programa paralelo es concurrente, pero no todo programa concurrente es paralelo. Un solo taquero alternando entre la plancha y el comal es concurrencia sin paralelismo; dos taqueros, uno en cada plancha, es paralelismo.

¿Por qué importa la distinción? Porque la concurrencia te sirve incluso con un solo núcleo de procesador. Si tu programa espera la respuesta de un servidor (operación lenta), con un diseño concurrente puede atender a otros usuarios mientras espera. No necesitas más hardware, necesitas mejor organización. Y cuando sí hay varios núcleos, el runtime de Go reparte tus goroutines entre ellos automáticamente y obtienes paralelismo sin cambiar tu código. Tú piensas en tareas independientes; Go se encarga del hardware. Esa separación de preocupaciones es una de las razones por las que la concurrencia en Go se siente tan natural.

Goroutines: concurrencia con una palabra

Una goroutine es una unidad de ejecución ligera manejada por el runtime de Go. No es un hilo del sistema operativo: es mucho más barata. Un hilo tradicional ocupa alrededor de un megabyte de memoria solo por existir y crear miles de ellos pone de rodillas al sistema. Una goroutine empieza con unos 2 kilobytes de pila que crecen y se encogen según lo necesite. Puedes tener cientos de miles de goroutines en una laptop normal sin problema. Por eso en Go la concurrencia no es un lujo reservado para casos especiales: es la forma normal de estructurar programas.

Crear una goroutine es tan simple que parece truco: escribes la palabra go delante de una llamada a función.

```
go hacerAlgo()
go hacerOtraCosa(42, "hola")
go func() {
    fmt.Println("goroutine anonima")
}()
```

En el momento en que Go ejecuta esa línea, la función arranca en una goroutine nueva y la ejecución continúa inmediatamente con la siguiente línea, sin esperar a que la función termine. Las goroutines también pueden ser funciones anónimas, lo cual es muy común para tareas cortas. Y ojo con un detalle: los argumentos de la función se evalúan en el momento de lanzar la goroutine, no cuando esta empieza a correr. Si escribes `go f(x)`, el valor de `x` se captura ahí mismo.

El programa no espera a tus goroutines

Aquí está la primera trampa que muerde a todo principiante. Cuando la función `main` termina, el programa termina, sin importar cuántas goroutines sigan trabajando. No hay espera automática, no hay aviso, no hay cortesía: `main` regresa y todo se apaga de golpe, como si cerraran el restaurante con los clientes adentro.

```
package main

import (
    "fmt"
    "time"
)

func saludo() {
    time.Sleep(time.Second)
    fmt.Println("Hola desde la goroutine")
}

func main() {
    go saludo()
    fmt.Println("Adios desde main")
}
```

Si corres ese programa, casi seguro solo verás "Adios desde main". La goroutine ni alcanza a despertar de su segundo de sueño antes de que main termine y el programa muera. Esto no es un bug del lenguaje: es el diseño. El programa principal es el dueño del local; cuando él se va, se apaga la luz. Por eso necesitas mecanismos de coordinación para decirle a main "espérate, todavía hay trabajo en la cocina". El más directo es `sync.WaitGroup`.

`sync.WaitGroup`: Add, Done y Wait

Un `WaitGroup` es un contador de tareas pendientes. Funciona con tres operaciones. `Add(n)` suma `n` al contador, normalmente con cuántas goroutines vas a lanzar. `Done()` resta uno, y cada goroutine lo llama cuando termina (casi siempre con `defer` para que se ejecute aunque la función salga por un error o un `return` temprano). `Wait()` bloquea la goroutine que lo llama hasta que el contador llega a cero.

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func trabajador(id int, wg *sync.WaitGroup) {
    defer wg.Done()
    fmt.Println("Trabajador", id, "empezando")
    time.Sleep(time.Second)
    fmt.Println("Trabajador", id, "terminado")
}

func main() {
    var wg sync.WaitGroup
    for i := 1; i <= 3; i++ {
        wg.Add(1)
        go trabajador(i, &wg)
    }
    wg.Wait()
    fmt.Println("Todos terminaron")
}
```

Tres detalles finos que te van a ahorrar horas de depuración. Primero: el `WaitGroup` se pasa por puntero (`&wg`), porque si lo pasaras por valor cada goroutine tendría su propia copia del contador y el original nunca llegaría a cero. De hecho, copiar un `WaitGroup` después de usarlo

está prohibido por la misma razón que copiar un Mutex, como veremos. Segundo: llama Add antes de lanzar la goroutine, no adentro, para evitar una carrera donde Wait vea el contador en cero antes de que la goroutine sume. Tercero: usa defer wg.Done() como primera línea de la goroutine para garantizar que el contador baje pase lo que pase.

Canales: los tubos de la concurrencia

Las goroutines necesitan hablarse, y para eso están los canales. Un canal es un conducto tipado por el que viajan valores de un tipo específico. Lo creas con make indicando el tipo de dato que va a transportar:

```
canal := make(chan int)
```

Enviar un valor se escribe con la flecha apuntando al canal: canal <- 42. Recibir se escribe con la flecha saliendo del canal: valor := <-canal. La flecha siempre apunta en la dirección en que viaja el dato, lo cual hace el código muy legible una vez que te acostumbras.

La propiedad fundamental de un canal sin buffer es que el envío y la recepción se sincronizan: el que envía se bloquea hasta que alguien recibe, y el que recibe se bloquea hasta que alguien envía. Es un apretón de manos: nadie avanza hasta que los dos están listos. Esto elimina por diseño una clase entera de errores de sincronización, porque no necesitas mecanismos adicionales para decir "ya terminé, aquí está tu dato". El canal lo garantiza.

```
package main

import "fmt"

func main() {
    canal := make(chan string)
    go func() {
        canal <- "hola desde la goroutine"
    }()
    mensaje := <-canal
    fmt.Println(mensaje)
}
```

La goroutine envía el mensaje y main lo recibe. Si main intentara recibir sin que nadie envíe, se quedaría esperando para siempre (y el runtime detectaría el deadlock, como veremos). Si la goroutine intentara enviar sin que nadie reciba, también se quedaría esperando. Esa espera mutua es la sincronización automática del canal.

Canales con buffer y sin buffer

Puedes darle a un canal una capacidad, un buffer, indicando cuántos valores puede guardar en fila sin que nadie los reciba todavía:

```
sinBuffer := make(chan int)    // capacidad 0: apretón de manos directo
conBuffer := make(chan int, 3) // capacidad 3: fila de hasta 3 valores
```

La analogía perfecta es la barra de la taquería. Un canal sin buffer es como entregarle el taco directamente en la mano al cliente: el taquero no puede seguir hasta que el cliente lo agarra, y el cliente no puede comer hasta que el taquero se lo da. Un canal con buffer es como la charola donde el taquero deja los tacos ya hechos: puede dejar hasta tres tacos y seguir cocinando sin esperar a nadie; los clientes los toman cuando llegan. Solo cuando la charola se llena el taquero tiene que esperar a que alguien tome uno, y solo cuando está vacía el cliente tiene que esperar a que salga el siguiente.

El buffer desacopla en el tiempo al productor y al consumidor: el productor puede adelantarse hasta llenar el buffer. Esto es útil para suavizar ritmos distintos, pero no abuses de los buffers grandes "por si acaso": un buffer enorme puede esconder problemas de diseño y hacer más difícil razonar sobre el programa. La regla práctica es empezar sin buffer (la sincronización más simple y predecible) y agregar buffer solo cuando tengas una razón concreta, como evitar que el productor se bloquee por picos temporales del consumidor.

Cerrar canales y el for-range

Cuando ya no vas a enviar más valores por un canal, lo cierras con `close(canal)`. Cerrar no destruye el canal: los valores que ya están en el buffer se pueden seguir recibiendo, pero ya no se puede enviar nada más (enviar a un canal cerrado provoca pánico). La convención es clara: quien envía cierra, quien recibe nunca cierra. El que produce los datos sabe cuándo se acabaron; el consumidor no.

El cierre es la señal de "ya no viene nada más", y se combina de maravilla con `for-range`, que recibe valores del canal hasta que se cierra:

```

package main

import "fmt"

func main() {
    canal := make(chan int)
    go func() {
        for i := 1; i <= 5; i++ {
            canal <- i
        }
        close(canal)
    }()
    for n := range canal {
        fmt.Println("Recibi:", n)
    }
    fmt.Println("Canal cerrado, terminamos")
}

```

El for-range se queda recibiendo hasta que el canal se cierra, y en ese momento sale del ciclo limpiamente. Sin el close, el for-range se quedaría esperando un sexto valor eternamente. Este patrón (goroutine que envía N valores, cierra, y main que recorre con for-range) es uno de los más comunes en Go y lo verás en pipelines y workers.

Detectar un canal cerrado: la coma-ok

A veces necesitas saber si el valor que recibiste vino de un canal abierto o si el canal ya se cerró. Para eso la recepción tiene una segunda forma que regresa dos valores, igual que la coma-ok de las interfaces:

```

valor, ok := <-canal
if ok {
    fmt.Println("Recibi:", valor)
} else {
    fmt.Println("El canal esta cerrado")
}

```


Cuando el canal está cerrado y vacío, la recepción regresa inmediatamente el valor cero del tipo y ok en falso. Sin la coma-ok, recibirías el valor cero sin saber si es un dato real o el eco del cierre, lo cual puede confundir tu lógica (¿el cero es un dato o es que se acabó?). En la práctica, el for-range maneja el cierre automáticamente y rara vez necesitas la coma-ok manual, pero es indispensable conocerla para los casos donde recibes valores sueltos y el cierre cambia lo que debes hacer.

select: el semáforo de los canales

Una goroutine frecuentemente necesita esperar en varios canales a la vez: "avanza con el primero que tenga datos". Para eso existe select, que se parece a un switch pero cada caso es una operación de canal. El select se bloquea hasta que uno de los casos puede avanzar, y si varios están listos al mismo tiempo, elige uno al azar (así evita favorecer siempre al mismo).

```
select {
case msg := <-canalA:
    fmt.Println("Llego por A:", msg)
case msg := <-canalB:
    fmt.Println("Llego por B:", msg)
}
```

Piensa en el select como el taquero atendiendo dos filas de clientes: atiende a quien esté listo primero, sin preferencia fija. Es la herramienta que permite construir servidores que atienden múltiples conexiones, programas que combinan resultados de varias fuentes y sistemas que reaccionan a eventos de distinto origen.

Timeouts con time.After

Una de las aplicaciones más valiosas del select es el timeout: esperar un resultado pero no eternamente. La función `time.After(duracion)` regresa un canal que recibe un valor cuando pasa el tiempo indicado. Lo combinas en el select con tu operación normal:

```
select {
case resultado := <-canal:
    fmt.Println("Llego a tiempo:", resultado)
case <-time.After(2 * time.Second):
    fmt.Println("Se acabo el tiempo, seguimos sin el dato")
}
```

Si el dato llega antes de dos segundos, se toma el primer caso; si no, a los dos segundos se dispara el segundo y tu programa sigue su camino en lugar de quedarse colgado. Esto es oro en sistemas reales: ningún servicio externo debería poder congelar tu programa para siempre porque tardó en responder. Todo lo que espera algo de afuera (red, disco, otro servicio) merece un timeout.

default: el select no bloqueante

Si agregas un caso default al select, este nunca se bloquea: si ningún canal está listo en ese instante, se ejecuta el default inmediatamente.

```
select {
  case msg := <-canal:
    fmt.Println("Había dato:", msg)
  default:
    fmt.Println("Nada listo, hago otra cosa")
}
```

Esto sirve para sondear sin esperar ("¿hay algo? si no, sigo"), para implementar reintentos, o para evitar bloqueos en situaciones delicadas. Úsalo con medida: un select con default dentro de un ciclo apretado sin pausas se convierte en un ciclo que gira consumiendo procesador a lo loco (busy loop). Si necesitas sondear repetidamente, dale un respiro al ciclo con un pequeño sleep o un time.After.

Patrones: pipeline

Un pipeline es una línea de ensamble: varias etapas conectadas por canales, donde cada etapa recibe valores, los transforma y los pasa a la siguiente. Es el patrón más natural con canales y aparece en procesamiento de datos, imágenes, texto y más.

```
generar -> duplicar -> imprimir
```

Cada etapa es una goroutine con un canal de entrada y uno de salida. La belleza del pipeline es que las etapas trabajan en paralelo sobre datos distintos: mientras la etapa 3 imprime el primer valor, la etapa 2 ya está duplicando el segundo y la etapa 1 generando el tercero. Es la taquería otra vez: mientras un taco se sirve, otro se arma y otro se cocina. Y cada etapa es independiente y reemplazable: puedes agregar una etapa de filtrado en medio sin tocar las demás.

Patrones: fan-out y fan-in

A veces una etapa del pipeline es más lenta que las demás y se vuelve el cuello de botella. La solución es el fan-out: en lugar de una goroutine haciendo el trabajo pesado, lanzas varias goroutines idénticas leyendo del mismo canal de entrada. El trabajo se reparte solo, porque cada worker toma el siguiente valor disponible. Luego el fan-in junta los resultados: varias goroutines escriben a un mismo canal de salida, y un multiplexor (o simplemente el consumidor leyendo de ese canal) los reúne. Fan-out reparte, fan-in reúne. Es como pasar de un taquero a tres taqueros en la plancha durante la hora pico: la fila avanza al triple de velocidad.

Patrones: worker pool

El worker pool es la versión disciplinada del fan-out: un número fijo de trabajadores (workers) que toman tareas de un canal compartido. Lo usas cuando quieres limitar cuántas cosas pasan al mismo tiempo: por ejemplo, no quieres abrir mil conexiones a una base de datos, solo diez. Creas diez workers, les das un canal de tareas, y cada worker procesa una tarea a la vez, tomando la siguiente cuando termina.

```
tareas := make(chan Tarea)
for i := 0; i < 10; i++ {
    go worker(tareas) // 10 workers fijos
}
// ... enviar tareas al canal ...
```

El pool te da control: ni una tarea a la vez (lento) ni un millón (colapso), sino exactamente las que tu sistema aguanta. Es el patrón estándar para servidores, procesadores de archivos y cualquier trabajo masivo con recursos limitados.

Mutex: cuando los canales no bastan

El proverbio dice "comparte memoria comunicándote", pero hay casos donde lo natural es compartir una variable entre goroutines: un contador, un caché, una configuración. Para eso está sync.Mutex, un candado de exclusión mutua. Solo una goroutine a la vez puede tener el candado; las demás esperan su turno.

```
var mu sync.Mutex
var contador int

mu.Lock()
contador++
mu.Unlock()
```

Todo lo que está entre Lock y Unlock es una sección crítica: se ejecuta sin interrupciones de otras goroutines que respeten el mismo candado. La trampa clásica es olvidar el Unlock (usa `defer mu.Unlock()` justo después del Lock para no olvidarlo nunca) o copiar el Mutex (pásalo siempre por puntero, igual que el `WaitGroup`). Go también ofrece `sync.RWMutex`, que permite muchas lecturas simultáneas pero escritura exclusiva, útil cuando lees mucho y escribes poco.

Canales o mutex: cómo decidir

La pregunta del millón. Usa canales cuando las goroutines se pasan la batuta: una produce datos y otra los consume, hay un flujo de información, quieres pipelines o repartir trabajo. Usa mutex cuando las goroutines comparten un estado: un contador, un mapa, un caché que varias leen y escriben. Una regla mental simple: si puedes describir el problema como "tareas que se mandan cosas", canales; si lo describes como "varias tareas tocando lo mismo", mutex. Y no son enemigos: los programas reales mezclan ambos, por ejemplo un worker pool (canales) donde los workers actualizan estadísticas compartidas (mutex). Lo importante es que cada dato tenga un dueño claro y una sola forma de acceso.

El detector de carreras: `go run -race`

Una carrera de datos (data race) ocurre cuando dos goroutines acceden a la misma variable al mismo tiempo y al menos una la modifica, sin sincronización. El resultado es impredecible: a veces funciona, a veces da valores corruptos, a veces truena en producción un martes a las 3 de la mañana. Lo traicionero es que las carreras no siempre se manifiestan en tus pruebas. Por eso Go incluye un detector de carreras integrado: corres tu programa con `go run -race` o `go test -race` y el runtime vigila todos los accesos a memoria, reportando con detalle dónde ocurrió la carrera, qué goroutines participaron y en qué líneas. No es perfecto (solo detecta las carreras que efectivamente ocurren en esa ejecución), pero es una red de seguridad extraordinaria. Acostúmbrate a correr tus pruebas con `-race` siempre que uses concurrencia. Es gratis y te salva de los bugs más feos.

Canales direccionales: solo envío y solo recepción

Hasta ahora todos nuestros canales son bidireccionales: sirven para enviar y recibir. Pero Go permite declarar canales de una sola dirección, y hacerlo es una excelente práctica. Un `chan<-int` solo permite enviar; un `<-chan int` solo permite recibir. La flecha pegada a `chan` indica la dirección permitida.

```
func producir(salida chan<- int) {  
    salida <- 42 // permitido: es de solo envio  
}  
  
func consumir(entrada <-chan int) {  
    valor := <-entrada // permitido: es de solo recepcion  
}
```

¿Por qué molestarse? Por dos razones. Primera, documentación viva: la firma de la función dice exactamente qué va a hacer con el canal, y quien la lea no tiene que adivinar. Segunda, seguridad del compilador: si por accidente intentas enviar por un canal de solo recepción, el código ni siquiera compila. Es el mismo espíritu de las interfaces pequeñas: restringe cada pieza a lo mínimo que necesita y el compilador se convierte en tu vigilante. Un detalle técnico: el canal se crea siempre bidireccional con `make`, y se "convierte" a direccional al pasarlo como parámetro o asignarlo a una variable direccional. La conversión solo puede quitar permisos, nunca agregarlos.

Cancelación: el canal `done`

Las goroutines que trabajan en ciclos largos necesitan una forma de detenerse cuando ya no las necesitas. El patrón clásico es un canal de señal llamado `done`: la goroutine lo revisa en cada vuelta con `select`, y cuando alguien lo cierra, la goroutine termina limpiamente.

```

done := make(chan struct{})

go func() {
    for {
        select {
        case <-done:
            fmt.Println("Recibi la senal, terminando")
            return
        default:
            // ... hacer trabajo ...
        }
    }
}()

// ... mas tarde, cuando ya no la necesitas:
close(done)

```

El tipo `struct{}` se usa porque no ocupa memoria: lo único que importa es la señal del cierre, no ningún valor. Cerrar el canal despierta a todos los que estén esperando en él, así que una sola señal puede detener a muchas goroutines a la vez. En código profesional moderno este patrón evolucionó al paquete `context`, con `context.WithCancel`, que hace lo mismo pero propagando la cancelación por llamadas encadenadas y con deadlines incluidos. Cuando domines el canal `done`, el paquete `context` te parecerá una extensión natural.

time.Ticker: el metrónomo

Ya conoces `time.After` para un solo disparo de tiempo. Su primo `time.Ticker` repite el disparo a intervalos regulares: es un metrónomo que manda una señal por su canal cada cierto tiempo. Sirve para trabajo periódico: revisar una bandeja cada minuto, mandar un latido (heartbeat) cada cinco segundos, refrescar un caché.

```

ticker := time.NewTicker(500 * time.Millisecond)
defer ticker.Stop()

for i := 0; i < 5; i++ {
    <-ticker.C
    fmt.Println("Tick", i+1)
}

```

Dos advertencias. Primera: siempre detén el ticker con `Stop` cuando termines, o su goroutine interna seguirá viva mandando señales a un canal que nadie lee (una fuga de recursos de

manual). El `defer ticker.Stop()` justo después de crearlo es la forma segura. Segunda: si tu trabajo tarda más que el intervalo, los ticks se acumulan o se pierden según el caso; el ticker marca el ritmo, no garantiza que alcances a bailarlo. Para ritmos críticos, mide y ajusta.

Patrones: semáforo con canal con buffer

Hay situaciones donde quieres lanzar muchas goroutines pero limitar cuántas trabajan al mismo tiempo, sin la estructura formal de un worker pool. Para eso sirve el semáforo implementado con un canal con buffer: el buffer representa los permisos disponibles.

```
sem := make(chan struct{}, 3) // maximo 3 trabajos simultaneos

sem <- struct{}{}           // tomar un permiso (se bloquea si no hay)
defer func() { <-sem }()    // devolverlo al terminar
```

Cada goroutine toma un permiso antes de trabajar y lo devuelve con `defer` al terminar. Si las tres plazas están ocupadas, la cuarta goroutine se queda esperando en el envío hasta que alguien libere. Es el antro con capacidad limitada: el cadenero deja pasar hasta tres y los demás esperan en la fila. La diferencia con el worker pool es de estilo: el pool tiene workers fijos que toman tareas de un canal, mientras que el semáforo deja que lances goroutines libremente pero acota cuántas entran a la zona crítica a la vez. Úsalo para limitar conexiones a una base de datos, descargas simultáneas o llamadas a una API con límite de peticiones. Es simple, expresivo y no necesita ninguna estructura adicional.

El orden no está garantizado (y cómo vivir con eso)

Hay una verdad incómoda de la concurrencia que debes interiorizar: el planificador de goroutines decide quién corre y cuándo, y no te promete ningún orden. Si lanzas tres goroutines que imprimen "uno", "dos" y "tres", la salida puede ser "dos, uno, tres" hoy y "tres, tres..." bueno, cada una imprime una vez, pero en cualquier orden, y mañana puede ser otro. El error clásico es escribir código que depende de un orden observado en pruebas: "en mi máquina siempre sale primero el worker 1". En producción, con otra carga y otro procesador, sale otro. Nunca, jamás, diseñes lógica que dependa del orden de ejecución de goroutines independientes.

Cuando el orden sí importa, lo impones tú con canales. Tres técnicas cubren casi todo. Primera: encadenar canales, donde cada goroutine espera el mensaje de la anterior (como la carrera de relevos del ejercicio 11). Segunda: etiquetar cada tarea con un número de secuencia y reordenar al recolectar. Tercera: usar una sola goroutine recolectora que recibe resultados y los procesa en el orden que ella decide. La moraleja es simple: la concurrencia te

da velocidad desordenada por defecto; el orden es un requisito que se diseña explícitamente, no algo que se asume.

Cuándo la concurrencia no ayuda

Después de tanta maravilla, un baño de realidad: la concurrencia no siempre hace tu programa más rápido, y a veces lo hace más lento. Cada goroutine y cada operación de canal tiene un costo, pequeño pero real. Si la tarea dura microsegundos, el costo de coordinarla puede superar el beneficio de paralelizarla: es como contratar un mensajero en moto para llevar una carta al vecino de enfrente. Además, si tu programa ya satura todos los núcleos con trabajo de cómputo, agregar más goroutines no crea más procesadores; solo agrega cambios de contexto.

La concurrencia brilla con operaciones que esperan: red, disco, bases de datos, temporizadores. Ahí el tiempo muerto es enorme y aprovecharlo es ganancia pura. Brilla menos con cómputo puro ya saturado o con tareas tan pequeñas que la coordinación cuesta más que el trabajo. Y hay un caso donde es contraproducente por diseño: cuando el orden estricto obliga a serializar todo de todos modos, terminas con la complejidad de la concurrencia y la velocidad de lo secuencial. La regla profesional es medir: si sospechas que la concurrencia ayuda, compruébalo con una prueba de rendimiento (benchmark), no con fe. Go hace que lanzar goroutines sea barato, pero "barato" no es "gratis", y el buen ingeniero sabe cuándo no usar su herramienta favorita.

Errores comunes

El deadlock es el error más famoso: dos o más goroutines esperándose mutuamente para siempre, o una goroutine esperando en un canal que nadie va a alimentar. El programa se congela y no avanza. Go te ayuda un poco: si todas las goroutines están bloqueadas, el runtime lo detecta y truena con "fatal error: all goroutines are asleep - deadlock!". Pero si al menos una goroutine sigue viva (por ejemplo main esperando un WaitGroup que nunca se completa), el programa se queda colgado en silencio. La prevención: asegúrate de que cada envío tenga su recepción, cada WaitGroup tenga sus Done, y cada canal del que lees en for-range se cierre en algún momento.

La fuga de goroutines (goroutine leak) es más silenciosa: una goroutine que se queda bloqueada esperando algo que nunca llega, viviendo para siempre y consumiendo memoria. Miles de fugas pueden tumbar un servidor por agotamiento de memoria. La prevención: toda goroutine debe tener una forma de terminar, ya sea porque su trabajo se acaba, porque se cierra un canal, o porque recibe una señal de cancelación (con la biblioteca context, que conocerás más adelante).

Copiar un Mutex o un WaitGroup es el tercer clásico: al copiarlos se duplica su estado interno y el candado o el contador dejan de funcionar como esperas. La regla es simple y absoluta: los tipos de sincronización siempre se pasan por puntero, nunca por valor. El comando `go vet` incluso te avisa si los copias por accidente. Úsalo.

Ejemplos paso a paso

Ejemplo 1: Lanzar goroutines y esperarlas con WaitGroup

El patrón base de toda la concurrencia en Go: varias tareas independientes, coordinadas para que `main` no termine antes de tiempo.

```

package main

import (
    "fmt"
    "sync"
    "time"
)

func descargar(archivo string, wg *sync.WaitGroup) {
    defer wg.Done()
    fmt.Println("Descargando", archivo, "...")
    time.Sleep(time.Second)
    fmt.Println(archivo, "descargado")
}

func main() {
    var wg sync.WaitGroup
    archivos := []string{"foto.jpg", "video.mp4", "doc.pdf"}

    for _, a := range archivos {
        wg.Add(1)
        go descargar(a, &wg)
    }

    wg.Wait()
    fmt.Println("Todas las descargas terminaron")
}

```

Tres observaciones. Primera: `wg.Add(1)` va antes del `go`, nunca adentro de la goroutine, para que el contador esté correcto cuando `Wait` lo revise. Segunda: el `WaitGroup` viaja por puntero; si lo pasaras por valor, cada goroutine contaría en su propia copia y `Wait` esperaría eternamente. Tercera: `defer wg.Done()` garantiza que el contador baje aunque la función termine antes de tiempo. Si corres el programa varias veces, verás que el orden de los mensajes cambia: las goroutines no garantizan orden, y eso es normal.

Ejemplo 2: Productor y consumidor con un canal

Una goroutine produce números, otra los consume. El canal los conecta y los sincroniza sin ningún candado.

```

package main

import "fmt"

func productor(canal chan int) {
    for i := 1; i <= 5; i++ {
        fmt.Println("Produciendo", i)
        canal <- i
    }
    close(canal)
}

func main() {
    canal := make(chan int)
    go productor(canal)

    for n := range canal {
        fmt.Println("Consumiendo", n)
    }
    fmt.Println("Se acabo la produccion")
}

```

El productor envía cinco valores y cierra el canal; main los recibe con for-range hasta el cierre. Como el canal no tiene buffer, cada envío espera a su recepción: producción y consumo avanzan al mismo ritmo, de la mano. Si el productor fuera más rápido, se frenaría solo; si el consumidor fuera más rápido, esperaría solo. Esa es la sincronización automática del canal sin buffer.

Ejemplo 3: Un pipeline de tres etapas

Línea de ensamble: generar números, duplicarlos y luego imprimirlos. Cada etapa es una goroutine y los canales son las bandas transportadoras.

```

package main

import "fmt"

// Etapa 1: genera numeros del 1 al 5 y cierra su salida.
func generar(salida chan int) {
    defer close(salida)
    for i := 1; i <= 5; i++ {
        salida <- i
    }
}

// Etapa 2: lee numeros, los duplica y los pasa adelante.
func duplicar(entrada, salida chan int) {
    defer close(salida)
    for n := range entrada {
        salida <- n * 2
    }
}

// Etapa 3: imprime lo que llega hasta que se cierra el canal.
func imprimir(entrada chan int) {
    for n := range entrada {
        fmt.Println("Resultado:", n)
    }
}

func main() {
    etapa1 := make(chan int)
    etapa2 := make(chan int)

    go generar(etapa1)
    go duplicar(etapa1, etapa2)
    imprimir(etapa2)

    fmt.Println("Pipeline terminado")
}

```

Cada etapa cierra su canal de salida cuando termina, lo que permite que la siguiente etapa salga de su for-range limpiamente. El cierre se propaga como una ola por el pipeline. Y las etapas trabajan en paralelo sobre datos distintos: mientras imprimir muestra el 2, duplicar ya está procesando el 3 y generar el 4. Si quisieras agregar una etapa de filtrado entre duplicar e imprimir, solo crearías la función y un canal más, sin tocar las etapas existentes.

Ejemplo 4: Worker pool con número fijo de trabajadores

Diez tareas pesadas, pero solo tres workers para no saturar el sistema. Las tareas esperan en un canal y cada worker toma la siguiente cuando se desocupa.

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func worker(id int, tareas <-chan int, wg *sync.WaitGroup) {
    defer wg.Done()
    for t := range tareas {
        fmt.Printf("Worker %d procesando tarea %d\n", id, t)
        time.Sleep(500 * time.Millisecond)
    }
    fmt.Printf("Worker %d sin mas tareas\n", id)
}

func main() {
    const numWorkers = 3
    const numTareas = 10

    tareas := make(chan int)
    var wg sync.WaitGroup

    for i := 1; i <= numWorkers; i++ {
        wg.Add(1)
        go worker(i, tareas, &wg)
    }

    for t := 1; t <= numTareas; t++ {
        tareas <- t
    }
    close(tareas)

    wg.Wait()
    fmt.Println("Todas las tareas completadas")
}
```

Nota el detalle del parámetro tareas <-chan int: es un canal de solo recepción. El worker no necesita enviar por él, y declararlo así deja clara la intención además de que el compilador te impide usarlo mal. Main envía las diez tareas, cierra el canal, y los workers salen de su for-range cuando se acaban. Con tres workers y tareas de medio segundo, las diez tareas toman unos dos segundos en lugar de cinco: control total sobre el paralelismo sin complicarte la vida.

Ejemplo 5: Select con timeout

Esperar una respuesta de un servicio lento, pero rendirse con dignidad si tarda demasiado.

```
package main

import (
    "fmt"
    "time"
)

func servicioLento(respuesta chan string) {
    time.Sleep(3 * time.Second)
    respuesta <- "datos del servidor"
}

func main() {
    respuesta := make(chan string)
    go servicioLento(respuesta)

    select {
    case datos := <-respuesta:
        fmt.Println("Llego a tiempo:", datos)
    case <-time.After(2 * time.Second):
        fmt.Println("Timeout: el servicio tardo demasiado")
    }
}
```

El servicio tarda tres segundos pero el timeout es de dos, así que verás el mensaje de timeout. Cambia el sleep a un segundo y verás la respuesta exitosa. Este patrón es obligatorio en código real: cada vez que esperes algo de la red, de un disco o de otro sistema, ponle un timeout. Un programa que puede quedarse esperando para siempre es un programa que algún día se va a quedar esperando para siempre.

Ejemplo 6: Mutex protegiendo un contador

Cien goroutines incrementando el mismo contador. Sin protección, el resultado sería impredecible; con Mutex, siempre es 100.

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    var mu sync.Mutex
    contador := 0
    var wg sync.WaitGroup

    for i := 0; i < 100; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            mu.Lock()
            contador++
            mu.Unlock()
        }()
    }

    wg.Wait()
    fmt.Println("Contador final:", contador)
}
```

La operación `contador++` parece atómica, pero no lo es: leer, sumar y escribir son pasos separados, y dos goroutines pueden leerse el mismo valor y pisarse. El Mutex garantiza que solo una goroutine a la vez ejecute la sección crítica. Prueba quitar el `Lock` y el `Unlock` y corre con `go run -race`: el detector te mostrará la carrera con lujo de detalle. Luego vuelve a ponerlos y verifica que el reporte desaparece. Esa es la forma profesional de validar código concurrente: no basta con que "parezca funcionar", tiene que pasar el `-race`.

Ejemplo 7: Fan-in, juntando resultados de varios trabajos

Dos productores generan datos a ritmos distintos y una función fan-in los reúne en un solo canal de salida. El consumidor no sabe ni le importa de dónde vino cada valor.

```

package main

import (
    "fmt"
    "sync"
    "time"
)

func generarLetras(salida chan<- string) {
    defer close(salida)
    for _, l := range []string{"a", "b", "c"} {
        salida <- l
        time.Sleep(300 * time.Millisecond)
    }
}

func generarNumeros(salida chan<- string) {
    defer close(salida)
    for _, n := range []string{"1", "2", "3"} {
        salida <- n
        time.Sleep(500 * time.Millisecond)
    }
}

// fanIn junta varios canales de entrada en uno solo de salida.
func fanIn(entradas ...chan string) chan string {
    salida := make(chan string)
    var wg sync.WaitGroup

    for _, e := range entradas {
        wg.Add(1)
        go func(c chan string) {
            defer wg.Done()
            for v := range c {
                salida <- v
            }
        }(e)
    }

    go func() {
        wg.Wait()
        close(salida)
    }()

    return salida
}

```



```

}

func main() {
    letras := make(chan string)
    numeros := make(chan string)

    go generarLetras(letras)
    go generarNumeros(numeros)

    for v := range fanIn(letras, numeros) {
        fmt.Println("Llego:", v)
    }
    fmt.Println("Todos los productores terminaron")
}

```

El truco está en la goroutine que espera al WaitGroup y luego cierra la salida: así el for-range del main termina limpio cuando ambos productores acaban. Los valores llegan intercalados según el ritmo de cada productor (las letras, más rápidas, tienden a llegar primero). Este es el esqueleto de sistemas que combinan resultados de múltiples fuentes: réplicas de servidores, sensores, archivos.

Ejemplo 8: Trabajador con cancelación elegante

Un trabajador procesa tareas en ciclo, pero revisa en cada vuelta si ya le pidieron detenerse mediante el canal done.

```

package main

import (
    "fmt"
    "time"
)

func trabajador(tareas <-chan int, done <-chan struct{}) {
    for {
        select {
        case t, ok := <-tareas:
            if !ok {
                fmt.Println("No hay mas tareas, terminando")
                return
            }
            fmt.Println("Procesando tarea", t)
            time.Sleep(400 * time.Millisecond)
        case <-done:
            fmt.Println("Cancelacion recibida, terminando")
        }
    }
}

```

```

        return
    }
}

func main() {
    tareas := make(chan int)
    done := make(chan struct{})

    go trabajador(tareas, done)

    go func() {
        defer close(tareas)
        for i := 1; i <= 10; i++ {
            select {
            case tareas <- i:
                // tarea entregada
            case <-done:
                fmt.Println("Productor: cancelacion recibida, dejo de enviar")
                return
            }
        }
    }()

    // Dejamos trabajar 1 segundo y luego cancelamos.
    time.Sleep(time.Second)
    close(done)
    time.Sleep(500 * time.Millisecond)
    fmt.Println("Main termina")
}

```

El trabajador alcanza a procesar un par de tareas antes de recibir la cancelación y termina de forma ordenada, sin dejar trabajo a medias de forma violenta. En sistemas reales, entre la señal y el return guardarías el progreso o cerrarías recursos. La lección: toda goroutine de vida larga merece una puerta de salida, y el canal done es la puerta más simple.

Ejemplo 9: Ticker como latido del sistema

Un monitor que reporta "sistema vivo" cada medio segundo, cinco veces, y luego se detiene limpiamente.

```

package main

import (
    "fmt"
    "time"
)

func main() {
    ticker := time.NewTicker(500 * time.Millisecond)
    defer ticker.Stop() // vital: libera los recursos del ticker

    contador := 0
    for range ticker.C {
        contador++
        fmt.Println("Latido", contador, "- sistema vivo")
        if contador >= 5 {
            break
        }
    }
    fmt.Println("Monitor detenido")
}

```

El for range sobre ticker.C recibe un valor en cada tick. El break sale del ciclo y el defer detiene el ticker. Sin el Stop, la goroutine interna del ticker seguiría existiendo hasta que el programa termine. Parece un detalle menor, pero en servidores que crean tickers por cada conexión, olvidar el Stop es una fuga de memoria garantizada.

Ejemplo 10: Semáforo que limita descargas simultáneas

Ocho descargas, pero nunca más de tres al mismo tiempo. El semáforo es un canal con buffer de capacidad 3.

```

package main

import (
    "fmt"
    "sync"
    "time"
)

func descargar(id int, sem chan struct{}, wg *sync.WaitGroup) {
    defer wg.Done()

    sem <- struct{}{} // pedir permiso
    defer func() { <-sem }() // devolverlo al terminar

    fmt.Printf("Descarga %d iniciada\n", id)
    time.Sleep(time.Second)
    fmt.Printf("Descarga %d terminada\n", id)
}

func main() {
    const maxSimultaneas = 3
    sem := make(chan struct{}, maxSimultaneas)
    var wg sync.WaitGroup

    for i := 1; i <= 8; i++ {
        wg.Add(1)
        go descargar(i, sem, &wg)
    }

    wg.Wait()
    fmt.Println("Todas las descargas terminaron")
}

```

Si observas la salida con atención, verás que siempre hay como máximo tres descargas "iniciada" sin su "terminada" correspondiente. Las demás esperan su turno en la fila del semáforo. Cambia maxSimultaneas a 8 y todo corre en paralelo; cámbialo a 1 y todo se vuelve secuencial. Un solo número controla el paralelismo de todo el sistema.

Ejemplo 11: Pipeline con fan-out en la etapa lenta

Integramos pipeline y fan-out: una etapa de generación, una etapa pesada repartida entre 3 workers y la impresión final. Todo conectado con canales.

```

package main

```

```

import (
    "fmt"
    "sync"
    "time"
)

// trabajoPesado simula una tarea que tarda 300 milisegundos.
func trabajoPesado(n int) int {
    time.Sleep(300 * time.Millisecond)
    return n * 10
}

func main() {
    entrada := make(chan int)
    salida := make(chan int)

    // Fan-out: 3 workers comparten la etapa pesada.
    var wg sync.WaitGroup
    for i := 0; i < 3; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            for n := range entrada {
                salida <- trabajoPesado(n)
            }
        }()
    }

    // Cuando los workers terminan, se cierra la salida.
    go func() {
        wg.Wait()
        close(salida)
    }()

    // Etapa de generacion.
    go func() {
        defer close(entrada)
        for i := 1; i <= 6; i++ {
            entrada <- i
        }
    }()

    for r := range salida {
        fmt.Println("Resultado:", r)
    }
}

```

```
}  
    fmt.Println("Pipeline terminado")  
}
```

Seis tareas de 300 ms con 3 workers toman alrededor de 600 ms en lugar de 1800 ms. El orden de los resultados varía entre ejecuciones porque los workers toman tareas según se desocupan: si el orden importara, habría que etiquetar cada tarea con su posición. Este ejemplo es, en miniatura, la arquitectura de muchos sistemas reales: generar, repartir lo pesado, reunir.

Práctica

Ejercicio 1: Pares e impares concurrentes

Lanza dos goroutines: una imprime los números pares del 2 al 10 y otra los impares del 1 al 9. Usa un `sync.WaitGroup` para que main espere a ambas antes de terminar.

Solución explicada: cada goroutine es una función con su propio ciclo. El `WaitGroup` cuenta dos tareas; cada goroutine llama `Done` con `defer` al terminar. El orden de impresión entre ambas es impredecible, y eso está bien: es la naturaleza de la concurrencia.

```

package main

import (
    "fmt"
    "sync"
)

func pares(wg *sync.WaitGroup) {
    defer wg.Done()
    for i := 2; i <= 10; i += 2 {
        fmt.Println("Par:", i)
    }
}

func impares(wg *sync.WaitGroup) {
    defer wg.Done()
    for i := 1; i <= 9; i += 2 {
        fmt.Println("Impar:", i)
    }
}

func main() {
    var wg sync.WaitGroup
    wg.Add(2)
    go pares(&wg)
    go impares(&wg)
    wg.Wait()
    fmt.Println("Listo")
}

```

Si quitas el `wg.Wait()` verás que el programa termina sin imprimir nada o a medias: `main` no espera a nadie por cortesía. Ese experimento vale más que mil explicaciones.

Ejercicio 2: Productor veloz, consumidor lento

Crea un canal con buffer de 5. Una goroutine productora envía los números del 1 al 10 lo más rápido posible (sin pausas). La goroutine consumidora los recibe con una pausa de 200 milisegundos entre cada uno e imprime el valor. Cierra el canal al terminar de producir y usa `for-range` para recibir.

Solución explicada: el buffer de 5 permite que el productor se adelante hasta cinco valores sin bloquearse; a partir de ahí tiene que esperar al consumidor. Verás en la salida que los primeros envíos son instantáneos y luego el ritmo lo marca el consumidor lento. Es la charola de tacos: se llena y el taquero espera.

```
package main

import (
    "fmt"
    "time"
)

func main() {
    canal := make(chan int, 5)

    go func() {
        for i := 1; i <= 10; i++ {
            canal <- i
            fmt.Println("Enviado", i)
        }
        close(canal)
    }()

    for n := range canal {
        time.Sleep(200 * time.Millisecond)
        fmt.Println("Recibido", n)
    }
    fmt.Println("Terminado")
}
```

Prueba cambiar el buffer a 0 y observa cómo cada envío espera su recepción: el programa se vuelve más lento pero la sincronización es total. Luego prueba con buffer de 10 y nota que el productor termina todo antes de que el consumidor procese el primero.

Ejercicio 3: Pipeline que calcula cuadrados

Arma un pipeline de dos etapas: la primera genera los números del 1 al 5, la segunda los eleva al cuadrado. Main imprime los resultados. Cada etapa debe cerrar su canal de salida.

Solución explicada: el patrón es generar -> transformar -> consumir. Cada etapa cierra su salida con defer para que la siguiente salga de su for-range. Si olvidas un close, el programa se queda colgado esperando: el deadlock clásico del pipeline.


```

package main

import "fmt"

func generar(salida chan int) {
    defer close(salida)
    for i := 1; i <= 5; i++ {
        salida <- i
    }
}

func cuadrados(entrada, salida chan int) {
    defer close(salida)
    for n := range entrada {
        salida <- n * n
    }
}

func main() {
    nums := make(chan int)
    cuad := make(chan int)
    go generar(nums)
    go cuadrados(nums, cuad)
    for c := range cuad {
        fmt.Println("Cuadrado:", c)
    }
}

```

Como reto extra, agrega una tercera etapa entre cuadrados y la impresión que sume 100 a cada valor. Notarás que solo tocas el main (un canal más y una goroutine más): las etapas existentes no cambian.

Ejercicio 4: El primero que responda gana

Crea dos canales. Lanza dos goroutines que simulan dos servicios: uno responde después de 1 segundo con "servidor A", otro después de 2 segundos con "servidor B". Usa select para tomar la respuesta del primero que llegue e imprimir de cuál vino.

Solución explicada: el select se bloquea hasta que uno de los casos puede avanzar. Como A responde primero, su caso gana. Si ambos estuvieran listos al mismo tiempo, el select elegiría al azar.

```

package main

import (
    "fmt"
    "time"
)

func servidor(nombre string, espera time.Duration, canal chan string) {
    time.Sleep(espera)
    canal <- nombre
}

func main() {
    a := make(chan string)
    b := make(chan string)

    go servidor("servidor A", time.Second, a)
    go servidor("servidor B", 2*time.Second, b)

    select {
    case r := <-a:
        fmt.Println("Gano", r)
    case r := <-b:
        fmt.Println("Gano", r)
    }
}

```

Invierte los tiempos y verifica que ahora gana B. Este es el esqueleto de patrones reales como "pregúntale a tres réplicas y quédate con la primera respuesta".

Ejercicio 5: Mini worker pool de 2 workers

Reparte 6 tareas entre 2 workers usando un canal de tareas. Cada tarea es un número; el worker imprime su id, el número y su cuadrado, simulando trabajo con 300 milisegundos de pausa. Cierra el canal al terminar de enviar y espera con WaitGroup.

Solución explicada: los dos workers leen del mismo canal compartido; Go reparte los valores automáticamente entre ellos. Con 6 tareas de 300 ms y 2 workers, el total ronda los 900 ms en lugar de 1800 ms secuenciales.

```

package main

import (
    "fmt"
    "sync"
    "time"
)

func worker(id int, tareas <-chan int, wg *sync.WaitGroup) {
    defer wg.Done()
    for t := range tareas {
        fmt.Printf("Worker %d: tarea %d -> cuadrado %d\n", id, t, t*t)
        time.Sleep(300 * time.Millisecond)
    }
}

func main() {
    tareas := make(chan int)
    var wg sync.WaitGroup

    for i := 1; i <= 2; i++ {
        wg.Add(1)
        go worker(i, tareas, &wg)
    }

    for t := 1; t <= 6; t++ {
        tareas <- t
    }
    close(tareas)
    wg.Wait()
    fmt.Println("Trabajo completado")
}

```

Observa que el parámetro es <-chan int (solo recepción): el worker no puede enviar por error a ese canal y el compilador lo garantiza. Pequeños detalles así hacen el código concurrente más seguro.

Ejercicio 6: Banco con depósitos concurrentes (reto integrador)

Crea un struct Cuenta con un saldo y un sync.Mutex. Implementa los métodos Depositar(monto) y Saldo() que protejan el acceso con el candado. Lanza 50 goroutines que depositan 10 pesos cada una y verifica con WaitGroup que el saldo final sea exactamente 500. Corre con go run -race para comprobar que no hay carreras.

Solución explicada: este ejercicio junta todo lo delicado: estado compartido, mutex por puntero (los métodos usan receptor de puntero para no copiar el candado), y verificación con el detector de carreras. El método Saldo también necesita el candado porque leer mientras otros escriben ya es una carrera.

```
package main

import (
    "fmt"
    "sync"
)

type Cuenta struct {
    mu    sync.Mutex
    saldo float64
}

func (c *Cuenta) Depositar(monto float64) {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.saldo += monto
}

func (c *Cuenta) Saldo() float64 {
    c.mu.Lock()
    defer c.mu.Unlock()
    return c.saldo
}

func main() {
    cuenta := &Cuenta{}
    var wg sync.WaitGroup

    for i := 0; i < 50; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            cuenta.Depositar(10)
        }()
    }

    wg.Wait()
    fmt.Printf("Saldo final: %.2f\n", cuenta.Saldo())
}
```

Si quitas los Lock/Unlock y corres con -race, el detector te mostrará la carrera entre las goroutines. Con el candado puesto, el reporte queda limpio y el saldo siempre es 500. Esa es la diferencia entre "creo que funciona" y "está verificado".

Ejercicio 7: Ping-pong entre dos goroutines

Dos goroutines se pasan una "pelota" (un entero) de ida y vuelta por un canal, 10 veces en total. Cada vez que una goroutine recibe la pelota, imprime quién la tiene y el número de golpe, la incrementa y la devuelve. Main inicia el juego enviando la pelota con valor 1 y espera a que termine.

Solución explicada: un solo canal bidireccional basta porque el turno se alterna solo: quien recibe, golpea y devuelve. El canal sin buffer garantiza que cada golpe espere su turno. Main usa un segundo canal (o un WaitGroup) para saber cuándo acabó el juego.

```

package main

import (
    "fmt"
    "sync"
)

func jugador(nombre string, mesa chan int, wg *sync.WaitGroup) {
    defer wg.Done()
    for golpe := range mesa {
        fmt.Printf("%s golpea la pelota (toque %d)\n", nombre, golpe)
        if golpe >= 10 {
            // Fin del juego: cerrar el canal despierta al otro jugador
            // para que su for-range termine y también llame Done.
            close(mesa)
            return
        }
        mesa <- golpe + 1
    }
    fmt.Printf("%s ve que el juego termino\n", nombre)
}

func main() {
    mesa := make(chan int)
    var wg sync.WaitGroup

    wg.Add(2)
    go jugador("Ana", mesa, &wg)
    go jugador("Luis", mesa, &wg)

    mesa <- 1
    wg.Wait()
    fmt.Println("Juego terminado")
}

```

Hay un detalle sutil que vale oro: quien hace el golpe 10 cierra el canal antes de salir. Sin ese `close`, el otro jugador se quedaría bloqueado para siempre en su `for-range` esperando una pelota que nunca llega, nunca llamaría `Done` y el `wg.Wait()` de `main` se colgaría eternamente: un deadlock de manual. El cierre es la forma correcta de decir "se acabó el juego" y despierta al compañero para que termine limpio. Y nota que solo uno de los dos llega al golpe 10, así que el canal se cierra exactamente una vez (cerrar dos veces provocaría pánico). Diseñar quién cierra cada canal es una de las decisiones más importantes en programas concurrentes.

Ejercicio 8: Suma paralela con fan-in (reto integrador)

Suma los números del 1 al 100 repartiendo el trabajo entre 4 goroutines: cada una suma su rango (1-25, 26-50, 51-75, 76-100) y envía su subtotal por un canal. Main recibe los 4 subtotales y los suma para obtener el total (debe dar 5050). Usa WaitGroup para coordinar y cierra el canal de resultados cuando todas terminen.

Solución explicada: es el patrón fan-out/fan-in en miniatura. Repartir el rango entre 4 trabajadores es el fan-out; juntar los 4 subtotales en el total es el fan-in. El cierre del canal tras el Wait permite usar for-range para recibir sin saber cuántos valores vienen.

```

package main

import (
    "fmt"
    "sync"
)

func sumarRango(inicio, fin int, resultados chan<- int, wg *sync.WaitGroup) {
    defer wg.Done()
    suma := 0
    for i := inicio; i <= fin; i++ {
        suma += i
    }
    resultados <- suma
}

func main() {
    resultados := make(chan int)
    var wg sync.WaitGroup

    rangos := [][]int{{1, 25}, {26, 50}, {51, 75}, {76, 100}}
    for _, r := range rangos {
        wg.Add(1)
        go sumarRango(r[0], r[1], resultados, &wg)
    }

    go func() {
        wg.Wait()
        close(resultados)
    }()

    total := 0
    for sub := range resultados {
        fmt.Println("Subtotal recibido:", sub)
        total += sub
    }
    fmt.Println("Total:", total)
}

```

Verifica que el total sea 5050. Como reto extra, cambia el número de trabajadores a 2 o a 10 (ajustando los rangos) y confirma que el resultado no cambia: la concurrencia reparte el trabajo, pero la matemática sigue siendo la misma.

Ejercicio 9: Alarma con ticker

Crea un ticker que dispare cada 300 milisegundos. Cada tick, una goroutine debe enviar el número de tick por un canal. Main recibe 4 ticks, los imprime, detiene el ticker y termina. Asegúrate de usar `defer ticker.Stop()`.

Solución explicada: el ticker vive en main (o en quien lo necesite) y su canal se recorre con `for-range`. Como solo queremos 4 ticks, contamos y rompemos el ciclo. El `Stop` con `defer` garantiza que no quede la goroutine interna viva.

```
package main

import (
    "fmt"
    "time"
)

func main() {
    ticker := time.NewTicker(300 * time.Millisecond)
    defer ticker.Stop()

    contador := 0
    for range ticker.C {
        contador++
        fmt.Println("Alarma", contador, ": han pasado", contador*300, "ms")
        if contador >= 4 {
            break
        }
    }
    fmt.Println("Alarma desactivada")
}
```

Como variante, mueve el conteo a una goroutine que envíe por un canal los ticks y que main reciba: practicarás la combinación de ticker + canal + goroutine, que es exactamente como se construyen los monitores en sistemas reales.

Ejercicio 10: Procesador con límite de concurrencia (reto integrador)

Procesa 10 tareas donde cada una tarda 400 milisegundos, pero permite máximo 2 tareas simultáneas usando un semáforo de canal con buffer. Mide el tiempo total con `time.Now()` y `time.Since()` e imprímelo: debe rondar los 2 segundos (10 tareas / 2 simultáneas * 400 ms), no los 4 segundos de la versión secuencial. Usa `WaitGroup` para esperar a todas.

Solución explicada: este ejercicio junta semáforo, WaitGroup y medición de tiempo. El semáforo acota el paralelismo; el WaitGroup coordina el final; la medición confirma que el límite se respetó. Si el tiempo da cerca de 800 ms, el semáforo no está limitando (revisa la capacidad); si da 4 segundos, algo se serializó de más.

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func tarea(id int, sem chan struct{}, wg *sync.WaitGroup) {
    defer wg.Done()

    sem <- struct{}{}
    defer func() { <-sem }()

    fmt.Printf("Tarea %d en proceso\n", id)
    time.Sleep(400 * time.Millisecond)
    fmt.Printf("Tarea %d lista\n", id)
}

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

    sem := make(chan struct{}, 2) // maximo 2 simultaneas
    var wg sync.WaitGroup

    for i := 1; i <= 10; i++ {
        wg.Add(1)
        go tarea(i, sem, &wg)
    }

    wg.Wait()
    fmt.Println("Tiempo total:", time.Since(inicio).Round(time.Millisecond))
}
```

El tiempo impreso debe andar por los 2 segundos. Prueba cambiar la capacidad del semáforo a 5 y observa cómo el tiempo baja a unos 800 ms, y a 10 para ver el paralelismo total. Con un solo número controlas el ritmo de todo el sistema: esa es la elegancia del semáforo con canal.

Ejercicio 11: Carrera de relevos (reto integrador)

Tres corredores (goroutines) deben pasar un testimonio en orden estricto: Ana -> Luis -> Marta. Cada corredor espera el testimonio por su canal de entrada, anuncia que lo recibió, le agrega su nombre y lo pasa al siguiente por su canal de salida. Main inicia la carrera enviando "salida" al primer canal y recibe el testimonio final por un canal meta. El resultado debe imprimirse como "salida -> Ana -> Luis -> Marta".

Solución explicada: este ejercicio es la respuesta práctica a la subsección anterior: cuando el orden importa, lo impones con canales encadenados. Cada corredor se bloquea hasta recibir, así que el orden es físicamente imposible de romper, sin importar lo que haga el planificador. Es un pipeline de una sola pieza de datos.

```
package main

import "fmt"

func corredor(nombre string, entrada <-chan string, salida chan<- string) {
    testimonio := <-entrada
    fmt.Printf("%s recibe el testimonio\n", nombre)
    salida <- testimonio + " -> " + nombre
}

func main() {
    e1 := make(chan string)
    e2 := make(chan string)
    e3 := make(chan string)
    meta := make(chan string)

    go corredor("Ana", e1, e2)
    go corredor("Luis", e2, e3)
    go corredor("Marta", e3, meta)

    e1 <- "salida"
    fmt.Println("Recorrido:", <-meta)
}
```

Nota los canales direccionales en la firma: cada corredor declara si su canal es de entrada o de salida, y el compilador impediría usarlos al revés. Como reto extra, agrega un cuarto corredor o haz que el testimonio dé dos vueltas completas antes de llegar a la meta (pista: la meta puede reenviar a e1 una vez más). Jugar con este ejemplo te entrena a pensar en flujo de datos en lugar de pasos secuenciales, que es exactamente el cambio de mentalidad que pide la concurrencia.

Ejercicio 12: Búsqueda paralela en un slice

Divide un slice de 10 enteros en dos mitades. Lanza dos goroutines, cada una buscando un número objetivo en su mitad: si lo encuentra, envía su posición dentro de la mitad; si no, envía -1. Usa un canal con buffer de 2 para recibir ambos resultados sin bloqueos y determina en qué mitad estaba el objetivo.

Solución explicada: partir los datos y buscar en paralelo es el fan-out aplicado a un problema clásico. El buffer de 2 evita que una goroutine se bloquee esperando a que main reciba, y como ambas siempre envían exactamente un valor, main puede recibir dos veces con tranquilidad.

```

package main

import "fmt"

func buscar(nombre string, datos []int, objetivo int, resultado chan<- int) {
    for i, v := range datos {
        if v == objetivo {
            fmt.Printf("%s lo encontro en la posicion %d de su mitad\n", nombre, i)
            resultado <- i
            return
        }
    }
    fmt.Printf("%s no lo encontro en su mitad\n", nombre)
    resultado <- -1
}

func main() {
    datos := []int{4, 8, 15, 16, 23, 42, 7, 19, 3, 11}
    objetivo := 42
    mitad := len(datos) / 2

    resultado := make(chan int, 2)
    go buscar("A", datos[:mitad], objetivo, resultado)
    go buscar("B", datos[mitad:], objetivo, resultado)

    r1 := <-resultado
    r2 := <-resultado
    fmt.Println("Resultados:", r1, r2)
    if r1 == -1 && r2 == -1 {
        fmt.Println("El objetivo no esta en el slice")
    } else {
        fmt.Println("El objetivo si aparece")
    }
}

```

El 42 está en el índice 5 del slice completo, es decir, en la posición 0 de la mitad de B, así que verás que B lo encuentra y A reporta -1. Cambia el objetivo a 99 y verás ambos -1. Este patrón escala a N partes: es la base de la búsqueda paralela en sistemas reales.

Resumen de la unidad

En esta unidad aprendiste la diferencia entre concurrencia y paralelismo: la concurrencia es organizar tu programa en tareas independientes que avanzan sin estorbarse, y el paralelismo es ejecutarlas físicamente al mismo tiempo cuando hay varios núcleos. Go te da la concurrencia con las goroutines, tareas ultraligeras que se lanzan con la palabra `go`, y el runtime las reparte por los núcleos disponibles para darte paralelismo gratis.

Descubriste la primera regla de oro: cuando `main` termina, todo termina, así que coordinas tus goroutines con `sync.WaitGroup` usando `Add` para contar tareas, `Done` con `defer` en cada goroutine y `Wait` para esperarlas a todas, pasando el `WaitGroup` siempre por puntero. Luego conociste los canales, los tubos tipados por donde las goroutines se comunican: se crean con `make(chan T)`, se envía con `canal <- valor` y se recibe con `<-canal`. Viste que un canal sin buffer sincroniza con un apretón de manos directo, mientras que uno con buffer permite que el productor se adelante hasta llenar la fila.

Aprendiste a cerrar canales con `close` (lo hace quien envía, nunca quien recibe), a recorrerlos con `for-range` hasta el cierre y a detectar el cierre manualmente con la coma-`ok`. Conociste `select` para esperar en varios canales a la vez, `time.After` para poner timeouts que evitan esperas eternas y `default` para el modo no bloqueante. Estudiaste los tres patrones clásicos: el pipeline como línea de ensamble por etapas, el fan-out/fan-in para repartir trabajo pesado y reunir resultados, y el worker pool para limitar cuántas tareas corren a la vez.

También aprendiste cuándo los canales no son la respuesta: `sync.Mutex` protege el estado compartido como contadores y cachés, con la regla de usarlo por puntero y liberar con `defer`. Viste cómo decidir entre canales (tareas que se mandan cosas) y mutex (tareas que tocan lo mismo), y cómo verificar tu código con `go run -race`. Y te advertimos de los tres errores clásicos: el deadlock donde todos se esperan para siempre, la fuga de goroutines que nunca terminan y la copia prohibida de `Mutex` y `WaitGroup`.

La idea que debes llevarte es el proverbio de Go: no te comuniques compartiendo memoria, comparte memoria comunicándote. Diseña tareas independientes que se hablan por canales, protege con mutex solo el estado que de verdad se comparte y verifica con el detector de carreras. Con interfaces sabías diseñar programas flexibles; con concurrencia sabes hacerlos rápidos. Ya tienes las dos superpotencias del programador Go. Úsalas con respeto, como el cuchillo del taquero.

UNIDAD 9: Manejo de errores y testing

En las unidades anteriores aprendiste a construir programas en Go que funcionan: variables, funciones, estructuras de datos, paquetes, goroutines y canales. Todo eso es el "camino feliz" del programa, el escenario en el que nada falla. Pero los programas de verdad, los que corren en servidores reales, los que usan personas reales, viven en un mundo donde las cosas fallan todos los días: un archivo que no existe, una red que se cae, una base de datos que se satura, un usuario que manda datos inválidos. Un programador que solo sabe manejar el camino feliz es un programador que apenas está empezando.

Esta unidad te enseña la filosofía completa de Go sobre los errores. Y es importante que entiendas que en Go el manejo de errores no es un tema aparte que se aprende "después": es parte del diseño del lenguaje. Go decidió hacer las cosas de forma deliberadamente distinta a otros lenguajes. En Java, en Python, en C#, los errores viajan por el programa como excepciones que saltan de un lugar a otro hasta que alguien las atrape. En Go, los errores son valores. Valores normales, que se regresan como cualquier otro valor, que se pueden guardar en variables, comparar, envolver y pasar. Esa decisión cambia todo el estilo de programación y, una vez que la interiorizas, te hace escribir código más honesto y más fácil de razonar.

También veremos las excepciones a la regla: panic y recover. En Go, panic existe, pero no es para el manejo de errores normal. Es para situaciones que realmente no deberían pasar nunca: errores de programador, configuraciones imposibles, estados contradictorios. Y recover te permite rescatarte de un panic en el límite de tu programa, para que un servidor no se muera por completo por un error inesperado.

La segunda mitad de la unidad está dedicada al testing. Go trae el testing integrado en el lenguaje y en la herramienta go: no necesitas instalar frameworks ni configuraciones complicadas. Aprenderás a escribir pruebas unitarias con el paquete testing, a usar el estilo idiomático de table-driven tests (pruebas basadas en tablas) que usa toda la comunidad de Go, a medir el rendimiento con benchmarks y a verificar que tu código esté bien probado con la cobertura de código.

Al terminar esta unidad vas a poder escribir programas que no se rompen en silencio, que informan de sus errores con claridad y que vienen con pruebas que demuestran que funcionan. Ese es el salto de "alguien que escribe código que corre" a "un desarrollador profesional en el que se puede confiar".

Explicación detallada

Los errores son valores, no excepciones

Esta es la idea central de toda la unidad, y vale la pena entenderla a fondo. En muchos lenguajes, cuando algo falla, el lenguaje "lanza" una excepción que interrumpe el flujo normal del programa y salta hasta el primer bloque catch que la pueda manejar. El problema de ese modelo es que el camino que sigue el error es invisible: leyendo el código no sabes qué funciones pueden fallar ni dónde se va a manejar el problema, porque las excepciones pueden saltar a través de muchas capas sin que ninguna firma de función lo anuncie.

Go tomó el camino opuesto: un error es un valor como cualquier otro. Las funciones que pueden fallar regresan, además de su resultado normal, un valor de tipo error. Y el programador lo revisa de forma explícita justo después de llamar a la función. El patrón más famoso de Go es este:

```
resultado, err := algunaFuncion()
if err != nil {
    // aquí se maneja el error
    return err
}
// aquí se usa resultado con confianza
```

Esta forma de escribir tiene dos consecuencias profundas. La primera es honestidad: la firma de la función te dice si puede fallar. Si una función regresa un error, sabes que puede fallar. Si no regresa error, no puede fallar (al menos no de formas que el programador haya previsto). No hay sorpresas escondidas.

La segunda consecuencia es que el manejo de errores es local y explícito. Ves exactamente en qué punto del código se puede fallar y qué se hace al respecto. Para algunos programadores que vienen de otros lenguajes esto se siente repetitivo al principio: "¿de verdad tengo que escribir `if err != nil` después de cada llamada?". La respuesta de la comunidad de Go es que sí, y que esa repetición es el precio de la claridad. Con el tiempo, tus ojos aprenden a leer ese patrón sin esfuerzo, y empiezas a apreciar que nunca hay duda sobre qué pasa cuando algo falla.

El tipo error es una interfaz. Es una de las interfaces más simples del lenguaje:

```
type error interface {
    Error() string
}
```


Cualquier tipo que tenga un método `Error()` que regrese un string es un error. Eso es todo. Esta simplicidad es deliberada: significa que puedes crear tus propios tipos de error con toda la información que necesites (códigos, contextos, datos adjuntos), y el lenguaje los trata igual que a cualquier error. Más adelante en esta unidad crearemos errores personalizados con `struct`.

Un detalle importante: el valor `nil` de tipo error significa "no hubo error". Por eso el patrón es comparar contra `nil`. Una función bien escrita regresa `nil` como error cuando todo salió bien, y un error no `nil` cuando algo falló. Nunca regreses un error `nil` junto con datos inválidos, y nunca regreses datos válidos junto con un error no `nil`: esa inconsistencia es fuente de bugs difíciles de rastrear.

Analogía: el error como paquete de regreso

Piensa en una función como un mensajero al que le pides que vaya a la tienda por leche. En el modelo de excepciones, el mensajero podría desaparecer misteriosamente a mitad del camino (la excepción salta) y tú te enteras mucho después, quién sabe dónde. En el modelo de Go, el mensajero siempre regresa a tu puerta. A veces trae la leche (el resultado). A veces trae la leche y una nota que dice "casi no alcanzo, la tienda iba a cerrar" (resultado con error, aunque esto es raro). Y a veces trae las manos vacías con una nota que explica qué pasó: "la tienda estaba cerrada" (error). Tú siempre recibes algo de vuelta, y la nota te dice exactamente qué pasó. No hay desapariciones misteriosas.

Otra analogía útil: los errores en Go son como los recibos de una transacción bancaria. Cada operación te da un recibo. Si la operación salió bien, el recibo dice "exitoso" (`nil`). Si falló, el recibo dice por qué. Y un buen cajero (tu código) revisa el recibo antes de seguir con lo siguiente, en lugar de asumir que todo salió bien.

El patrón `if err != nil` en la práctica

Veamos el patrón en acción con código real de la biblioteca estándar. Abrir un archivo es la operación clásica que puede fallar:

```

package main

import (
    "fmt"
    "os"
)

func main() {
    datos, err := os.ReadFile("config.txt")
    if err != nil {
        fmt.Println("No se pudo leer el archivo:", err)
        return
    }
    fmt.Println("Contenido:", string(datos))
}

```

Observa la estructura: la llamada y la revisión del error van pegadas. Esto no es casualidad: es el estilo que la comunidad recomienda porque mantiene el "camino feliz" (el código que se ejecuta cuando todo sale bien) con la menor indentación posible. El manejo del error se resuelve rápido (con return, continue, break, o log.Fatal) y el código principal sigue en el nivel de indentación más bajo. Este estilo se llama a veces "early return" (retorno temprano) y es una de las marcas de un código Go bien escrito.

Cuando estás dentro de una función que también regresa error, lo idiomático es propagar el error hacia arriba:

```

func leerConfig(ruta string) ([]byte, error) {
    datos, err := os.ReadFile(ruta)
    if err != nil {
        return nil, err
    }
    return datos, nil
}

```

Fíjate en que se regresa nil como primer valor cuando hay error. La convención es que los demás valores de retorno no son confiables cuando err no es nil. Quien llame a leerConfig debe revisar el error antes de tocar los datos.

Errores comunes de principiantes con err

Hay varios errores típicos que cometen quienes están aprendiendo este patrón. El primero y más peligroso es ignorar el error con el identificador en blanco:

```
datos, _ := os.ReadFile("config.txt") // MAL: el error se ignora
fmt.Println(string(datos))
```

Usar `_` para descartar el error es válido en Go (compila sin problema), pero es una mala práctica casi siempre. Si el archivo no existe, `datos` será `nil` y el programa seguirá como si nada, produciendo resultados incorrectos o un panic más adelante, lejos del verdadero origen del problema. La única excepción aceptable es cuando estás absolutamente seguro de que la operación no puede fallar, y aun así conviene documentarlo con un comentario.

El segundo error común es seguir usando el valor aunque `err` no sea `nil`:

```
datos, err := os.ReadFile("config.txt")
if err != nil {
    fmt.Println("Hubo un error:", err)
}
fmt.Println(string(datos)) // MAL: se usa datos aunque pudo fallar
```

Si hubo error, `datos` probablemente es `nil` o está incompleto. El manejo del error debe interrumpir el flujo normal: con `return`, con un `else`, o reestructurando el código.

El tercer error común es el sombreado accidental de `err` con `:=` en bloques anidados:

```
datos, err := os.ReadFile("a.txt")
if err != nil {
    return err
}
if condicion {
    datos2, err := os.ReadFile("b.txt") // este err es NUEVO, solo vive en el if
    _ = datos2
}
```

Esto no es un error en sí mismo (a veces es lo que quieres), pero puede confundir cuando esperas que el `err` de afuera se actualice. La regla: `:=` crea variables nuevas en el ámbito actual. Si quieres reutilizar el `err` de afuera dentro de un bloque, usa `=` con variables ya declaradas.

El cuarto error es comparar errores con `==` cuando no deberías, o no saber cuándo sí se puede. Los errores centinela (como `io.EOF`) sí se comparan con `==`, pero los errores envueltos requieren `errors.Is`. Lo veremos en detalle más adelante.

Crear errores: errors.New y fmt.Errorf

Hasta ahora hemos manejado errores que regresan otras funciones. Pero tus propias funciones también necesitan crear errores cuando detectan un problema. El paquete errors de la biblioteca estándar te da la forma más simple:

```
package main

import (
    "errors"
    "fmt"
)

func dividir(a, b float64) (float64, error) {
    if b == 0 {
        return 0, errors.New("division entre cero")
    }
    return a / b, nil
}

func main() {
    resultado, err := dividir(10, 0)
    if err != nil {
        fmt.Println("Error:", err)
        return
    }
    fmt.Println("Resultado:", resultado)
}
```

errors.New crea un error con un mensaje fijo. Es perfecto para errores simples y estáticos. Pero muchas veces necesitas incluir valores en el mensaje: qué archivo falló, qué número era inválido, cuántos reintentos se hicieron. Para eso está fmt.Errorf, que funciona como fmt.Sprintf pero regresa un error:

```
func buscarUsuario(id int) (string, error) {
    if id <= 0 {
        return "", fmt.Errorf("id de usuario invalido: %d", id)
    }
    // ... buscar en base de datos ...
    return "Esteban", nil
}
```

Buenas prácticas para los mensajes de error: escríbelos en minúsculas, sin punto final, y que sean descriptivos pero concisos. "archivo de configuración no encontrado: config.yaml" es mejor que "error". Y evita poner mayúsculas al inicio porque los mensajes se suelen concatenar o envolver con otros mensajes. Esta convención viene del estilo de la biblioteca estándar de Go y la sigue toda la comunidad.

Envolver errores: el verbo %w

Cuando una función de bajo nivel falla y tú propagas el error hacia arriba, a veces quieres agregar contexto: no solo "qué falló" sino "dónde falló" o "mientras hacía qué". Para eso existe el envoltorio de errores, introducido en Go 1.13 con el verbo %w de `fmt.Errorf`:

```
func cargarConfiguracion(ruta string) ([]byte, error) {
    datos, err := os.ReadFile(ruta)
    if err != nil {
        return nil, fmt.Errorf("no se pudo cargar la configuracion %q: %w", ruta,
err)
    }
    return datos, nil
}
```

El %w "envuelve" el error original dentro del nuevo error. El mensaje final será algo como: `no se pudo cargar la configuracion "config.yaml": open config.yaml: no such file or directory`. Tienes el contexto de alto nivel (qué intentabas hacer) más el detalle de bajo nivel (qué falló exactamente).

¿Por qué envolver en lugar de simplemente concatenar el mensaje con %v o %s? Porque el error envuelto conserva la identidad del error original. Más adelante, quien maneje el error puede "desenvolverlo" y preguntar: "¿el error original fue un error de archivo no encontrado?" o "¿fue un error de tiempo de espera agotado?". Si solo concatenas el texto, esa información se pierde: el texto ya no es el error original, es solo una cadena.

La cadena de envoltura forma una lista enlazada de errores: cada error envuelto apunta al anterior mediante el método `Unwrap()` error. Puedes desenvolver manualmente, pero lo normal es usar las funciones del paquete `errors`: `errors.Is` y `errors.As`.

errors.Is: preguntar por la causa

`errors.Is` recorre la cadena de errores envueltos y te dice si alguno de ellos es (o contiene) un error específico. Esto es útil con los errores "centinela" de la biblioteca estándar, como `os.ErrNotExist` o `io.EOF`:

```

package main

import (
    "errors"
    "fmt"
    "os"
)

func main() {
    _, err := os.ReadFile("archivo-que-no-existe.txt")
    if err != nil {
        if errors.Is(err, os.ErrNotExist) {
            fmt.Println("El archivo no existe, usando valores por defecto")
        } else {
            fmt.Println("Otro error:", err)
        }
    }
}

```

Sin `errors.Is`, tendrías que comparar con `==`, y eso falla cuando el error viene envuelto varias capas arriba. `errors.Is` desenrolla toda la cadena por ti. También puedes definir tus propios errores centinela en tu paquete:

```

package usuarios

import "errors"

var ErrNoEncontrado = errors.New("usuario no encontrado")

func Buscar(id int) (string, error) {
    // ...
    return "", fmt.Errorf("buscar usuario %d: %w", id, ErrNoEncontrado)
}

```

Y quien use tu paquete puede verificar con `errors.Is(err, usuarios.ErrNoEncontrado)` sin importar cómo lo envolviste por dentro. Esta es una forma limpia de diseñar APIs: expones errores centinela documentados y el usuario decide cómo reaccionar a cada uno.

errors.As: extraer el tipo concreto

A veces no te basta saber si el error "es" uno específico: necesitas acceder a los datos del error concreto. Por ejemplo, `*os.PathError` tiene campos con la operación que falló y la ruta. `errors.As` busca en la cadena el primer error que sea del tipo que le pides y lo extrae:

```
package main

import (
    "errors"
    "fmt"
    "os"
)

func main() {
    _, err := os.ReadFile("/ruta/protegida/secreto.txt")
    if err != nil {
        var errRuta *os.PathError
        if errors.As(err, &errRuta) {
            fmt.Println("Operacion que fallo:", errRuta.Op)
            fmt.Println("Ruta:", errRuta.Path)
            fmt.Println("Error interno:", errRuta.Err)
        }
    }
}
```

Fíjate en los detalles: `errors.As` recibe un apuntador al tipo que buscas (`&errRuta` donde `errRuta` es `*os.PathError`). Si encuentra un error de ese tipo en la cadena, lo asigna a tu variable y regresa `true`. Esto es muy poderoso porque te permite manejar errores con lógica específica según su tipo, sin importar cuántas capas de envoltura haya encima.

La diferencia entre `Is` y `As`, en una frase: `Is` pregunta "¿este error contiene a este error específico?" (comparación por identidad o por método `Is`), mientras que `As` pregunta "¿este error contiene un error de este tipo? y dámelo para usarlo".

Errores personalizados: structs que implementan `Error()`

Como error es una interfaz con un solo método, puedes crear tus propios tipos de error con toda la información estructurada que necesites. Esto es ideal cuando el que maneja el error necesita más que un mensaje: códigos de error, datos de contexto, sugerencias:

```
package main
```

```

import (
    "fmt"
)

// ErrorValidacion describe un problema con datos de entrada.
type ErrorValidacion struct {
    Campo    string
    Valor    string
    Motivo   string
   Codigo   int
}

func (e *ErrorValidacion) Error() string {
    return fmt.Sprintf("validacion fallida en campo %q (valor %q): %s [codigo %d]",
        e.Campo, e.Valor, e.Motivo, e.Codigo)
}

func validarEdad(edad int) error {
    if edad < 0 {
        return &ErrorValidacion{
            Campo: "edad",
            Valor:  fmt.Sprint(edad),
            Motivo: "la edad no puede ser negativa",
            Codigo: 1001,
        }
    }
    if edad > 150 {
        return &ErrorValidacion{
            Campo: "edad",
            Valor:  fmt.Sprint(edad),
            Motivo: "la edad no puede ser mayor a 150",
            Codigo: 1002,
        }
    }
    return nil
}

func main() {
    err := validarEdad(-5)
    if err != nil {
        fmt.Println(err)
        // Si necesito los datos estructurados:
        if ev, ok := err.(*ErrorValidacion); ok {
            fmt.Println("Codigo de error:", ev.Codigo)
        }
    }
}

```



```
}  
}
```

Nota que el método `Error()` está definido sobre el apuntador `*ErrorValidacion`. Es una convención común definirlo sobre el apuntador para evitar copiar el struct, aunque también funciona sobre el valor. Lo importante es ser consistente: si el método está en el apuntador, siempre regresa apuntadores (`&ErrorValidacion{...}`), no valores.

Combinando errores personalizados con envoltura (`%w`) y `errors.As`, obtienes un sistema de errores muy expresivo: errores con datos estructurados que viajan por las capas del programa conservando su identidad y su contexto. Este es el nivel de manejo de errores que se espera en código Go profesional.

Buenas prácticas del manejo de errores

Antes de pasar a `panic` y `testing`, reunamos las buenas prácticas en una lista que puedas consultar:

1. Maneja los errores donde tengas contexto para hacerlo bien. No tiene sentido manejar un error de base de datos en la capa más baja si ahí no puedes decidir nada útil; envuélvelo con contexto y déjalo subir hasta donde sí se pueda decidir (reintentar, usar un valor por defecto, mostrar un mensaje al usuario).
2. Envuelve con `%w` cuando agregues contexto, no con `%v`. Así conservas la cadena para `errors.Is` y `errors.As`.
3. No abuses de los errores centinela: úsalos para condiciones que el usuario de tu código realmente necesite distinguir (no encontrado, ya existe, sin permiso).
4. Los mensajes de error van en minúsculas, sin punto final, y describen qué pasó, no culpan al usuario.
5. Nunca ignores errores con `_` a menos que tengas una razón documentada.
6. En funciones que regresan (valor, error), regresa el valor cero cuando hay error.
7. No uses `panic` para errores esperados: `panic` es para lo inesperado.

panic: cuando algo realmente no debería pasar

panic es el mecanismo de Go para situaciones excepcionales que indican un bug o un estado imposible. Cuando llamas a panic, el programa empieza a desenrollar la pila de llamadas función por función, ejecutando los defer que encuentre, hasta que el programa termina (con un mensaje de error y un stack trace) o hasta que alguien lo detiene con recover.

¿Cuándo Sí usar panic? La regla general: para errores de programador y condiciones que "no pueden pasar" si el código es correcto. Ejemplos legítimos:

- Índices fuera de rango en un slice (esto lo hace el runtime automáticamente).
- Una función que recibe un argumento que viola su contrato documentado y no hay forma razonable de continuar.
- Fallos en la inicialización del programa: si tu servidor no puede leer su configuración obligatoria o no puede abrir el puerto, no tiene sentido seguir. Aquí es común usar log.Fatal, que internamente termina el programa (aunque log.Fatal usa os.Exit, no panic; lo veremos en la sección de logging).

```
package main

import "fmt"

func obtenerElemento(slice []string, indice int) string {
    if indice < 0 || indice >= len(slice) {
        panic(fmt.Sprintf("indice %d fuera de rango para slice de longitud %d",
            indice, len(slice)))
    }
    return slice[indice]
}

func main() {
    nombres := []string{"Ana", "Luis", "Maria"}
    fmt.Println(obtenerElemento(nombres, 1)) // bien: Luis
    // fmt.Println(obtenerElemento(nombres, 10)) // panic: indice fuera de rango
}
```

¿Cuándo NO usar panic? Para errores esperados y recuperables: un archivo que no existe, una entrada de usuario inválida, un timeout de red, una respuesta inesperada de una API. Todos esos son errores normales que se manejan con el patrón if err != nil. Si usas panic para errores de operación normal, estás usando el mecanismo equivocado y tu programa se volverá frágil: cualquier condición adversa lo tumbará en lugar de manejarla con gracia.

Una analogía: panic es como la alarma de incendios del edificio. No la activas porque se quemó una tortilla en la cocina (eso lo manejas con el extintor = if err != nil). La activas cuando hay fuego real que amenaza la estructura (un bug grave, un estado imposible). Y recover es el protocolo de evacuación: no evita el incendio, pero evita que todo el edificio colapse por un problema localizado.

recover con defer: el protocolo de evacuación

recover es una función especial que solo funciona cuando se llama dentro de una función diferida (defer) mientras la pila se está desenrollando por un panic. Si la llamas en cualquier otro momento, regresa nil y no hace nada. Su trabajo: detener el panic y regresar el valor que se pasó a panic, permitiendo que el programa continúe.

El patrón completo se ve así:

```
package main

import "fmt"

func tareaRiesgosa() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Se recupero de un panic:", r)
        }
    }()

    fmt.Println("Iniciando tarea...")
    panic("algo salio muy mal")
    fmt.Println("Esta linea nunca se ejecuta")
}

func main() {
    tareaRiesgosa()
    fmt.Println("El programa sigue vivo despues del recover")
}
```

El defer se registra al entrar a la función. Cuando el panic ocurre, Go empieza a desenrollar la pila y ejecuta las funciones diferidas. Dentro del defer, recover() detiene el desenrollado y regresa el valor del panic ("algo salio muy mal"). La función tareaRiesgosa termina de forma anormal pero controlada, y main continúa su ejecución normal.

¿Dónde se usa esto en la práctica? El caso más común es en servidores web: cada petición HTTP se maneja típicamente en su propia goroutine. Si el código que atiende una petición entra en panic por un bug, sin recover el panic tumbaría todo el servidor y todas las demás peticiones. Con un middleware de recover, solo esa petición falla (se regresa un error 500) y el servidor sigue atendiendo a los demás. En la Unidad 10, cuando construyamos servidores web, este patrón aparecerá de forma natural.

Otro uso: bibliotecas que procesan datos de terceros (parsers, por ejemplo) a veces usan panic/recover internamente para simplificar el código de propagación de errores graves de formato, recuperándose en el límite de la biblioteca y convirtiendo el panic en un error normal para el usuario. Esto es una técnica avanzada y debe usarse con cuidado: nunca dejes que un panic cruce los límites de tu paquete sin documentarlo.

Advertencia importante: recover solo atrapa panics, no errores normales. Y un recover mal usado puede ocultar bugs: si recuperas todos los panics indiscriminadamente y sigues como si nada, tu programa puede continuar en un estado corrupto. Úsalo en límites bien definidos (una petición, una tarea en background) y registra siempre el panic con un log para poder investigar después.

Logging: el paquete log

Cuando algo falla en producción, no hay nadie mirando la pantalla. Por eso los programas serios registran lo que hacen en logs: líneas de texto con marcas de tiempo que cuentan la historia de la ejecución. Go incluye el paquete log en la biblioteca estándar, simple y suficiente para la mayoría de los casos:

```

package main

import (
    "log"
    "os"
)

func main() {
    log.Println("Iniciando el servidor...")
    log.Printf("Puerto configurado: %d", 8080)

    datos, err := os.ReadFile("config.yaml")
    if err != nil {
        log.Fatalf("No se pudo leer la configuracion: %v", err)
    }
    _ = datos
    log.Println("Configuracion cargada correctamente")
}

```

Las funciones principales: `log.Println` y `log.Printf` escriben mensajes con la fecha y hora automáticamente. `log.Fatal` y `log.Fatalf` escriben el mensaje y terminan el programa con `os.Exit(1)`. Ojo: `log.Fatal` NO ejecuta los `defer` pendientes, porque `os.Exit` termina el programa de inmediato. Por eso `log.Fatal` se usa típicamente solo en `main`, durante la inicialización, donde no hay recursos importantes que limpiar con `defer`.

Para programas más grandes existen bibliotecas de logging estructurado (como `slog`, que viene en la biblioteca estándar desde Go 1.21, o `zap` de terceros), que en lugar de texto libre emiten logs en formato clave=valor o JSON, fáciles de procesar con herramientas. Pero el paquete `log` clásico es el punto de partida correcto: simple, sin dependencias, y suficiente para la mayoría de los programas que vas a escribir ahora.

Buena práctica de logging: registra los errores con contexto suficiente para diagnosticar el problema después (qué operación, con qué parámetros, a qué hora), pero sin registrar datos sensibles (contraseñas, tokens, información personal). Y usa niveles de forma consistente: información para el flujo normal importante, errores para lo que falló.

Testing en Go: archivos `_test.go`

Go trae las pruebas integradas: no necesitas instalar nada. La convención es simple: por cada archivo fuente ``mates.go``, creas un archivo ``mates_test.go`` en el mismo paquete, con funciones de prueba. La herramienta ``go test`` los encuentra y ejecuta automáticamente.

Una función de prueba tiene esta firma exacta:

```
func TestNombre(t *testing.T) {  
    // ...  
}
```

Debe empezar con ``Test``, recibir un ``*testing.T``, y no regresar nada. Si algo falla, llamas a ``t.Error`` o ``t.Errorf`` (marca la prueba como fallida pero continúa) o ``t.Fatal`` / ``t.Fatalf`` (marca como fallida y detiene esa prueba de inmediato).

Comandos esenciales:

- ``go test`` : ejecuta las pruebas del paquete actual. Si todo pasa, imprime ``ok``.
- ``go test -v`` : modo verboso, muestra cada prueba con su resultado (PASS/FAIL).
- ``go test -run Nombre`` : ejecuta solo las pruebas cuyo nombre coincida con la expresión regular `Nombre`. Útil para correr una sola prueba mientras depuras.
- ``go test ./...`` : ejecuta las pruebas de todos los paquetes del módulo.

Veamos un ejemplo completo. Supón que tienes este archivo ``mates.go``:

```
package mates  
  
// Suma regresa la suma de dos enteros.  
func Suma(a, b int) int {  
    return a + b  
}  
  
// Dividir regresa el cociente de a entre b, o un error si b es cero.  
func Dividir(a, b float64) (float64, error) {  
    if b == 0 {  
        return 0, errors.New("division entre cero")  
    }  
    return a / b, nil  
}
```

Y este archivo ``mates_test.go``:

```

package mates

import "testing"

func TestSuma(t *testing.T) {
    resultado := Suma(2, 3)
    esperado := 5
    if resultado != esperado {
        t.Errorf("Suma(2, 3) = %d; se esperaba %d", resultado, esperado)
    }
}

func TestDividir(t *testing.T) {
    resultado, err := Dividir(10, 2)
    if err != nil {
        t.Fatalf("Dividir(10, 2) regreso error inesperado: %v", err)
    }
    if resultado != 5 {
        t.Errorf("Dividir(10, 2) = %v; se esperaba 5", resultado)
    }
}

func TestDividirEntreCero(t *testing.T) {
    _, err := Dividir(10, 0)
    if err == nil {
        t.Error("Dividir(10, 0) deberia regresar un error, pero regreso nil")
    }
}

```

Ejecutando `go test -v` verías algo como:

```

=== RUN   TestSuma
--- PASS: TestSuma (0.00s)
=== RUN   TestDividir
--- PASS: TestDividir (0.00s)
=== RUN   TestDividirEntreCero
--- PASS: TestDividirEntreCero (0.00s)
PASS
ok      ejemplo/mates    0.123s

```

Fíjate en la filosofía: las pruebas también usan el patrón de errores de Go. Cuando pruebas una función que regresa error, verificas tanto el caso feliz como el caso de error. Una prueba que solo verifica que "no hubo error" es débil; una buena prueba verifica que el resultado sea el correcto Y que los errores ocurran cuando deben ocurrir.

Table-driven tests: el estilo idiomático de Go

Si tienes que probar una función con muchos casos (entradas válidas, inválidas, bordes), escribir una función Test por cada caso es repetitivo. La comunidad de Go usa un patrón llamado table-driven tests (pruebas basadas en tablas): defines una lista de casos como datos y los recorres en un ciclo, ejecutando la misma lógica de verificación para cada uno. Es el estilo que verás en la biblioteca estándar y en casi todo proyecto Go serio.

```
package mates

import "testing"

func TestSumaTabla(t *testing.T) {
    casos := []struct {
        nombre  string
        a, b    int
        esperado int
    }{
        {"positivos", 2, 3, 5},
        {"con cero", 0, 5, 5},
        {"negativos", -2, -3, -5},
        {"mixto", -2, 5, 3},
        {"ceros", 0, 0, 0},
    }

    for _, c := range casos {
        t.Run(c.nombre, func(t *testing.T) {
            resultado := Suma(c.a, c.b)
            if resultado != c.esperado {
                t.Errorf("Suma(%d, %d) = %d; se esperaba %d",
                    c.a, c.b, resultado, c.esperado)
            }
        })
    }
}
```


Cada caso es un struct anónimo con nombre, entradas y salida esperada. El ciclo recorre los casos y `t.Run`` crea una subprueba con nombre por cada caso. Con `go test -v`` verás cada subprueba por separado, y si una falla sabes exactamente cuál caso fue:

```
=== RUN    TestSumaTabla
=== RUN    TestSumaTabla/positivos
=== RUN    TestSumaTabla/con_cero
...
```

Para funciones que regresan error, la tabla incluye el error esperado:

```

func TestDividirTabla(t *testing.T) {
    casos := []struct {
        nombre      string
        a, b         float64
        esperado     float64
        esperaError bool
    }{
        {"division normal", 10, 2, 5, false},
        {"entre cero", 10, 0, 0, true},
        {"cero entre numero", 0, 5, 0, false},
        {"negativos", -10, -2, 5, false},
        {"decimales", 7.5, 2.5, 3, false},
    }

    for _, c := range casos {
        t.Run(c.nombre, func(t *testing.T) {
            resultado, err := Dividir(c.a, c.b)
            if c.esperaError {
                if err == nil {
                    t.Errorf("Dividir(%v, %v): se esperaba error, se obtuvo nil",
c.a, c.b)
                }
                return
            }
            if err != nil {
                t.Fatalf("Dividir(%v, %v): error inesperado: %v", c.a, c.b, err)
            }
            if resultado != c.esperado {
                t.Errorf("Dividir(%v, %v) = %v; se esperaba %v",
c.a, c.b, resultado, c.esperado)
            }
        })
    }
}

```

Este patrón escala muy bien: agregar un caso nuevo es agregar una línea a la tabla. Y cuando la función cambia de comportamiento, actualizas la tabla, no diez funciones separadas. Acostúmbrate a escribir tus pruebas así desde el principio: es una de las señales que distinguen a un programador Go experimentado.

Benchmarks: medir el rendimiento

Además de verificar que el código es correcto, a veces necesitas saber qué tan rápido es. Go incluye benchmarks en el mismo framework de testing. Una función benchmark tiene esta firma:

```
func BenchmarkNombre(b *testing.B) {
    for i := 0; i < b.N; i++ {
        // código a medir
    }
}
```

El framework ajusta b.N automáticamente hasta obtener una medición estable, y reporta el tiempo por operación y las operaciones por segundo. Se ejecutan con:

```
go test -bench=.
```

El punto es una expresión regular que selecciona qué benchmarks correr (`.` = todos). También puedes combinarlo: `go test -bench=. -benchmem` agrega estadísticas de memoria (bytes por operación y alojamientos por operación), muy útil para detectar código que genera basura innecesaria.

Ejemplo:

```
package mates

import "testing"

func BenchmarkSuma(b *testing.B) {
    for i := 0; i < b.N; i++ {
        Suma(2, 3)
    }
}
```

Salida típica:

```
BenchmarkSuma-8      10000000000    0.25 ns/op
```

El `-8` indica el número de CPUs usados. Los benchmarks son la forma correcta de responder preguntas como "¿es más rápido concatenar strings con `+` o con `strings.Builder`?" o "¿esta optimización realmente ayuda?": en lugar de adivinar, mides. Y ojo con una trampa común: el

compilador de Go puede eliminar código cuyo resultado no se usa, así que en benchmarks reales asegúrate de que el resultado se utilice (por ejemplo, asignándolo a una variable global) o la medición no significará nada.

Cobertura: go test -cover

La cobertura de código mide qué porcentaje de tu código ejecutan las pruebas. No es una garantía de calidad (puedes tener 100% de cobertura con pruebas malas), pero es una señal útil de qué partes del código no están probadas:

```
go test -cover
```

Imprime algo como ``coverage: 85.3% of statements``. Para un reporte detallado en HTML:

```
go test -coverprofile=coverage.out
go tool cover -html=coverage.out
```

Esto abre una página donde las líneas cubiertas aparecen en verde y las no cubiertas en rojo. Es una herramienta excelente para descubrir ramas de código olvidadas: ese ``if err != nil`` que nunca se prueba, ese caso borde que nadie consideró.

Una meta razonable para código nuevo es apuntar a una cobertura alta en la lógica de negocio (80-90%), sin obsesionarse con el 100%: hay código (como ciertas ramas defensivas o código de inicialización) donde forzar la cobertura cuesta más de lo que aporta. Lo importante es que la cobertura te sirva como mapa de lo no probado, no como una medalla.

El idioma comma-ok: cuando no necesitas un error

No todo lo que puede salir mal merece un error. Go tiene un idioma más ligero para los casos donde "no se encontró" o "no se pudo" es un resultado normal y no una falla: el comma-ok. Lo viste con los mapas:

```
valor, ok := mapa["clave"]
if !ok {
    // la clave no existe: no es un error, es un resultado posible
}
```

El segundo valor booleano dice si la operación tuvo éxito. Este patrón aparece en varios lugares de la biblioteca estándar: las aserciones de tipo (``v, ok := x.(string)``), la recepción de canales (``v, ok := <-ch``, donde ok es false si el canal está cerrado), y los type switches.

¿Cuándo usar comma-ok y cuándo usar error? La regla práctica: usa comma-ok cuando la condición de fallo es una sola, esperada y sin matices (existe o no existe, es de este tipo o no). Usa error cuando puede fallar de varias formas distintas, cuando necesitas explicar por qué falló, o cuando el fallo debe propagarse por varias capas. Un mapa que no tiene la clave no necesita un mensaje de error; una lectura de base de datos que falla sí, porque puede fallar por diez razones distintas y quien la llama necesita saber cuál.

Hay un tercer caso: funciones que regresan (valor, bool) inventadas por ti. Es válido para búsquedas simples en tus propios tipos, pero si la función puede fallar de formas que el llamador deba distinguir, prefiere error. Cuando dudas, error es la opción segura: siempre puedes ignorar un error que no te importe (conscientemente), pero no puedes sacarle información a un bool que no la tiene.

Diseñar APIs que regresan errores: contrato y documentación

Cuando escribes un paquete que otros van a usar, tus errores son parte del contrato de tu API, igual que los nombres de tus funciones. Algunas reglas de diseño:

1. Documenta qué errores puede regresar cada función. Un comentario como `"/ Regresa ErrNoEncontrado si el usuario no existe"` le dice al usuario qué debe manejar. Sin documentación, el usuario tiene que adivinar o leer tu código fuente.
2. Expón errores centinela para las condiciones que el usuario necesite distinguir programáticamente (no encontrado, ya existe, sin permiso, timeout). No expongas un centinela por cada cosa: solo para las decisiones reales que el usuario tomará.
3. No cambies el "tipo" de los errores entre versiones sin avisar: si tu función regresaba `*ErrorValidacion` y de pronto regresa un error genérico, rompes el código de quien usaba `errors.As`. Los errores son API pública.
4. Envuelve los errores de tus dependencias, no los expongas tal cual. Si tu paquete usa internamente una base de datos y regresas sus errores crudos, atas a tus usuarios a esa base de datos: el día que la cambies, sus verificaciones con `errors.Is` se rompen. Envuelve con contexto propio y expón tus propios centinelas.
5. Prefiere regresar errores de tus propios tipos antes que `errors.New` sueltos cuando el usuario pueda necesitar datos: un error con un campo `ReintentosSugeridos` vale más que un string.

Un ejemplo de API bien diseñada en miniatura:

```

// Package tienda ofrece un catalogo de productos.
package tienda

import (
    "errors"
    "fmt"
)

// ErrNoEncontrado se regresa cuando el producto no existe en el catalogo.
var ErrNoEncontrado = errors.New("tienda: producto no encontrado")

// ErrSinStock se regresa cuando no hay unidades suficientes.
var ErrSinStock = errors.New("tienda: sin stock suficiente")

type Catalogo struct {
    stock map[string]int
}

func Nuevo() *Catalogo {
    return &Catalogo{stock: map[string]int{}}
}

// Apartar reserva unidades de un producto.
// Regresa ErrNoEncontrado si el producto no existe,
// o ErrSinStock si no hay unidades suficientes.
func (c *Catalogo) Apartar(producto string, unidades int) error {
    disponibles, ok := c.stock[producto]
    if !ok {
        return fmt.Errorf("apartar %q: %w", producto, ErrNoEncontrado)
    }
    if unidades > disponibles {
        return fmt.Errorf("apartar %q: piden %d, hay %d: %w",
            producto, unidades, disponibles, ErrSinStock)
    }
    c.stock[producto] -= unidades
    return nil
}

```

Quien usa este paquete sabe exactamente qué puede salir mal y cómo reaccionar, sin leer una línea de la implementación. Esa es la diferencia entre una API profesional y una que solo "funciona".

Qué aprender de los errores de la biblioteca estándar

La biblioteca estándar de Go es la mejor escuela de manejo de errores porque la escribieron las mismas personas que diseñaron el lenguaje. Algunos patrones que vale la pena imitar:

- `io.EOF` no es un "error" en el sentido de que algo salió mal: es la señal de "se acabó la entrada". Muchas funciones regresan `io.EOF` como forma normal de decir "ya no hay más datos". La lección: un error puede ser un valor de control legítimo, no solo una falla. Cuando diseñes tus APIs, considera si "fin de los datos" merece un error centinela propio en lugar de un `bool`.
- `os.PathError` envuelve casi todos los errores de archivos con la operación y la ruta (lo vimos con `errors.As`). La lección: los errores de bajo nivel deben llevar el contexto de qué se intentaba hacer y sobre qué. Un "permiso denegado" sin decir qué archivo es mucho menos útil que un "open /etc/app/config.yaml: permiso denegado".
- `database/sql.ErrNoRows` es el centinela para "la consulta no trajo filas". La lección: las condiciones de "vacío" que el usuario debe distinguir merecen su propio error documentado, en lugar de obligar al usuario a interpretar un resultado vacío.
- Los paquetes bien hechos nunca hacen panic por entradas inválidas del usuario: regresan error. El panic queda reservado para violaciones de contrato del programador. Si estás escribiendo una biblioteca y dudas entre panic y error ante un argumento malo, elige error casi siempre; el panic solo cuando continuar sería incorrecto o peligroso.

Errores y concurrencia: no pierdas los errores de las goroutines

Hay un error clásico y peligroso cuando se mezcla el manejo de errores con goroutines: lanzar una goroutine que puede fallar y no tener forma de enterarse. Una goroutine no puede "regresar" un error a quien la lanzó como lo hace una función normal; si la goroutine falla en silencio, tu programa cree que todo salió bien.

```
// MAL: si procesar falla, nadie se entera.  
go procesar(archivo)
```

La solución idiomática es que la goroutine envíe su error por un canal, o que lo guarde en una variable protegida. El patrón más común con varias goroutines es un canal de errores con buffer:

```

package main

import (
    "errors"
    "fmt"
    "sync"
)

func procesar(id int) error {
    if id == 2 {
        return errors.New("fallo simulado")
    }
    fmt.Printf("Tarea %d ok\n", id)
    return nil
}

func main() {
    var wg sync.WaitGroup
    errores := make(chan error, 5) // buffer para no bloquear a las goroutines

    for id := 1; id <= 5; id++ {
        wg.Add(1)
        go func(id int) {
            defer wg.Done()
            if err := procesar(id); err != nil {
                errores <- fmt.Errorf("tarea %d: %w", id, err)
            }
        }(id)
    }

    wg.Wait()
    close(errores)

    huboErrores := false
    for err := range errores {
        fmt.Println("Error recolectado:", err)
        huboErrores = true
    }
    if huboErrores {
        fmt.Println("El lote termino con errores")
    } else {
        fmt.Println("Todo el lote salio bien")
    }
}

```


El canal con buffer del tamaño del número de tareas garantiza que ninguna goroutine se bloquee enviando su error aunque nadie esté leyendo todavía. Después de `wg.Wait()` se cierra el canal y se recolectan todos los errores. Cada error se envuelve con el id de la tarea para saber cuál falló.

Variantes de este patrón: si solo te importa el primer error, puedes usar un canal sin buffer de capacidad 1 y un select con default para no bloquear; si usas `errgroup` (del paquete golang.org/x/sync), el grupo recolecta el primer error por ti y cancela el resto vía contexto. Pero el principio es el mismo y no cambia nunca: todo error que ocurra en una goroutine debe tener un camino de regreso. Un programa concurrente que pierde errores es un programa que miente sobre su propio estado, y eso en producción se paga caro.

Ejemplos paso a paso

Ejemplo 1: Función que valida y regresa errores descriptivos

Problema: escribir una función que registre usuarios con validaciones, regresando errores claros para cada problema.

```

package main

import (
    "errors"
    "fmt"
    "strings"
)

func registrarUsuario(nombre, correo string, edad int) error {
    if strings.TrimSpace(nombre) == "" {
        return errors.New("el nombre no puede estar vacio")
    }
    if !strings.Contains(correo, "@") {
        return fmt.Errorf("correo invalido: %q", correo)
    }
    if edad < 18 {
        return fmt.Errorf("edad insuficiente: %d (minimo 18)", edad)
    }
    fmt.Printf("Usuario %s registrado con exito\n", nombre)
    return nil
}

func main() {
    casos := []struct {
        nombre, correo string
        edad            int
    }{
        {"Ana", "ana@correo.com", 25},
        {"", "bob@correo.com", 30},
        {"Luis", "luis-sin-arroba", 20},
        {"Maria", "maria@correo.com", 16},
    }
    for _, c := range casos {
        err := registrarUsuario(c.nombre, c.correo, c.edad)
        if err != nil {
            fmt.Printf("Fallo registro (%s): %v\n", c.nombre, err)
        }
    }
}

```

Explicación: cada validación regresa de inmediato con un error descriptivo (early return). El llamador revisa el error y decide qué hacer. Observa cómo `fmt.Errorf` permite incluir el valor problemático en el mensaje, lo que facilita el diagnóstico.

Ejemplo 2: Envolver errores con contexto a través de capas

Problema: simular tres capas (base de datos, servicio, manejador) donde el error se envuelve en cada nivel con contexto adicional.

```
package main

import (
    "errors"
    "fmt"
)

var ErrRegistroNoEncontrado = errors.New("registro no encontrado")

// Capa de datos: simula una base de datos.
func buscarEnBD(id int) (string, error) {
    if id != 42 {
        return "", fmt.Errorf("bd: id %d: %w", id, ErrRegistroNoEncontrado)
    }
    return "datos del 42", nil
}

// Capa de servicio: agrega contexto de negocio.
func obtenerReporte(id int) (string, error) {
    datos, err := buscarEnBD(id)
    if err != nil {
        return "", fmt.Errorf("servicio: no se pudo generar el reporte: %w", err)
    }
    return "reporte: " + datos, nil
}

func main() {
    _, err := obtenerReporte(7)
    if err != nil {
        fmt.Println("Error completo:", err)
        if errors.Is(err, ErrRegistroNoEncontrado) {
            fmt.Println("-> Se detecto que la causa fue: registro no encontrado")
        }
    }
}
```

Explicación: cada capa envuelve con %w agregando su contexto. El mensaje final cuenta la historia completa: `servicio: no se pudo generar el reporte: bd: id 7: registro no encontrado`. Y gracias a la envoltura, errors.Is puede identificar la causa raíz aunque haya dos capas encima. Si hubiéramos usado %v en lugar de %w, la identificación sería imposible.

Ejemplo 3: Error personalizado con errors.As

Problema: un validador de formularios que regresa errores estructurados con código y campo, y un manejador que reacciona distinto según el tipo.

```

package main

import (
    "errors"
    "fmt"
)

type ErrorCampo struct {
    Campo  string
   Codigo int
}

func (e *ErrorCampo) Error() string {
    return fmt.Sprintf("campo %q invalido (codigo %d)", e.Campo, e.Codigo)
}

func validarFormulario(nombre string, edad int) error {
    if nombre == "" {
        return &ErrorCampo{Campo: "nombre", Codigo: 101}
    }
    if edad < 0 || edad > 130 {
        return &ErrorCampo{Campo: "edad", Codigo: 102}
    }
    return nil
}

func main() {
    err := validarFormulario("", 25)
    if err != nil {
        var ec *ErrorCampo
        if errors.As(err, &ec) {
            fmt.Printf("Error de validacion: campo=%s codigo=%d\n", ec.Campo,
ec.Codigo)
            if ec.Codigo == 101 {
                fmt.Println("Sugerencia: el nombre es obligatorio")
            }
        } else {
            fmt.Println("Error desconocido:", err)
        }
    }
}

```

Explicación: ErrorCampo implementa la interfaz error con un método Error() sobre el apuntador. El manejador usa errors.As para extraer el tipo concreto y acceder a sus campos,

permitiendo lógica específica por código de error. Este es el patrón que usan las APIs bien diseñadas para que sus usuarios puedan reaccionar de forma programática.

Ejemplo 4: panic y recover en un procesador por lotes

Problema: procesar una lista de tareas donde una puede causar panic por un bug; el procesador debe recuperarse, registrar el problema y continuar con las demás.

```
package main

import "fmt"

func procesarTarea(id int) {
    defer func() {
        if r := recover(); r != nil {
            fmt.Printf("Tarea %d fallo con panic (recuperado): %v\n", id, r)
        }
    }()

    fmt.Printf("Procesando tarea %d...\n", id)
    if id == 3 {
        panic("bug simulado: indice fuera de rango")
    }
    fmt.Printf("Tarea %d completada\n", id)
}

func main() {
    for id := 1; id <= 5; id++ {
        procesarTarea(id)
    }
    fmt.Println("Lote terminado: el programa sobrevivio al panic")
}
```

Explicación: cada tarea tiene su propio defer con recover, que actúa como cortafuegos: el panic de la tarea 3 se detiene ahí, se registra, y el ciclo continúa con las tareas 4 y 5. Sin el recover, el panic habría terminado todo el programa en la tarea 3. Este es exactamente el patrón que se usa en servidores para aislar fallos por petición.

Ejemplo 5: Table-driven test completo con subpruebas

Problema: probar una función que clasifica temperaturas, cubriendo casos normales, bordes y errores.

```

package main

import (
    "errors"
    "fmt"
)

func clasificarTemp(celsius float64) (string, error) {
    if celsius < -273.15 {
        return "", errors.New("temperatura bajo el cero absoluto")
    }
    switch {
    case celsius < 0:
        return "congelado", nil
    case celsius < 20:
        return "frio", nil
    case celsius < 30:
        return "templado", nil
    default:
        return "caliente", nil
    }
}

```

Y su prueba en `main\_test.go` (mismo paquete main para este ejemplo):

```

package main

import "testing"

func TestClasificarTemp(t *testing.T) {
    casos := []struct {
        nombre      string
        entrada      float64
        esperado     string
        esperaError bool
    }{
        {"bajo cero absoluto", -300, "", true},
        {"cero absoluto exacto", -273.15, "congelado", false},
        {"congelado", -10, "congelado", false},
        {"frio limite inferior", 0, "frio", false},
        {"frio", 10, "frio", false},
        {"templado limite", 20, "templado", false},
        {"templado", 25, "templado", false},
        {"caliente limite", 30, "caliente", false},
        {"caliente", 45, "caliente", false},
    }
}

```

```

    }

    for _, c := range casos {
        t.Run(c.nombre, func(t *testing.T) {
            resultado, err := clasificarTemp(c.entrada)
            if c.esperaError {
                if err == nil {
                    t.Errorf("clasificarTemp(%v): se esperaba error", c.entrada)
                }
                return
            }
            if err != nil {
                t.Fatalf("clasificarTemp(%v): error inesperado: %v", c.entrada, err)
            }
            if resultado != c.esperado {
                t.Errorf("clasificarTemp(%v) = %q; se esperaba %q",
                    c.entrada, resultado, c.esperado)
            }
        })
    }
}

func ExampleclasificarTemp() {
    r, _ := clasificarTemp(25)
    fmt.Println(r)
    // Output: templado
}

```

Explicación: la tabla cubre el error, los límites exactos de cada rango (0, 20, 30) y valores intermedios. Los casos de borde son los que más bugs revelan: ¿qué pasa justo en el límite? Además incluimos una Example function (func Example... con comentario // Output:), que Go ejecuta como prueba y también usa como documentación. Si la salida no coincide, la prueba falla.

Ejemplo 6: Benchmark comparando dos formas de concatenar

Problema: decidir si concatenar strings con += en un ciclo o con strings.Builder es más rápido. En lugar de adivinar, medimos.


```

package main

import (
    "strings"
    "testing"
)

func concatMas(n int) string {
    s := ""
    for i := 0; i < n; i++ {
        s += "x"
    }
    return s
}

func concatBuilder(n int) string {
    var b strings.Builder
    for i := 0; i < n; i++ {
        b.WriteString("x")
    }
    return b.String()
}

func BenchmarkConcatMas(b *testing.B) {
    for i := 0; i < b.N; i++ {
        concatMas(1000)
    }
}

func BenchmarkConcatBuilder(b *testing.B) {
    for i := 0; i < b.N; i++ {
        concatBuilder(1000)
    }
}

```

Con `go test -bench=. -benchmem` obtendrías algo como:

BenchmarkConcatMas-8	5000	250000 ns/op	5000000 B/op	1000
allocs/op				
BenchmarkConcatBuilder-8	500000	2500 ns/op	2048 B/op	2
allocs/op				

Explicación: strings.Builder es órdenes de magnitud más rápido y hace muchas menos alojamientos de memoria, porque += crea un string nuevo en cada iteración (los strings en Go

son inmutables). Los benchmarks convierten una discusión de opiniones en datos. Nota las columnas B/op y allocs/op con `-benchmem`: te dicen cuánta memoria usa cada operación, información clave para código que corre millones de veces.

Ejemplo 7: Reintentos con espera exponencial

Problema: una operación de red falla de forma intermitente; se quiere reintentarla hasta 3 veces con esperas crecientes (1s, 2s, 4s), envolviendo el último error con contexto si todos los intentos fallan.

```

package main

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

// operacionInestable simula una llamada de red que falla a veces.
func operacionInestable() error {
    if rand.Intn(3) == 0 { // 1 de cada 3 veces tiene exito
        return nil
    }
    return errors.New("timeout de red")
}

func conReintentos(intentos int, operacion func() error) error {
    var err error
    espera := time.Second
    for i := 1; i <= intentos; i++ {
        err = operacion()
        if err == nil {
            return nil // exito: no hay nada mas que hacer
        }
        fmt.Printf("Intento %d fallo: %v. Reintentando en %v...\n", i, err, espera)
        time.Sleep(espera)
        espera *= 2 // espera exponencial: 1s, 2s, 4s...
    }
    return fmt.Errorf("fallo despues de %d intentos: %w", intentos, err)
}

func main() {
    rand.Seed(time.Now().UnixNano())
    if err := conReintentos(3, operacionInestable); err != nil {
        fmt.Println("Error final:", err)
        return
    }
    fmt.Println("Operacion exitosa")
}

```

Explicación: el ciclo intenta la operación y sale de inmediato con nil si tiene éxito (early return también aplica aquí). Si falla, espera un tiempo que se duplica en cada intento: la espera exponencial evita saturar al servidor caído con reintentos frenéticos, que es lo que haría un

ciclo sin espera. Al final, si todo falló, se envuelve el último error con %w agregando cuántos intentos se hicieron. Este patrón (reintentos + backoff + error envuelto) aparece en casi todo cliente de red profesional. Nota: en código real de producción usarías context.Context para poder cancelar los reintentos, tema de la Unidad 8 que aquí simplificamos con un número fijo de intentos.

Práctica

Ejercicio 1: Validador de contraseñas con errores descriptivos

Escribe una función `validarPassword(pw string) error` que verifique: al menos 8 caracteres, al menos un dígito, al menos una mayúscula. Debe regresar un error descriptivo específico para la primera regla que falle, o nil si la contraseña es válida. Luego escribe una table-driven test con al menos 6 casos.

Solución explicada:

```

package main

import (
    "errors"
    "fmt"
    "unicode"
)

func validarPassword(pw string) error {
    if len(pw) < 8 {
        return fmt.Errorf("password muy corto: %d caracteres (minimo 8)", len(pw))
    }
    var tieneDigito, tieneMayus bool
    for _, r := range pw {
        if unicode.IsDigit(r) {
            tieneDigito = true
        }
        if unicode.IsUpper(r) {
            tieneMayus = true
        }
    }
    if !tieneDigito {
        return errors.New("el password debe contener al menos un digito")
    }
    if !tieneMayus {
        return errors.New("el password debe contener al menos una mayuscula")
    }
    return nil
}

```

La prueba:

```

func TestValidarPassword(t *testing.T) {
    casos := []struct {
        nombre      string
        pw           string
        esperaError bool
    }{
        {"valido", "Secreto123", false},
        {"muy corto", "Ab1", true},
        {"sin digito", "Secretooo", true},
        {"sin mayuscula", "secreto123", true},
        {"vacio", "", true},
        {"exacto 8 valido", "Abcdefg1", false},
    }
    for _, c := range casos {
        t.Run(c.nombre, func(t *testing.T) {
            err := validarPassword(c.pw)
            if c.esperaError && err == nil {
                t.Errorf("validarPassword(%q): se esperaba error", c.pw)
            }
            if !c.esperaError && err != nil {
                t.Errorf("validarPassword(%q): error inesperado: %v", c.pw, err)
            }
        })
    }
}

```

Puntos clave: se valida en orden y se regresa al primer fallo (el usuario corrige una cosa a la vez). El mensaje del error de longitud incluye el dato concreto (cuántos caracteres hay), siguiendo la buena práctica de errores descriptivos. La tabla incluye el caso borde de exactamente 8 caracteres.

Ejercicio 2: Error centinela y errors.Is

Define un error centinela `ErrSaldoInsuficiente` en un paquete de cuentas bancarias. La función `Retirar(cuenta \*Cuenta, monto float64) error` debe envolver ese centinela con contexto usando `%w` cuando el saldo no alcance. En main, haz un retiro que falle y usa `errors.Is` para detectar el centinela y mostrar un mensaje amable al usuario.

Solución explicada:

```

package main

import (
    "errors"
    "fmt"
)

var ErrSaldoInsuficiente = errors.New("saldo insuficiente")

type Cuenta struct {
    Titular string
    Saldo    float64
}

func Retirar(c *Cuenta, monto float64) error {
    if monto <= 0 {
        return fmt.Errorf("monto invalido: %.2f", monto)
    }
    if monto > c.Saldo {
        return fmt.Errorf("retiro de %.2f con saldo %.2f: %w",
            monto, c.Saldo, ErrSaldoInsuficiente)
    }
    c.Saldo -= monto
    return nil
}

func main() {
    c := &Cuenta{Titular: "Ana", Saldo: 500}
    err := Retirar(c, 800)
    if err != nil {
        if errors.Is(err, ErrSaldoInsuficiente) {
            fmt.Println("No te alcanza, carnal: tienes", c.Saldo)
        } else {
            fmt.Println("Error:", err)
        }
    }
}

```

Puntos clave: el centinela se define una vez con `errors.New` y se documenta como parte de la API. La función lo envuelve con `%w` agregando montos concretos. El llamador no necesita saber cómo se envolvió: `errors.Is` lo encuentra en la cadena. Así puedes cambiar el mensaje de contexto mañana sin romper el código de quien usa tu paquete.

Ejercicio 3: Error personalizado para una API de inventario

Crea un tipo `ErrorStock` con campos `ProductoID int` y `Disponible int`, que implemente error. La función `apartarProducto(id, cantidad int, stock map[int]int) error` regresa `*ErrorStock` cuando no hay suficiente inventario. En el manejador, usa `errors.As` para extraer el error y mostrar cuántas unidades hay disponibles.

Solución explicada:


```

package main

import (
    "errors"
    "fmt"
)

type ErrorStock struct {
    ProductoID int
    Disponible int
    Solicitado int
}

func (e *ErrorStock) Error() string {
    return fmt.Sprintf("stock insuficiente para producto %d: hay %d, se pidieron %d",
        e.ProductoID, e.Disponible, e.Solicitado)
}

func apartarProducto(id, cantidad int, stock map[int]int) error {
    disponible := stock[id] // cero si no existe
    if cantidad > disponible {
        return &ErrorStock{ProductoID: id, Disponible: disponible, Solicitado:
cantidad}
    }
    stock[id] -= cantidad
    return nil
}

func main() {
    stock := map[int]int{1: 10, 2: 3}
    err := apartarProducto(2, 5, stock)
    if err != nil {
        var es *ErrorStock
        if errors.As(err, &es) {
            fmt.Printf("Solo quedan %d unidades del producto %d\n",
                es.Disponible, es.ProductoID)
        } else {
            fmt.Println("Error:", err)
        }
    }
}

```

Puntos clave: el error lleva datos estructurados (no solo texto), así el manejador puede tomar decisiones programáticas: mostrar el disponible, sugerir una cantidad menor, registrar el producto con problema. El método `Error()` produce un mensaje legible para humanos, y `errors.As` da acceso a la máquina. Las dos caras del mismo error.

Ejercicio 4: Middleware de recover para tareas en background

Escribe una función `ejecutarSeguro(nombre string, tarea func())` que ejecute la tarea dentro de un defer con recover: si hay panic, debe imprimir "TAREA <nombre> RECUPERADA" con el valor del panic y un mensaje de que el sistema continúa. Pruébala con tres tareas, una de las cuales hace panic.`

Solución explicada:

```
package main

import "fmt"

func ejecutarSeguro(nombre string, tarea func()) {
    defer func() {
        if r := recover(); r != nil {
            fmt.Printf("TAREA %s RECUPERADA del panic: %v\n", nombre, r)
            fmt.Println("El sistema continua operando")
        }
    }()
    tarea()
    fmt.Printf("Tarea %s termino sin problemas\n", nombre)
}

func main() {
    ejecutarSeguro("respaldo", func() {
        fmt.Println("Haciendo respaldo...")
    })
    ejecutarSeguro("reporte", func() {
        panic("formato de datos corrupto")
    })
    ejecutarSeguro("limpieza", func() {
        fmt.Println("Limpiando temporales...")
    })
}
```

Puntos clave: el `recover` está en el límite de cada tarea, no en `main`: así el fallo de una no afecta a las demás. La función diferida es anónima y se registra antes de ejecutar la tarea. En

un sistema real, aquí también escribirías el panic en un log con el stack trace (debug.Stack()) para investigar después. Nunca uses este patrón para ocultar errores esperados: es solo para bugs inesperados en unidades aisladas de trabajo.

Ejercicio 5: Benchmark y cobertura de una función de búsqueda

Escribe una función `contiene(numeros []int, objetivo int) bool` que busque un número en un slice. Escribe table-driven tests que cubran: elemento al inicio, en medio, al final, no existente, slice vacío. Corre `go test -cover` y verifica que la cobertura sea del 100%. Luego escribe un benchmark que la mida con un slice de 10,000 elementos.

Solución explicada:

```
package main

func contiene(numeros []int, objetivo int) bool {
    for _, n := range numeros {
        if n == objetivo {
            return true
        }
    }
    return false
}
```

La prueba:

```

func TestContiene(t *testing.T) {
    casos := []struct {
        nombre    string
        numeros    []int
        objetivo   int
        esperado   bool
    }{
        {"al inicio", []int{1, 2, 3}, 1, true},
        {"en medio", []int{1, 2, 3}, 2, true},
        {"al final", []int{1, 2, 3}, 3, true},
        {"no existe", []int{1, 2, 3}, 9, false},
        {"slice vacio", []int{}, 1, false},
        {"slice nil", nil, 1, false},
    }
    for _, c := range casos {
        t.Run(c.nombre, func(t *testing.T) {
            if got := contiene(c.numeros, c.objetivo); got != c.esperado {
                t.Errorf("contiene(%v, %d) = %v; se esperaba %v",
                    c.numeros, c.objetivo, got, c.esperado)
            }
        })
    }
}

func BenchmarkContiene(b *testing.B) {
    numeros := make([]int, 10000)
    for i := range numeros {
        numeros[i] = i
    }
    b.ResetTimer() // no contar la preparacion
    for i := 0; i < b.N; i++ {
        contiene(numeros, 9999) // peor caso: esta al final
    }
}

```

Puntos clave: los casos cubren todas las ramas (encontrado en cada posición, no encontrado, vacío, nil), por eso la cobertura llega al 100%. En el benchmark, `b.ResetTimer()` excluye el tiempo de preparación de la medición, y se busca el último elemento para medir el peor caso. Con `-benchmem` verías cero alojamientos: la función no genera basura, buena señal.

Ejercicio 6: Cadena de envoltura de tres niveles con errors.Is y errors.As

Simula un programa que lee un archivo de configuración, lo interpreta y lo valida. Define un error centinela `ErrConfigInvalida` y un tipo personalizado `ErrorLinea{Linea int}` para errores de sintaxis. La lectura puede fallar con error de archivo, la interpretación puede fallar con `ErrorLinea`, la validación con el centinela. Envuelve en cada nivel con `%w`. En main provoca cada tipo de fallo y demuestra que `errors.Is` detecta el centinela y `errors.As` extrae la línea del error de sintaxis.

Solución explicada:

```
package main

import (
    "errors"
    "fmt"
    "os"
    "strings"
)

var ErrConfigInvalida = errors.New("configuracion invalida")

type ErrorLinea struct {
    Linea int
    Texto string
}

func (e *ErrorLinea) Error() string {
    return fmt.Sprintf("error de sintaxis en linea %d: %q", e.Linea, e.Texto)
}

func leerArchivo(ruta string) (string, error) {
    datos, err := os.ReadFile(ruta)
    if err != nil {
        return "", fmt.Errorf("leer configuracion: %w", err)
    }
    return string(datos), nil
}

func interpretar(contenido string) (map[string]string, error) {
    cfg := map[string]string{}
    for i, linea := range strings.Split(contenido, "\n") {
        linea = strings.TrimSpace(linea)
        if linea == "" {
            continue
        }
    }
}
```

```

    }
    partes := strings.SplitN(linea, "=", 2)
    if len(partes) != 2 {
        return nil, fmt.Errorf("interpretar: %w",
            &ErrorLinea{Linea: i + 1, Texto: linea})
    }
    cfg[strings.TrimSpace(partes[0])] = strings.TrimSpace(partes[1])
}
return cfg, nil
}

func validar(cfg map[string]string) error {
    if cfg["puerto"] == "" {
        return fmt.Errorf("validar: falta el puerto: %w", ErrConfigInvalida)
    }
    return nil
}

func cargarConfig(ruta string) error {
    contenido, err := leerArchivo(ruta)
    if err != nil {
        return fmt.Errorf("cargarConfig: %w", err)
    }
    cfg, err := interpretar(contenido)
    if err != nil {
        return fmt.Errorf("cargarConfig: %w", err)
    }
    if err := validar(cfg); err != nil {
        return fmt.Errorf("cargarConfig: %w", err)
    }
    fmt.Println("Configuracion valida:", cfg)
    return nil
}

func manejarError(err error) {
    var el *ErrorLinea
    switch {
    case errors.As(err, &el):
        fmt.Printf("Hay un error de sintaxis en la linea %d: %q\n", el.Linea,
            el.Texto)
    case errors.Is(err, ErrConfigInvalida):
        fmt.Println("La configuracion esta incompleta:", err)
    case errors.Is(err, os.ErrNotExist):
        fmt.Println("No existe el archivo de configuracion, usando valores por
defecto")
    }
}

```

```

    default:
        fmt.Println("Error inesperado:", err)
    }
}

func main() {
    // Caso 1: archivo inexistente
    manejarError(cargarConfig("no-existe.conf"))

    // Caso 2: archivo con error de sintaxis
    os.WriteFile("mala.conf", []byte("puerto=8080\nlinea rota sin igual\n"), 0644)
    manejarError(cargarConfig("mala.conf"))

    // Caso 3: archivo sin puerto
    os.WriteFile("incompleta.conf", []byte("host=localhost\n"), 0644)
    manejarError(cargarConfig("incompleta.conf"))

    os.Remove("mala.conf")
    os.Remove("incompleta.conf")
}

```

Puntos clave: este ejercicio junta todo lo de la unidad. Tres niveles de envoltura con %w, un centinela, un error personalizado, y un manejador que usa errors.As primero (para el tipo concreto) y errors.Is después (para los centinelas). El orden del switch importa: As antes que Is cuando el tipo concreto da más información. Este es exactamente el diseño de manejo de errores que encontrarás en servidores Go profesionales.

Ejercicio 7: Hacer testeable la función de reintentos

La función conReintentos del Ejemplo 7 es difícil de probar: si falla, duerme 1+2+4 = 7 segundos, y los tests lentos nadie los corre. Refactorízala para recibir la función de espera como parámetro (inyección de dependencias), de modo que en el test puedas pasar una espera falsa que no duerma y que registre cuánto se habría esperado. Escribe la table-driven test que verifique: éxito al primer intento (cero esperas), éxito al tercer intento (esperas de 1s y 2s), y fallo total (3 intentos y error envuelto).

Solución explicada:

```

package main

import (
    "errors"
    "fmt"
    "testing"

```

```

    "time"
)

func conReintentos(intentos int, espera func(time.Duration), operacion func() error)
error {
    var err error
    pausa := time.Second
    for i := 1; i <= intentos; i++ {
        err = operacion()
        if err == nil {
            return nil
        }
        espera(pausa)
        pausa *= 2
    }
    return fmt.Errorf("fallo despues de %d intentos: %w", intentos, err)
}

func TestConReintentos(t *testing.T) {
    casos := []struct {
        nombre      string
        fallosAntesDeExito int // cuantos fallos antes de tener exito; -1 = siempre
falla
        esperasEsperadas []time.Duration
        esperaError      bool
    }{
        {"exito inmediato", 0, nil, false},
        {"exito al tercer intento", 2,
            []time.Duration{time.Second, 2 * time.Second}, false},
        {"fallo total", -1,
            []time.Duration{time.Second, 2 * time.Second, 4 * time.Second}, true},
    }

    for _, c := range casos {
        t.Run(c.nombre, func(t *testing.T) {
            var esperas []time.Duration
            esperaFalsa := func(d time.Duration) { esperas = append(esperas, d) }

            llamadas := 0
            operacion := func() error {
                llamadas++
                if c.fallosAntesDeExito >= 0 && llamadas > c.fallosAntesDeExito {
                    return nil
                }
                return errors.New("fallo")
            }

```



```

    }

    err := conReintentos(3, esperaFalsa, operacion)

    if c.esperaError && err == nil {
        t.Error("se esperaba error, se obtuvo nil")
    }
    if !c.esperaError && err != nil {
        t.Errorf("error inesperado: %v", err)
    }
    if len(esperas) != len(c.esperasEsperadas) {
        t.Fatalf("esperas = %v; se esperaban %v", esperas, c.esperasEsperadas)
    }
    for i := range esperas {
        if esperas[i] != c.esperasEsperadas[i] {
            t.Errorf("espera[%d] = %v; se esperaba %v",
                i, esperas[i], c.esperasEsperadas[i])
        }
    }
    })
}
}

```

Puntos clave: la lección grande de este ejercicio no es el test en sí, sino el diseño. Al recibir `espera` como parámetro, la función se vuelve testeable sin cambiar su comportamiento real (en producción pasas `time.Sleep`). Esta técnica, inyección de dependencias, es la que permite probar código que habla con la red, el disco o el reloj: en el test inyectas dobles falsos. El test corre en milisegundos y verifica tanto el número de reintentos como la progresión exacta de las esperas (1s, 2s, 4s). Código difícil de probar suele ser código mal diseñado; si te cuesta escribir el test, probablemente la función necesita que le inyectes sus dependencias.

Resumen de la unidad

En esta unidad aprendiste la filosofía de errores de Go de principio a fin. Los puntos esenciales que debes llevarte:

- En Go los errores son valores, no excepciones. Las funciones que pueden fallar regresan un error como último valor de retorno, y el llamador lo revisa con el patrón `if err != nil` inmediatamente después de la llamada.
- El patrón `if err != nil` con retorno temprano mantiene el camino feliz sin indentación y

hace visible exactamente dónde puede fallar cada operación. Nunca ignores errores con `_` sin una razón documentada, y nunca uses un valor cuando su error no es nil.

- Creas errores con `errors.New` para mensajes fijos y con `fmt.Errorf` para mensajes con valores. Los mensajes van en minúsculas, sin punto final, descriptivos y concisos.
- Envuelves errores con el verbo `%w` para agregar contexto conservando la cadena del error original. Nunca uses `%v` si quieres que la causa siga siendo identificable.
- `errors.Is` recorre la cadena buscando un error centinela específico; `errors.As` extrae el primer error de un tipo concreto para acceder a sus datos. Úsalos en ese orden de especificidad cuando manejes errores de varias capas.
- Puedes crear errores personalizados con structs que implementen el método `Error()` string, dándoles datos estructurados (códigos, campos, contexto) además del mensaje.
- `panic` es para errores de programador y estados imposibles, no para errores de operación esperados. `recover`, llamado dentro de un `defer`, detiene un `panic` en un límite definido (una petición, una tarea) para que el resto del programa sobreviva. Úsalo con moderación y registra siempre lo recuperado.
- El paquete `log` cubre el logging básico (`log.Println`, `log.Printf`, `log.Fatal`); `log.Fatal` termina el programa sin ejecutar `defers`, así que úsalo solo en inicialización.
- El testing viene integrado: archivos `_test.go`, funciones `TestXxx(t *testing.T)`, y los comandos `go test`, `go test -v`, `go test -run` y `go test ./...`
- El estilo idiomático son los table-driven tests: una tabla de casos con nombre, entradas y salidas esperadas, recorrida en un ciclo con `t.Run` por caso. Escala sin repetición y hace obvio qué caso falló.
- Los benchmarks (`func BenchmarkXxx(b *testing.B)`, `go test -bench`) miden rendimiento con datos en lugar de opiniones; `-benchmem` agrega memoria y alojamientos.
- `go test -cover` mide la cobertura: úsala como mapa de lo no probado, apuntando a cobertura alta en la lógica importante sin obsesionarte con el 100%.

Con errores bien manejados y pruebas que lo demuestran, tu código deja de ser "algo que funciona en tu máquina" y se convierte en software confiable. En la siguiente unidad llevaremos todo esto a la práctica construyendo servidores web reales: ahí verás cómo el manejo de errores y el testing se vuelven indispensables cuando tu programa atiende a usuarios de verdad por la red.

UNIDAD 10: Go en la práctica: servidores web y despliegue

Llegaste a la última unidad del libro, y es la que pone todo junto. Hasta ahora has aprendido el lenguaje pieza por pieza: tipos, funciones, structs, interfaces, errores, concurrencia, testing. Todo eso era el entrenamiento. Ahora viene el partido: construir programas que viven en la red, que atienden peticiones de usuarios reales, que se compilan en un solo binario y se despliegan en un servidor para correr día y noche.

No es casualidad que Go sea uno de los lenguajes favoritos para el backend moderno. Fue diseñado en Google para resolver exactamente este problema: programas de red que deben ser rápidos, que manejan miles de conexiones simultáneas sin complicarse la vida, y que se despliegan con un solo archivo ejecutable, sin intérpretes ni máquinas virtuales ni cientos de dependencias del sistema. Cuando termines esta unidad vas a entender por qué empresas como Google, Uber, Twitch, Dropbox y Cloudflare usan Go para su infraestructura crítica, y vas a tener las herramientas para construir ese tipo de software tú mismo.

El plan de la unidad: primero el paquete `net/http`, que es todo lo que necesitas para levantar un servidor web sin instalar nada. Aprenderás cómo funcionan las peticiones y respuestas, cómo registrar rutas (incluyendo la sintaxis moderna de Go 1.22 con métodos HTTP en el patrón), y cómo leer y escribir JSON, que es el idioma de las APIs modernas. Después construiremos una API REST completa y funcional: un CRUD de tareas en memoria con los cuatro verbos (GET, POST, PUT, DELETE), con manejo de errores decente y estructura limpia. Luego veremos plantillas HTML con `html/template` para cuando necesites servir páginas web de verdad, no solo datos.

La segunda mitad mira hacia producción: variables de entorno para configurar tu programa sin recompilarlo, compilación con `go build` (incluyendo binarios estáticos y compilación cruzada con GOOS y GOARCH), y un `Dockerfile` multietapa para empaquetar tu servidor en un contenedor listo para cualquier nube. Cerraremos con el mapa de a dónde seguir: frameworks como Gin y Echo, gRPC, bases de datos con `database/sql`, un proyecto final integrador paso a paso (una API de biblioteca), y consejos honestos para conseguir trabajo como desarrollador Go.

Explicación detallada

El paquete net/http: un servidor en diez líneas

La biblioteca estándar de Go incluye un servidor HTTP completo, listo para producción, en el paquete net/http. Sin frameworks, sin dependencias externas. El servidor más simple posible se ve así:

```
package main

import (
    "fmt"
    "net/http"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hola desde Go")
    })
    fmt.Println("Servidor en http://localhost:8080")
    http.ListenAndServe(":8080", nil)
}
```

Guárdalo, corre `go run main.go`, abre tu navegador en `http://localhost:8080` y verás el mensaje. Diez líneas y ya tienes un servidor web funcionando. Desmenucemos qué hace cada parte:

- `http.HandleFunc("/", ...)` registra un manejador (handler) para la ruta `/`. Un handler es una función que recibe dos cosas: un `http.ResponseWriter` (w), que es por donde escribes la respuesta, y un `*http.Request` (r), que trae toda la información de la petición (método, URL, encabezados, cuerpo).
- `http.ListenAndServe(":8080", nil)` abre el puerto 8080 y empieza a aceptar conexiones.

El nil significa "usa el manejador por defecto" (el `DefaultServeMux`, un enrutador global). Esta llamada bloquea: no regresa hasta que el servidor se detiene o falla. Por eso el `fmt.Println` va antes.

Cada petición que llega se atiende en su propia goroutine. Esto es clave y es gratis: no tienes que configurar hilos ni pools. Go crea una goroutine ligera por conexión y el runtime las multiplexa sobre los hilos del sistema. Un servidor net/http puede atender miles de conexiones concurrentes sin que tú escribas una sola línea de código de concurrencia. Es una de las razones por las que Go brilla en el backend.

Analogía: piensa en el servidor como un restaurante. ListenAndServe es abrir la puerta y colgar el letrero de "abierto". El mux (multiplexor) es el recepcionista que mira la petición ("mesa para la ruta /tarear") y la lleva al mesero correcto (el handler registrado para esa ruta). Cada mesa (petición) la atiende un mesero distinto al mismo tiempo (goroutines), sin que se estorben entre sí. Y el ResponseWriter es la charola donde el mesero pone la respuesta para llevarla de vuelta al cliente.

Request y Response: las dos mitades de la conversación

Toda comunicación HTTP es una conversación de dos mensajes: el cliente manda una petición (request) y el servidor regresa una respuesta (response). En tu handler tienes acceso completo a ambas.

Del `*http.Request` (`r`), lo que más vas a usar:

- `r.Method`: el verbo HTTP ("GET", "POST", "PUT", "DELETE"...).
- `r.URL.Path`: la ruta pedida ("/tarear", "/usuarios/42").
- `r.URL.Query()`: los parámetros de la URL (?nombre=ana&edad=25) como mapa.
- `r.Header`: los encabezados de la petición (Content-Type, Authorization...).
- `r.Body`: el cuerpo de la petición (los datos que manda el cliente en POST/PUT).
- `r.PathValue("id")`: el valor de un segmento de ruta con nombre (Go 1.22+, lo veremos abajo).

Con el `http.ResponseWriter` (`w`), construyes la respuesta:

- `w.Write([]byte("texto"))` o `w_fmt.Fprintln(w, ...)`: escribe el cuerpo.
- `w.Header().Set("Content-Type", "application/json")`: pone encabezados. Importante:

los encabezados deben ponerse ANTES de escribir el cuerpo.

- `w.WriteHeader(http.StatusCreated)`: pone el código de estado (200, 201, 404, 500...).

Si no lo llamas, Go manda 200 OK automáticamente al escribir el cuerpo. También debe ir antes de escribir el cuerpo.

El orden importa y es una fuente clásica de confusión: primero encabezados, luego código de estado, luego cuerpo. Una vez que escribes el primer byte del cuerpo, los encabezados ya se enviaron y no puedes cambiarlos. El patrón correcto:

```
w.Header().Set("Content-Type", "application/json")
w.WriteHeader(http.StatusCreated)
w.Write(datosJSON)
```

Rutas en Go 1.22+: métodos y parámetros en el patrón

Durante años, el mux estándar solo distinguía rutas por la URL, sin importar el método HTTP. Desde Go 1.22, el patrón de registro puede incluir el método y segmentos con nombre, lo que elimina la necesidad de un enrutador externo para la mayoría de las APIs:

```
mux := http.NewServeMux()
mux.HandleFunc("GET /tareas", listarTareas)
mux.HandleFunc("POST /tareas", crearTarea)
mux.HandleFunc("GET /tareas/{id}", obtenerTarea)
mux.HandleFunc("PUT /tareas/{id}", actualizarTarea)
mux.HandleFunc("DELETE /tareas/{id}", eliminarTarea)

http.ListenAndServe(":8080", mux)
```

Fíjate en las novedades:

- `"GET /tareas"` registra el handler solo para peticiones GET a esa ruta. Un POST a `/tareas` irá a otro handler. Antes de Go 1.22 tenías que revisar `r.Method` a mano dentro del handler.
- `"/tareas/{id}"` define un segmento con nombre. Dentro del handler lo recuperas con `r.PathValue("id")`, que regresa el string ("42"). Si necesitas un entero, lo conviertes con `strconv.Atoi` y manejas el error si no es un número.
- Usamos `http.NewServeMux()` en lugar del mux global. Es mejor práctica: tu servidor no depende de estado global y puedes crear varios muxes (por ejemplo, uno para la API y otro para archivos estáticos) y probarlos por separado.

Si una petición no coincide con ninguna ruta, el mux responde 404 automáticamente. Si coincide la ruta pero no el método (por ejemplo, DELETE a una ruta que solo tiene GET), el mux moderno responde 405 Method Not Allowed automáticamente. Antes tenías que implementar eso a mano.

JSON: el idioma de las APIs

Casi todas las APIs modernas hablan JSON. Go lo maneja con el paquete `encoding/json`, que convierte entre structs de Go y texto JSON en ambas direcciones:

- `json.Marshal(valor)` convierte un valor Go a JSON (`[]byte`). "Marshal" es el término técnico para serializar.
- `json.Unmarshal(datos, &variable)` convierte JSON a un valor Go. Necesita un apuntador porque va a modificar la variable.

Las etiquetas (tags) de struct controlan los nombres en el JSON:


```

package main

import (
    "encoding/json"
    "fmt"
    "log"
)

type Tarea struct {
    ID          int    `json:"id"`
    Titulo      string `json:"titulo"`
    Completada  bool   `json:"completada"`
    Secreto     string `json:"-"` // el guion la excluye del JSON
}

func main() {
    t := Tarea{ID: 1, Titulo: "Aprender Go", Completada: false, Secreto: "xxx"}

    datos, err := json.Marshal(t)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(string(datos))
    // {"id":1,"titulo":"Aprender Go","completada":false}

    var t2 Tarea
    err = json.Unmarshal(datos, &t2)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("%+v\n", t2)
}

```

Detalles importantes:

- Solo los campos exportados (que empiezan con mayúscula) se convierten a JSON. Los campos no exportados se ignoran silenciosamente. Este es un error clásico: si tu struct tiene ``titulo`` en minúscula, el JSON saldrá vacío ``{}`` sin ningún error.
- El tag ``json:"titulo"`` define el nombre en el JSON. Sin tag, se usa el nombre del campo tal cual ("Titulo").

- El tag ``json:"-"` excluye el campo: útil para datos internos que nunca deben salir (como el campo Secreto del ejemplo).
- ``json:"titulo,omitempty"` omite el campo si tiene su valor cero (string vacío, 0, false, nil). Útil para respuestas limpias.
- `json.Marshal` puede fallar (por ejemplo, con valores que no se pueden representar como float64 válidos como NaN), así que siempre revisa su error. En handlers HTTP, un error de Marshal típicamente se traduce en un 500.

Para leer el cuerpo de una petición en un handler, la forma moderna y simple es `json.NewDecoder(r.Body).Decode(&variable)`:

```
func crearTarea(w http.ResponseWriter, r *http.Request) {
    var t Tarea
    if err := json.NewDecoder(r.Body).Decode(&t); err != nil {
        http.Error(w, "JSON invalido: "+err.Error(), http.StatusBadRequest)
        return
    }
    // ... guardar t ...
}
```

``http.Error`` es un atajo útil: escribe el mensaje y el código de estado de una vez. Y para responder JSON, el patrón estándar es:

```
w.Header().Set("Content-Type", "application/json")
w.WriteHeader(http.StatusCreated)
json.NewEncoder(w).Encode(respuesta)
```

`json.NewEncoder(w).Encode` escribe el JSON directamente al `ResponseWriter`. Su error rara vez se revisa en la práctica (si falló la escritura, la conexión ya está rota), pero en código muy cuidadoso puedes capturarlo para un log.

Buenas prácticas para handlers HTTP

Antes de construir la API completa, algunas reglas que te ahorrarán dolores de cabeza:

1. Un handler, una responsabilidad. Si tu handler valida, consulta la base de datos, formatea la respuesta y manda correos, es demasiado grande. Separa en funciones.
2. Valida la entrada al inicio y falla rápido con 400 Bad Request si viene mal. No dejes

que datos inválidos viajen por tu sistema.

3. Usa los códigos de estado correctos: 200 OK (éxito con cuerpo), 201 Created (recurso creado), 204 No Content (éxito sin cuerpo, típico en DELETE), 400 Bad Request (el cliente mandó algo mal), 404 Not Found (no existe), 405 Method Not Allowed (método incorrecto), 500 Internal Server Error (falló algo de tu lado).
4. Nunca filtres datos sensibles en las respuestas: usa ``json:"-"` para contraseñas, tokens y datos internos.
5. Protege el acceso concurrente a datos compartidos con `sync.Mutex`. Recuerda: cada petición corre en su propia goroutine, así que dos peticiones pueden tocar tu mapa al mismo tiempo. Un mapa de Go sin protección ante escritura concurrente causa un `panic fatal` que ni `recover` puede atrapar bien (es un error del runtime). En el ejemplo de la API verás el mutex en acción.
6. Pon timeouts al servidor en producción (`http.Server` con `ReadTimeout` y `WriteTimeout`) para que clientes lentos o maliciosos no acaparen tus conexiones.

Variables de entorno: configuración sin recompilar

Un programa bien hecho no lleva sus secretos ni su configuración quemados en el código. El puerto, la cadena de conexión a la base de datos, las llaves de API: todo eso cambia entre tu máquina, el servidor de pruebas y producción. La forma estándar de inyectar configuración es con variables de entorno, que lees con `os.Getenv`:

```

package main

import (
    "fmt"
    "net/http"
    "os"
)

func obtenerEnv(clave, valorPorDefecto string) string {
    if v := os.Getenv(clave); v != "" {
        return v
    }
    return valorPorDefecto
}

func main() {
    puerto := obtenerEnv("PUERTO", "8080")
    ambiente := obtenerEnv("AMBIENTE", "desarrollo")

    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, "Ambiente: %s", ambiente)
    })

    fmt.Println("Escuchando en puerto", puerto)
    http.ListenAndServe(":"+puerto, nil)
}

```

Corres con `PUERTO=9000 AMBIENTE=produccion go run main.go` y el programa se adapta sin recompilar. El helper `obtenerEnv` con valor por defecto es un patrón que verás en todos lados: hace que el programa funcione "de cajón" en desarrollo pero sea configurable en producción.

Regla de oro: nunca subas secretos al repositorio. Las contraseñas y tokens viven en variables de entorno del servidor (o en un manejador de secretos), jamás en el código fuente. Si un secreto alguna vez quedó en un commit, considéralo comprometido y revócalo: borrar el commit no basta porque el historial de git lo conserva.

Compilar para producción: go build

Hasta ahora has corrido programas con `go run`, que compila y ejecuta en un paso. Para producción usas `go build`, que produce un binario ejecutable independiente:

```
go build -o servidor ./cmd/servidor
```

Eso genera un archivo `servidor` que puedes copiar a cualquier máquina con el mismo sistema operativo y arquitectura, y correrlo directamente: `./servidor`. Sin instalar Go, sin runtime, sin dependencias. Esta es una de las grandes ventajas de Go frente a lenguajes interpretados: el despliegue es copiar un archivo.

Dos detalles importantes del binario de Go:

1. Es estático por defecto (cuando usas solo la biblioteca estándar): no depende de bibliotecas del sistema. Por eso los contenedores de Go pueden ser diminutos.
2. Incluye el runtime y el recolector de basura: el binario "hola mundo" pesa unos 2 MB, y un servidor típico entre 10 y 30 MB. Nada grave para los estándares actuales.

Para binarios más pequeños puedes usar banderas del linker:

```
go build -ldflags="-s -w" -o servidor ./cmd/servidor
```

`-s` quita la tabla de símbolos y `-w` la información de depuración. Ahorra algunos MB; el costo es que los stack traces de panics serán menos informativos. En producción muchos equipos lo usan; en desarrollo, no.

Compilación cruzada: GOOS y GOARCH

Go puede compilar para un sistema operativo y arquitectura distintos de los de tu máquina, sin herramientas adicionales. Solo pones dos variables de entorno:

```
# Compilar en Linux para Windows de 64 bits:
GOOS=windows GOARCH=amd64 go build -o servidor.exe ./cmd/servidor

# Compilar en Mac para Linux de 64 bits (típico para desplegar en servidores):
GOOS=linux GOARCH=amd64 go build -o servidor ./cmd/servidor

# Para una Raspberry Pi o un servidor ARM (como AWS Graviton):
GOOS=linux GOARCH=arm64 go build -o servidor ./cmd/servidor
```

Los valores comunes de GOOS: linux, windows, darwin (macOS). Los de GOARCH: amd64 (PCs y servidores Intel/AMD de 64 bits), arm64 (Macs M1/M2/M3, Raspberry Pi modernas, servidores ARM). `go tool dist list` muestra todas las combinaciones soportadas.

Esto es extremadamente práctico: desarrollas en tu laptop (Windows o Mac) y compilas el binario para el servidor Linux donde va a correr, sin máquinas virtuales ni contenedores intermedios. El binario resultante corre tal cual en el destino.

Dockerfile multietapa para Go

El despliegue moderno suele ser con contenedores. Un Dockerfile multietapa para Go usa una imagen grande con el toolchain para compilar, y una imagen mínima para correr:

```
# Etapa 1: compilacion
FROM golang:1.22-alpine AS builder
WORKDIR /app
COPY go.mod go.sum ./
RUN go mod download
COPY . .
RUN CGO_ENABLED=0 GOOS=linux go build -ldflags="-s -w" -o servidor ./cmd/servidor

# Etapa 2: imagen final minima
FROM alpine:latest
WORKDIR /app
COPY --from=builder /app/servidor .
EXPOSE 8080
CMD ["/servidor"]
```

Qué hace cada parte:

- La primera etapa parte de la imagen oficial de Go, descarga dependencias y compila. ``CGO_ENABLED=0`` fuerza un binario 100% estático (sin dependencias de C), necesario para que corra en imágenes mínimas como alpine o scratch.
- Se copian primero `go.mod` y `go.sum` y se descargan dependencias antes de copiar el código: así Docker reutiliza la capa de dependencias si solo cambió tu código, acelerando las construcciones.
- La segunda etapa parte de alpine (unos 5 MB) y solo copia el binario compilado. La imagen final pesa ~20-30 MB en lugar de los ~500 MB de la imagen de Go. Menos tamaño = despliegues más rápidos y menor superficie de ataque.

Aún más minimalista: puedes usar ``FROM scratch`` (imagen totalmente vacía) en lugar de alpine, quedando solo tu binario. Funciona porque el binario es estático, pero pierdes herramientas básicas (ni siquiera hay shell para depurar) y certificados SSL del sistema, que

necesitarías copiar a mano si tu programa hace peticiones HTTPS salientes. Para empezar, alpine es el equilibrio sensato.

A dónde seguir: el ecosistema Go

Con lo de este libro ya puedes construir software real, pero el ecosistema de Go es vasto. Este es el mapa de los siguientes pasos, en orden de utilidad práctica:

- Frameworks web: Gin y Echo. `net/http` basta para muchas APIs, pero Gin y Echo agregan enrutamiento avanzado, middleware encadenable, validación y manejo de errores con menos código repetitivo. Aprende `net/http` primero (ya lo hiciste) y un framework después: así entenderás qué magia hace el framework por ti.
- Bases de datos: el paquete `database/sql` de la biblioteca estándar define la interfaz común, y los drivers específicos (para PostgreSQL, MySQL, SQLite) la implementan. Para no escribir SQL a mano hay ORMs como GORM, aunque muchos gophers prefieren SQL directo con `sqlx` por control y rendimiento.
- gRPC: framework de comunicación entre servicios de Google, con Go como lenguaje de primera clase. Cuando tus sistemas crecen de una API a muchos microservicios que se hablan entre sí, gRPC con Protocol Buffers es el siguiente nivel.
- Concurrencia avanzada: los patrones de la Unidad 8 (worker pools, fan-out/fan-in, context para cancelación) son el pan de cada día en backends reales.
- Observabilidad: logging estructurado con slog, métricas con Prometheus, tracing distribuido. Cuando tu servidor atiende miles de peticiones, necesitas ver qué pasa adentro sin adivinar.

Contexto en peticiones: `context.Context` y cancelación

Cada `*http.Request` trae un contexto (`r.Context()`) que representa la vida de esa petición: si el cliente se desconecta o cancela, el contexto se cancela. Tu handler puede (y debe) escuchar esa señal para dejar de trabajar en algo que ya nadie espera. Esto evita que tu servidor siga quemando CPU y memoria en peticiones abandonadas.

```

mux.HandleFunc("GET /reporte", func(w http.ResponseWriter, r *http.Request) {
    ctx := r.Context()
    for i := 0; i < 100; i++ {
        select {
        case <-ctx.Done():
            // El cliente se fue: dejar de trabajar.
            log.Println("peticion cancelada por el cliente")
            return
        default:
            // ... procesar el pedazo i del reporte ...
            time.Sleep(100 * time.Millisecond)
        }
    }
    fmt.Fprintln(w, "reporte listo")
})

```

El canal `ctx.Done()` se cierra cuando la petición termina o se cancela. El `select` lo revisa en cada iteración sin bloquearse (gracias al `default`). Si el cliente cierra su navegador a mitad del reporte, tu handler lo detecta y deja de trabajar en lugar de seguir 10 segundos más para nada.

El contexto también viaja hacia abajo: cuando tu handler llama a la base de datos o a otra API, pásale el contexto (`db.QueryContext(ctx, ...)`, `http.NewRequestWithContext(...)`). Así la cancelación se propaga por toda la cadena: si el cliente se desconecta, la consulta a la base de datos también se cancela. Regla: el contexto entra por la petición y baja por las llamadas; nunca lo guardes en structs ni lo uses para pasar datos de negocio (para eso están los parámetros normales).

Graceful shutdown: apagar sin cortar peticiones en vuelo

Cuando despliegas una versión nueva, tu servidor necesita apagarse para que el nuevo ocupe su lugar. La forma brusca (matar el proceso) corta las peticiones que están a la mitad: el usuario recibe un error aunque su petición casi terminaba. El apagado elegante (graceful shutdown) hace esto: deja de aceptar peticiones nuevas, espera a que terminen las que están en vuelo (con un límite de tiempo), y entonces sí se apaga.


```

package main

import (
    "context"
    "fmt"
    "net/http"
    "os"
    "os/signal"
    "syscall"
    "time"
)

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /lento", func(w http.ResponseWriter, r *http.Request) {
        time.Sleep(5 * time.Second) // simula trabajo largo
        fmt.Fprintln(w, "terminado")
    })

    servidor := &http.Server{Addr: ":8080", Handler: mux}

    // Canal que avisa cuando el SO pide terminar (Ctrl+C o docker stop).
    parar := make(chan os.Signal, 1)
    signal.Notify(parar, os.Interrupt, syscall.SIGTERM)

    go func() {
        fmt.Println("Servidor en http://localhost:8080")
        if err := servidor.ListenAndServe(); err != http.ErrServerClosed {
            fmt.Println("Error del servidor:", err)
        }
    }()

    <-parar // esperar la señal de apagado
    fmt.Println("\nApagando con gracia...")

    ctx, cancelar := context.WithTimeout(context.Background(), 10*time.Second)
    defer cancelar()
    if err := servidor.Shutdown(ctx); err != nil {
        fmt.Println("Apagado forzoso:", err)
    }
    fmt.Println("Servidor detenido")
}

```

Cómo funciona: el servidor corre en una goroutine para no bloquear el main. El main se queda esperando en `<-parar` hasta que llega `SIGINT` (Ctrl+C) o `SIGTERM` (lo que manda ``docker stop`` y Kubernetes al terminar un contenedor). Entonces llama a `Shutdown` con un contexto de 10 segundos: el servidor cierra el listener (ya no acepta conexiones nuevas) pero deja terminar las activas; si en 10 segundos alguna no termina, se corta a la fuerza. Prueba: pide `/lento` en una terminal, dale Ctrl+C al servidor en otra, y verás que la petición lenta sí recibe su respuesta antes de que el servidor muera. En producción este patrón no es opcional: es lo que separa un despliegue limpio de usuarios viendo errores 502.

Estructura de proyecto: cómo organizar un servicio Go

Los ejemplos de esta unidad viven en un solo archivo `main` para que sean fáciles de seguir, pero un servicio real se organiza en paquetes. La convención más usada en la comunidad:

```
mi-api/  
  go.mod  
  cmd/  
    servidor/  
      main.go      // solo: leer config, crear dependencias, arrancar  
  internal/  
    tareas/  
      tareas.go    // modelo + tienda (logica de negocio)  
      http.go      // handlers HTTP  
  config/  
    config.go      // leer variables de entorno  
  Dockerfile  
  README.md
```

Las reglas detrás de esta estructura:

- `cmd/` contiene los puntos de entrada (uno por binario). El main debe ser delgado: armar piezas y arrancar, sin lógica de negocio.
- `internal/` es especial en Go: el compilador impide que paquetes fuera de tu módulo importen lo que está adentro. Es el lugar para tu código privado.
- Un paquete por dominio (tareas, usuarios, config): cada uno con su modelo, su lógica y sus handlers. Los handlers reciben sus dependencias (la tienda) en lugar de usar variables globales: así son testeables con `httptest`, como viste en el Ejercicio 6.
- La lógica de negocio (la tienda) no sabe nada de HTTP: trabaja con structs y errores de

Go puro. Los handlers traducen entre HTTP y esa lógica. Esta separación es la que permite probar la lógica sin servidor y reusarla si mañana agregas otro frontend (un CLI, gRPC, etc.).

No hay una estructura oficial obligatoria y verás variantes (algunos usan pkg/ para código compartible, otros ponen todo en la raíz para proyectos chicos). Para empezar, esta es más que suficiente; lo importante es el principio: main delgado, dependencias inyectadas, HTTP separado de la lógica.

Servir HTTPS: TLS con un certificado

En producción tu API debe hablar HTTPS, no HTTP plano: sin cifrado, cualquiera en la red puede leer los datos (incluidos tokens y contraseñas). Go trae TLS integrado; lo único que necesitas es un certificado. Para desarrollo puedes generar uno autofirmado:

```
go run /usr/local/go/src/crypto/tls/generate_cert.go --host=localhost
```

(En instalaciones normales de Go ese programa de ejemplo viene incluido; genera cert.pem y key.pem.) Y el servidor cambia una línea:

```
// En lugar de ListenAndServe:  
log.Fatal(servidor.ListenAndServeTLS("cert.pem", "key.pem"))
```

ListenAndServeTLS configura todo el cifrado por ti. El navegador te advertirá que el certificado autofirmado no es confiable (es normal: tú lo firmaste, ninguna autoridad lo avala); en producción usarías un certificado real, típicamente gratuito vía Let's Encrypt, o lo terminarías en un balanceador/reverse proxy (nginx, Caddy, o el balanceador de tu nube) que hace el TLS y le habla a tu servidor Go en HTTP interno. Esa última arquitectura es la más común: tu binario Go escucha en localhost:8080 sin TLS, y el proxy de enfrente maneja los certificados y el tráfico público. Así tu código no se entera del TLS y el proxy, que está hecho para eso, lo hace mejor.

Métricas básicas: saber qué hace tu servidor en producción

Un servidor en producción es una caja negra si no lo instrumentas: ¿cuántas peticiones atiende? ¿cuáles son lentas? ¿cuántas fallan? El paquete expvar de la biblioteca estándar expone variables y contadores por HTTP sin dependencias:

```

package main

import (
    "expvar"
    "fmt"
    "net/http"
    "time"
)

var (
    peticionesTotal = expvar.NewInt("peticiones_total")
    peticionesOK    = expvar.NewInt("peticiones_ok")
    peticionesErr   = expvar.NewInt("peticiones_error")
)

func conMetricas(siguiete http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        peticionesTotal.Add(1)
        inicio := time.Now()
        siguiete.ServeHTTP(w, r)
        _ = inicio
        // En un middleware real capturarias el codigo de estado con un
        // ResponseWriter envuelto; aqui simplificamos:
        peticionesOK.Add(1)
    })
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /hola", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "hola")
    })
    // expvar publica todo en /debug/vars automaticamente
    fmt.Println("Metricas en http://localhost:8080/debug/vars")
    http.ListenAndServe(":8080", conMetricas(mux))
}

```

Visita `/debug/vars` y verás un JSON con tus contadores más estadísticas del runtime (goroutines, memoria). En sistemas reales estas métricas las recoge Prometheus (hay bibliotecas cliente para Go) y las graficas en dashboards con alertas ("si los errores 500 suben de X por minuto, avísame"). La lección: instrumenta desde el primer día, aunque sea con contadores simples. Cuando algo falle en producción a las 3 de la mañana, la diferencia entre "no sé qué pasa" y "veo exactamente qué endpoint falla" es tener métricas.

Tu primera base de datos: database/sql

El almacenamiento en memoria se pierde al reiniciar el servidor. El paso natural es una base de datos, y Go trae la interfaz estándar en el paquete `database/sql`: tú escribes el mismo código sin importar la base de datos, y un driver específico implementa la conexión. Para empezar, SQLite es ideal: es un archivo local, sin servidor que instalar.

```
package main

import (
    "database/sql"
    "fmt"
    "log"

    _ "github.com/mattn/go-sqlite3" // el driver se registra solo al importarlo
)

type Tarea struct {
    ID          int
    Titulo      string
    Completada  bool
}

func main() {
    db, err := sql.Open("sqlite3", "tareas.db")
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()

    // Crear la tabla si no existe.
    _, err = db.Exec(`CREATE TABLE IF NOT EXISTS tareas (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        titulo TEXT NOT NULL,
        completada INTEGER NOT NULL DEFAULT 0
    )`)
    if err != nil {
        log.Fatal(err)
    }

    // Insertar.
    res, err := db.Exec("INSERT INTO tareas (titulo) VALUES (?)", "Aprender Go")
    if err != nil {
        log.Fatal(err)
    }
}
```

```

id, _ := res.LastInsertId()
fmt.Println("Insertada con id:", id)

// Consultar.
filas, err := db.Query("SELECT id, titulo, completada FROM tareas")
if err != nil {
    log.Fatal(err)
}
defer filas.Close()

for filas.Next() {
    var t Tarea
    var comp int
    if err := filas.Scan(&t.ID, &t.Titulo, &comp); err != nil {
        log.Fatal(err)
    }
    t.Completada = comp == 1
    fmt.Printf("%+v\n", t)
}
if err := filas.Err(); err != nil {
    log.Fatal(err)
}
}

```

Los puntos clave del patrón database/sql:

- sql.Open no conecta de inmediato: solo valida los argumentos. La primera consulta real abre la conexión (o falla). Por eso un buen programa hace db.Ping() al arrancar para fallar rápido si la base no está disponible.
- db es un pool de conexiones, seguro para usar desde múltiples goroutines: no necesitas mutex, el paquete lo maneja.
- Los ? son placeholders: NUNCA concatenes valores en el SQL a mano (eso abre la puerta a inyección SQL). Cada driver tiene su estilo (PostgreSQL usa \$1, \$2), pero la idea es la misma.
- filas.Close() con defer y revisar filas.Err() al final: los errores pueden ocurrir a mitad de la iteración, no solo al hacer Query.

Para usarlo necesitas el módulo y el driver: ``go mod init miapp`` y ``go get github.com/mattn/go-sqlite3``. (Nota: ese driver usa cgo, así que para binarios 100% estáticos

en Docker se suele preferir modernc.org/sqlite, que es puro Go.) El salto desde aquí a PostgreSQL es cambiar el driver y la cadena de conexión: tu código con `database/sql` casi no cambia. Y cuando quieras, la API de tareas del Ejemplo 3 puede reescribir su Tienda para guardar en la base en lugar del mapa, sin tocar los handlers: esa es la recompensa de haber separado la lógica del HTTP.

Despliegue en un VPS paso a paso (sin Docker)

No todo despliegue necesita contenedores. Para un proyecto chico o tu primer servidor en producción, un VPS barato con tu binario corriendo como servicio del sistema es simple y suficiente. El flujo completo:

1. Compila para el servidor: ``GOOS=linux GOARCH=amd64 go build -ldflags="-s -w" -o servidor ./cmd/servidor`` (si tu VPS es ARM, usa `GOARCH=arm64`).
2. Copia el binario al servidor: ``scp servidor usuario@tu-servidor:/home/usuario/``.
3. En el servidor, crea un servicio systemd en `/etc/systemd/system/mi-api.service`:

```
[Unit]
Description=Mí API en Go
After=network.target

[Service]
User=usuario
WorkingDirectory=/home/usuario
Environment=PUERTO=8080
Environment=AMBIENTE=produccion
ExecStart=/home/usuario/servidor
Restart=always
RestartSec=5

[Install]
WantedBy=multi-user.target
```

4. Actívalo: ``sudo systemctl enable --now mi-api``. Con `Restart=always`, si tu programa muere por cualquier razón, el sistema lo levanta solo en 5 segundos.
5. Pon un reverse proxy enfrente (Caddy es el más simple: una línea ``tu-dominio.com { reverse_proxy localhost:8080 }`` y te da HTTPS automático con Let's Encrypt).

Los logs del programa los ves con `journalctl -u mi-api -f`. Las variables de entorno van en el archivo del servicio, no en tu código. Y para actualizar: compilas el binario nuevo, lo copias, y `sudo systemctl restart mi-api`. Con el graceful shutdown de la sección anterior, el restart no corta peticiones en vuelo. Este esquema atiende sin problemas miles de peticiones por minuto; cuando necesites más, ya será momento de contenedores y orquestadores, pero no antes.

Seguridad mínima: lo que todo servidor debe tener

Un servidor expuesto a internet recibe tráfico malicioso en minutos, a veces segundos. No necesitas ser experto en seguridad, pero sí cubrir lo básico desde el primer despliegue:

1. Valida toda entrada como si fuera hostil. Límites de tamaño en el cuerpo

(`http.MaxBytesReader` para que nadie te mande un JSON de 1 GB), validación de tipos y rangos, y nunca interpolar datos del usuario en comandos, SQL o HTML sin escapar. `html/template` ya escapa el HTML; `database/sql` con placeholders evita la inyección SQL.

2. Pon encabezados de seguridad. Un middleware sencillo agrega varias capas:

```
func conSeguridad(siguiete http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("X-Content-Type-Options", "nosniff")
        w.Header().Set("X-Frame-Options", "DENY")
        w.Header().Set("Content-Security-Policy", "default-src 'self'")
        siguiete.ServeHTTP(w, r)
    })
}
```

3. Limita el tamaño de lo que aceptas: `r.Body = http.MaxBytesReader(w, r.Body, 1<<20)`

(1 MB) al inicio de los handlers que reciben cuerpos. Sin esto, un cliente puede agotar tu memoria con una sola petición.

4. No expongas información interna en los errores: al cliente un 500 genérico ("error interno"), el detalle al log. Un stack trace en la respuesta le regala al atacante el mapa de tu código.

5. Mantén Go actualizado: las versiones nuevas incluyen parches de seguridad de `net/http` y `crypto/tls`. Revisa `go version` en tus servidores cada pocos meses.

Ninguna de estas medidas es difícil, y juntas eliminan la gran mayoría de los ataques oportunistas. La seguridad perfecta no existe; la seguridad "suficientemente molesta para el atacante" sí, y se construye con estos hábitos.

Frameworks: cuándo usar Gin o Echo y cuándo no

Con todo lo que ya sabes hacer con net/http, ¿para qué un framework? Gin y Echo no hacen magia: por dentro usan lo mismo que tú ya conoces. Lo que te ahorran es código repetitivo en APIs grandes: validación automática de structs con tags, grupos de rutas con middleware por grupo, manejo centralizado de errores, y renderizado de respuestas en una línea. Un ejemplo mínimo en Gin para comparar:

```
// Con net/http necesitas ~15 lineas para validar y responder JSON.
// Con Gin, el mismo endpoint:
r := gin.Default()
r.POST("/tareass", func(c *gin.Context) {
    var entrada struct {
        Titulo string `json:"titulo" binding:"required"`
    }
    if err := c.ShouldBindJSON(&entrada); err != nil {
        c.JSON(400, gin.H{"error": err.Error()})
        return
    }
    c.JSON(201, gin.H{"titulo": entrada.Titulo})
})
```

El tag ``binding:"required"``` valida solo: si falta el título, regresa 400 sin que escribas la validación. En una API de 50 endpoints, ese ahorro se acumula.

¿Cuándo NO usar framework? En servicios pequeños, en bibliotecas, cuando quieres cero dependencias externas, o cuando estás aprendiendo (como ahora): si empiezas con el framework, nunca entiendes qué hace por ti y te atorras cuando algo se sale del camino feliz. La recomendación de la comunidad: domina net/http primero (ya lo hiciste en esta unidad) y adopta Gin o Echo cuando el proyecto crezca al punto donde el código repetitivo duela. Y aun con framework, todo lo de esta unidad sigue aplicando: los errores se manejan igual, el mutex sigue siendo necesario, el graceful shutdown se configura igual. El framework cambia la envoltura, no los fundamentos.

Ejemplos paso a paso

Ejemplo 1: Servidor básico con varias rutas y método

Un servidor que responde distinto según la ruta y que muestra información de la petición.

```

package main

import (
    "fmt"
    "net/http"
)

func main() {
    mux := http.NewServeMux()

    mux.HandleFunc("GET /", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Bienvenido a mi servidor Go")
    })

    mux.HandleFunc("GET /saludo", func(w http.ResponseWriter, r *http.Request) {
        nombre := r.URL.Query().Get("nombre")
        if nombre == "" {
            nombre = "visitante"
        }
        fmt.Fprintf(w, "Hola, %s\n", nombre)
    })

    mux.HandleFunc("GET /info", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, "Metodo: %s\nRuta: %s\nProtocolo: %s\n",
            r.Method, r.URL.Path, r.Proto)
        fmt.Fprintf(w, "Tu navegador dice ser: %s\n", r.UserAgent())
    })

    fmt.Println("Servidor en http://localhost:8080")
    fmt.Println("Prueba: /saludo?nombre=Ana")
    http.ListenAndServe(":8080", mux)
}

```

Explicación: tres rutas GET en un mux propio. /saludo lee el parámetro de consulta `nombre` con `r.URL.Query().Get()`; si no viene, usa un valor por defecto. /info muestra metadatos de la petición: método, ruta, protocolo y user agent. Prueba con curl: ``curl "http://localhost:8080/saludo?nombre=Ana"``.

Ejemplo 2: Servidor JSON que responde y recibe

Un endpoint que recibe JSON por POST, lo procesa y responde JSON, con manejo de errores correcto.

```

package main

import (
    "encoding/json"
    "fmt"
    "net/http"
    "strings"
)

type Peticion struct {
    Nombre string `json:"nombre"`
}

type Respuesta struct {
    Mensaje string `json:"mensaje"`
    Largo   int    `json:"largo"`
}

func gritar(w http.ResponseWriter, r *http.Request) {
    var p Peticion
    if err := json.NewDecoder(r.Body).Decode(&p); err != nil {
        http.Error(w, "JSON invalido: "+err.Error(), http.StatusBadRequest)
        return
    }
    if strings.TrimSpace(p.Nombre) == "" {
        http.Error(w, "el campo nombre es obligatorio", http.StatusBadRequest)
        return
    }

    resp := Respuesta{
        Mensaje: strings.ToUpper("hola " + p.Nombre),
        Largo:   len(p.Nombre),
    }
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusOK)
    json.NewEncoder(w).Encode(resp)
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("POST /gritar", gritar)
    fmt.Println("Servidor en http://localhost:8080")
    http.ListenAndServe(":8080", mux)
}

```

Explicación: el handler decodifica el cuerpo JSON; si el JSON está mal formado regresa 400 de inmediato. Luego valida el contenido (nombre no vacío) con otro 400. Solo entonces procesa y responde 200 con JSON. Prueba con: ``curl -X POST localhost:8080/gritar -d '{"nombre":"Ana"}'`` y prueba los errores con JSON roto o sin nombre para ver los 400.

Ejemplo 3: API REST completa, CRUD de tareas en memoria

El ejemplo central de la unidad: una API REST funcional con las cinco operaciones, almacenamiento en memoria protegido con mutex, y códigos de estado correctos. Es largo, pero cada parte usa cosas que ya viste.

```
package main

import (
    "encoding/json"
    "fmt"
    "net/http"
    "strconv"
    "sync"
)

type Tarea struct {
    ID          int    `json:"id"`
    Titulo      string `json:"titulo"`
    Completada  bool   `json:"completada"`
}

// Tienda guarda las tareas en memoria de forma segura para concurrencia.
type Tienda struct {
    mu          sync.Mutex
    tareas      map[int]Tarea
    siguienteID int
}

func NuevaTienda() *Tienda {
    return &Tienda{tareas: map[int]Tarea{}, siguienteID: 1}
}

func (t *Tienda) listar() []Tarea {
    t.mu.Lock()
    defer t.mu.Unlock()
    lista := make([]Tarea, 0, len(t.tareas))
    for _, tarea := range t.tareas {
        lista = append(lista, tarea)
    }
}
```

```

    }
    return lista
}

func (t *Tienda) obtener(id int) (Tarea, bool) {
    t.mu.Lock()
    defer t.mu.Unlock()
    tarea, ok := t.tareas[id]
    return tarea, ok
}

func (t *Tienda) crear(titulo string) Tarea {
    t.mu.Lock()
    defer t.mu.Unlock()
    tarea := Tarea{ID: t.siguienteID, Titulo: titulo}
    t.tareas[t.siguienteID] = tarea
    t.siguienteID++
    return tarea
}

func (t *Tienda) actualizar(id int, titulo string, completada bool) (Tarea, bool) {
    t.mu.Lock()
    defer t.mu.Unlock()
    tarea, ok := t.tareas[id]
    if !ok {
        return Tarea{}, false
    }
    tarea.Titulo = titulo
    tarea.Completada = completada
    t.tareas[id] = tarea
    return tarea, true
}

func (t *Tienda) eliminar(id int) bool {
    t.mu.Lock()
    defer t.mu.Unlock()
    if _, ok := t.tareas[id]; !ok {
        return false
    }
    delete(t.tareas, id)
    return true
}

// responderJSON escribe una respuesta JSON con el código de estado dado.
func responderJSON(w http.ResponseWriter, codigo int, v any) {

```

```

w.Header().Set("Content-Type", "application/json")
w.WriteHeader(codigo)
json.NewEncoder(w).Encode(v)
}

func main() {
    tienda := NuevaTienda()
    mux := http.NewServeMux()

    // GET /tareass -> lista todas
    mux.HandleFunc("GET /tareass", func(w http.ResponseWriter, r *http.Request) {
        responderJSON(w, http.StatusOK, tienda.listar())
    })

    // POST /tareass -> crea una
    mux.HandleFunc("POST /tareass", func(w http.ResponseWriter, r *http.Request) {
        var entrada struct {
            Titulo string `json:"titulo"`
        }
        if err := json.NewDecoder(r.Body).Decode(&entrada); err != nil {
            http.Error(w, "JSON invalido", http.StatusBadRequest)
            return
        }
        if entrada.Titulo == "" {
            http.Error(w, "el titulo es obligatorio", http.StatusBadRequest)
            return
        }
        tarea := tienda.crear(entrada.Titulo)
        responderJSON(w, http.StatusCreated, tarea)
    })

    // GET /tareass/{id} -> obtiene una
    mux.HandleFunc("GET /tareass/{id}", func(w http.ResponseWriter, r *http.Request) {
        id, err := strconv.Atoi(r.PathValue("id"))
        if err != nil {
            http.Error(w, "id invalido", http.StatusBadRequest)
            return
        }
        tarea, ok := tienda.obtener(id)
        if !ok {
            http.Error(w, "tarea no encontrada", http.StatusNotFound)
            return
        }
        responderJSON(w, http.StatusOK, tarea)
    })
}

```

```

// PUT /tareas/{id} -> actualiza una
mux.HandleFunc("PUT /tareas/{id}", func(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(r.PathValue("id"))
    if err != nil {
        http.Error(w, "id invalido", http.StatusBadRequest)
        return
    }
    var entrada struct {
        Titulo      string `json:"titulo"`
        Completada bool   `json:"completada"`
    }
    if err := json.NewDecoder(r.Body).Decode(&entrada); err != nil {
        http.Error(w, "JSON invalido", http.StatusBadRequest)
        return
    }
    tarea, ok := tienda.actualizar(id, entrada.Titulo, entrada.Completada)
    if !ok {
        http.Error(w, "tarea no encontrada", http.StatusNotFound)
        return
    }
    responderJSON(w, http.StatusOK, tarea)
})

// DELETE /tareas/{id} -> elimina una
mux.HandleFunc("DELETE /tareas/{id}", func(w http.ResponseWriter, r *http.Request)
{
    id, err := strconv.Atoi(r.PathValue("id"))
    if err != nil {
        http.Error(w, "id invalido", http.StatusBadRequest)
        return
    }
    if !tienda.eliminar(id) {
        http.Error(w, "tarea no encontrada", http.StatusNotFound)
        return
    }
    w.WriteHeader(http.StatusNoContent)
})

fmt.Println("API de tareas en http://localhost:8080")
http.ListenAndServe(":8080", mux)
}

```

Explicación por partes:

- El struct Tarea es el modelo con tags JSON. La Tienda encapsula el mapa y el mutex:

ningún handler toca el mapa directamente, todos pasan por métodos que bloquean el mutex. Así, cien peticiones simultáneas no corrompen los datos.

- responderJSON centraliza el patrón de respuesta (encabezado + código + cuerpo) para no

repetirlo en cada handler.

- Cada handler sigue el mismo ritmo: extraer y validar entrada (400 si está mal), operar, responder con el código correcto (200, 201, 204, 404). El id de la ruta se convierte con `strconv.Atoi`; si no es número, 400.

- DELETE regresa 204 No Content sin cuerpo: el estándar para "se eliminó, no hay nada más

que decir".

Para probarla:

```
curl -X POST localhost:8080/tareas -d '{"titulo":"Aprender Go"}'
curl localhost:8080/tareas
curl localhost:8080/tareas/1
curl -X PUT localhost:8080/tareas/1 -d '{"titulo":"Aprender Go","completada":true}'
curl -X DELETE localhost:8080/tareas/1
```

Este ejemplo es intencionalmente parecido a lo que harías en un trabajo real, solo que con almacenamiento en memoria en lugar de base de datos. La estructura (modelo, tienda con métodos, handlers delgados, helper de respuesta) escala directamente a proyectos profesionales.

Ejemplo 4: Plantilla HTML con `html/template`

Cuando necesitas servir páginas web (no solo JSON), el paquete `html/template` genera HTML de forma segura: escapa automáticamente los datos para prevenir ataques XSS.


```

package main

import (
    "fmt"
    "html/template"
    "net/http"
)

type Pagina struct {
    Titulo string
    Tareas []string
}

var plantilla = template.Must(template.New("lista").Parse(`
<!DOCTYPE html>
<html>
<head><title>{{.Titulo}}</title></head>
<body>
<h1>{{.Titulo}}</h1>
<ul>
{{range .Tareas}}<li>{{.}}</li>{{end}}
</ul>
</body>
</html>
`))

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /", func(w http.ResponseWriter, r *http.Request) {
        datos := Pagina{
            Titulo: "Mis tareas",
            Tareas: []string{"Aprender Go", "Hacer la API", "Desplegar"},
        }
        w.Header().Set("Content-Type", "text/html; charset=utf-8")
        plantilla.Execute(w, datos)
    })
    fmt.Println("Servidor en http://localhost:8080")
    http.ListenAndServe(":8080", mux)
}

```

Explicación: la plantilla se compila una vez al inicio con `template.Must` (si la plantilla tiene un error de sintaxis, el programa falla al arrancar en lugar de fallar en la primera petición: fail fast). ``{{.Titulo}}`` inserta el campo, ``{{range .Tareas}}...{{end}}`` itera. Si una tarea contuviera ``<script>alert("x")</script>``, `html/template` lo escaparía a texto visible en lugar de ejecutarlo:

esa es la protección XSS automática. Nota la diferencia con text/template (sin escape, para texto plano): en web siempre usa html/template.

Ejemplo 5: Middleware de logging y recover

El middleware es una función que envuelve a un handler para agregarle comportamiento (logging, autenticación, recover) sin modificarlo. Es el patrón estándar para preocupaciones transversales.

```
package main

import (
    "fmt"
    "log"
    "net/http"
    "time"
)

// conLogging mide y registra cada petición.
func conLogging(siguiete http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        inicio := time.Now()
        siguiete.ServeHTTP(w, r)
        log.Printf("%s %s %v", r.Method, r.URL.Path, time.Since(inicio))
    })
}

// conRecover evita que un panic en un handler tumbe el servidor.
func conRecover(siguiete http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        defer func() {
            if rec := recover(); rec != nil {
                log.Printf("panic recuperado: %v", rec)
                http.Error(w, "error interno", http.StatusInternalServerError)
            }
        }()
        siguiete.ServeHTTP(w, r)
    })
}

func hola(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Hola con middleware")
}

func explota(w http.ResponseWriter, r *http.Request) {
```

```

    panic("bug simulado en el handler")
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /hola", hola)
    mux.HandleFunc("GET /explota", explota)

    // Encadenar: primero recover (afuera), luego logging, luego el mux.
    manejador := conRecover(conLogging(mux))

    fmt.Println("Servidor en http://localhost:8080")
    fmt.Println("Prueba /explota: el servidor sobrevive")
    http.ListenAndServe(":8080", manejador)
}

```

Explicación: cada middleware recibe el siguiente handler y regresa uno nuevo que lo envuelve. El orden de encadenado importa: `conRecover` va afuera para atrapar panics de todo lo de adentro (incluyendo el logging). Si visitas `/explota`, el panic se recupera, se registra en el log, el cliente recibe un 500, y el servidor sigue atendiendo `/hola` como si nada. Sin este middleware, un solo panic tumbaría todo el proceso. En la Unidad 9 vimos `recover` como concepto; aquí está su uso real más común.

Ejemplo 6: Servidor configurable con variables de entorno y timeouts

Juntando configuración por entorno y un `http.Server` con timeouts: así se ve un main de producción.

```

package main

import (
    "fmt"
    "net/http"
    "os"
    "time"
)

func obtenerEnv(clave, defecto string) string {
    if v := os.Getenv(clave); v != "" {
        return v
    }
    return defecto
}

func main() {
    puerto := obtenerEnv("PUERTO", "8080")

    mux := http.NewServeMux()
    mux.HandleFunc("GET /salud", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "ok")
    })

    servidor := &http.Server{
        Addr:         ":" + puerto,
        Handler:       mux,
        ReadTimeout:  5 * time.Second,
        WriteTimeout: 10 * time.Second,
        IdleTimeout:  60 * time.Second,
    }

    fmt.Println("Servidor en puerto", puerto)
    if err := servidor.ListenAndServe(); err != nil {
        fmt.Println("El servidor termino:", err)
    }
}

```

Explicación: en lugar de la función global `http.ListenAndServe` usamos un `http.Server` configurado. Los timeouts protegen contra clientes lentos: `ReadTimeout` limita el tiempo para leer la petición completa, `WriteTimeout` el tiempo para escribir la respuesta, `IdleTimeout` cuánto puede durar una conexión inactiva en keep-alive. El endpoint `/salud` (health check) es convención en producción: los balanceadores y sistemas de monitoreo lo usan para saber si tu servidor sigue vivo. Y el puerto sale de la variable `PUERTO` con valor por defecto 8080.

Ejemplo 7: Listado con paginación y filtrado

Problema: el endpoint `GET /tareas` devuelve todas las tareas; con miles de registros eso es lento y pesado. Agregar paginación (`?pagina=2&limite=10`) y filtrado (`?completada=true`), con valores por defecto sensatos y validación.

```
package main

import (
    "encoding/json"
    "fmt"
    "net/http"
    "strconv"
)

type Tarea struct {
    ID          int    `json:"id"`
    Titulo      string `json:"titulo"`
    Completada  bool   `json:"completada"`
}

var tareas = []Tarea{
    {1, "Aprender Go", true},
    {2, "Hacer la API", true},
    {3, "Escribir tests", false},
    {4, "Desplegar", false},
    {5, "Celebrar", false},
}

type PaginaRespuesta struct {
    Pagina    int    `json:"pagina"`
    Limite    int    `json:"limite"`
    Total     int    `json:"total"`
    Paginas   int    `json:"paginas"`
    Tareas    []Tarea `json:"tareas"`
}
```

```

func listarPaginado(w http.ResponseWriter, r *http.Request) {
    q := r.URL.Query()

    pagina, err := strconv.Atoi(q.Get("pagina"))
    if err != nil || pagina < 1 {
        pagina = 1
    }
    limite, err := strconv.Atoi(q.Get("limite"))
    if err != nil || limite < 1 || limite > 100 {
        limite = 10
    }

    // Filtrado opcional por completada.
    filtrar := q.Get("completada") // "true", "false" o vacio
    filtradas := tareas[:0:0] // slice vacio con capacidad 0
    filtradas = []Tarea{}
    for _, t := range tareas {
        switch filtrar {
            case "true":
                if t.Completada {
                    filtradas = append(filtradas, t)
                }
            case "false":
                if !t.Completada {
                    filtradas = append(filtradas, t)
                }
            default:
                filtradas = append(filtradas, t)
            }
        }

    total := len(filtradas)
    paginas := (total + limite - 1) / limite // division redondeando hacia arriba
    inicio := (pagina - 1) * limite
    if inicio > total {
        inicio = total
    }
    fin := inicio + limite
    if fin > total {
        fin = total
    }

    resp := PaginaRespuesta{
        Pagina: pagina,
        Limite: limite,
    }
}

```

```

        Total:  total,
        Paginas: paginas,
        Tareas:  filtradas[inicio:fin],
    }
    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(resp)
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /tareas", listarPaginado)
    fmt.Println("Prueba: /tareas?pagina=1&limite=2&completada=false")
    http.ListenAndServe(":8080", mux)
}

```

Explicación: los parámetros se leen con `q.Get()` y se convierten con `strconv.Atoi`; si vienen mal o fuera de rango, se usan valores por defecto en lugar de regresar error (decisión de diseño: para paginación los defaults son amables; para un id inválido en la ruta sí regresábamos 400). El límite máximo de 100 protege al servidor de un cliente que pida un millón de registros. La respuesta incluye metadatos (total, páginas) para que el cliente sepa si hay más. Este es el formato de paginación que verás en casi todas las APIs públicas, y ahora sabes construirlo.

Práctica

Ejercicio 1: API de chistes con contador

Crea un servidor con dos rutas: ``GET /chiste`` regresa un chiste aleatorio en JSON ``{"chiste": "...}"`, y ``GET /estadisticas`` regresa ``{"servidos": N}`` con cuántos chistes se han servido. Usa un slice de al menos 5 chistes, `math/rand` para elegir, y un mutex para el contador (las peticiones son concurrentes).

Solución explicada:

```

package main

import (
    "encoding/json"
    "fmt"
    "math/rand"
    "net/http"
    "sync"
)

```

```

var chistes = []string{
    "Por que el programador de Go no usa paraguas? Porque maneja sus propios errores.",
    "Cuantos programadores se necesitan para cambiar un foco? Ninguno, es problema de hardware.",
    "Por que los programadores confunden Halloween con Navidad? Porque OCT 31 == DEC 25.",
    "Mi codigo no tiene bugs, solo desarrolla funcionalidades aleatorias.",
    "Hay 10 tipos de personas: las que entienden binario y las que no.",
}

type Servidor struct {
    mu      sync.Mutex
    servidos int
}

func (s *Servidor) chiste(w http.ResponseWriter, r *http.Request) {
    s.mu.Lock()
    s.servidos++
    total := s.servidos
    s.mu.Unlock()

    c := chistes[rand.Intn(len(chistes))]
    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(map[string]any{
        "chiste": c,
        "numero": total,
    })
}

func (s *Servidor) estadisticas(w http.ResponseWriter, r *http.Request) {
    s.mu.Lock()
    total := s.servidos
    s.mu.Unlock()

    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(map[string]any{"servidos": total})
}

func main() {
    s := &Servidor{}
    mux := http.NewServeMux()
    mux.HandleFunc("GET /chiste", s.chiste)
    mux.HandleFunc("GET /estadisticas", s.estadisticas)
    fmt.Println("Servidor en http://localhost:8080")
}

```



```
http.ListenAndServe(":8080", mux)
}
```

Puntos clave: el estado mutable (servidos) vive en un struct con mutex, y los handlers son métodos que lo comparten. El mutex se libera antes de escribir la respuesta (no hace falta mantenerlo bloqueado durante la escritura). Se usa `map[string]any` para respuestas rápidas sin definir structs; en APIs grandes es mejor definir structs con nombre.

Ejercicio 2: Acortador de URLs en memoria

Implementa un acortador: `POST /acortar` recibe `{ "url": "https://..." }` y regresa `{ "clave": "abc123", "corta": "http://localhost:8080/abc123" }`; `GET /{clave}` redirige con `http.Redirect` a la URL original (302) o 404 si no existe. Genera claves con un contador en base 62 o con letras aleatorias. Valida que la URL empiece con `http://` o `https://`.

Solución explicada:

```
package main

import (
    "encoding/json"
    "fmt"
    "math/rand"
    "net/http"
    "strings"
    "sync"
)

type Acortador struct {
    mu    sync.Mutex
    urls  map[string]string
}

func (a *Acortador) acortar(w http.ResponseWriter, r *http.Request) {
    var entrada struct {
        URL string `json:"url"`
    }
    if err := json.NewDecoder(r.Body).Decode(&entrada); err != nil {
        http.Error(w, "JSON invalido", http.StatusBadRequest)
        return
    }
    if !strings.HasPrefix(entrada.URL, "http://") &&
        !strings.HasPrefix(entrada.URL, "https://") {
        http.Error(w, "la url debe empezar con http:// o https://",
            http.StatusBadRequest)
    }
}
```

```

        return
    }

    const letras = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
    clave := make([]byte, 6)
    for i := range clave {
        clave[i] = letras[rand.Intn(len(letras))]
    }

    a.mu.Lock()
    a.urls[string(clave)] = entrada.URL
    a.mu.Unlock()

    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusCreated)
    json.NewEncoder(w).Encode(map[string]string{
        "clave": string(clave),
        "corta": "http://localhost:8080/" + string(clave),
    })
}

func (a *Acortador) redirigir(w http.ResponseWriter, r *http.Request) {
    clave := r.PathValue("clave")
    a.mu.Lock()
    destino, ok := a.urls[clave]
    a.mu.Unlock()
    if !ok {
        http.Error(w, "enlace no encontrado", http.StatusNotFound)
        return
    }
    http.Redirect(w, r, destino, http.StatusFound)
}

func main() {
    a := &Acortador{urls: map[string]string{}}
    mux := http.NewServeMux()
    mux.HandleFunc("POST /acortar", a.acortar)
    mux.HandleFunc("GET /{clave}", a.redirigir)
    fmt.Println("Acortador en http://localhost:8080")
    http.ListenAndServe(":8080", mux)
}

```

Puntos clave: `http.Redirect` hace todo el trabajo de la redirección (pone el encabezado `Location` y el código 302). La validación del esquema evita guardar "urls" inútiles. En un acortador real las claves se guardarían en base de datos y se verificarían colisiones; aquí 6

caracteres aleatorios de 62 posibles dan 56 mil millones de combinaciones, suficiente para el ejercicio.

Ejercicio 3: Middleware de autenticación por token

Escribe un middleware `conAuth` que revise el encabezado `Authorization`: debe ser exactamente `Bearer secreto123`; si no, responde 401 Unauthorized con mensaje JSON. Aplícalo solo a las rutas bajo `/privado/`. Incluye una ruta pública `/publico` sin protección y una privada `/privado/datos` que regrese `{"mensaje": "datos secretos"}`.

Solución explicada:

```

package main

import (
    "encoding/json"
    "fmt"
    "net/http"
    "strings"
)

func conAuth(siguiete http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        auth := r.Header.Get("Authorization")
        token, ok := strings.CutPrefix(auth, "Bearer ")
        if !ok || token != "secreto123" {
            w.Header().Set("Content-Type", "application/json")
            w.WriteHeader(http.StatusUnauthorized)
            json.NewEncoder(w).Encode(map[string]string{
                "error": "no autorizado: token invalido o ausente",
            })
            return
        }
        siguiete.ServeHTTP(w, r)
    })
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /publico", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Esto lo ve todo el mundo")
    })

    muxPrivado := http.NewServeMux()
    muxPrivado.HandleFunc("GET /privado/datos", func(w http.ResponseWriter, r
    *http.Request) {
        w.Header().Set("Content-Type", "application/json")
        json.NewEncoder(w).Encode(map[string]string{"mensaje": "datos secretos"})
    })

    // El middleware envuelve solo el mux privado.
    mux.Handle("/privado/", conAuth(http.StripPrefix("/privado", muxPrivado)))

    fmt.Println("Servidor en http://localhost:8080")
    http.ListenAndServe(":8080", mux)
}

```

Puntos clave: el middleware no llama al siguiente handler si la autenticación falla (short-circuit). Se usan dos muxes: el público va directo y el privado va envuelto; así el middleware solo protege lo que debe. `strings.CutPrefix` (Go 1.20+) separa el esquema "Bearer " del token limpiamente. En producción el token vendría de una variable de entorno, no quemado en el código, y se compararían con `subtle.ConstantTimeCompare` para evitar ataques de timing; para el ejercicio la comparación directa basta.

Ejercicio 4: Servidor de archivos estáticos + API

Combina `http.FileServer` para servir archivos estáticos desde `./estatico` en la ruta `/archivos/`, con una API JSON en `/api/saludo` que regrese `{"mensaje": "hola"}`. Crea un archivo `estatico/hola.txt` de prueba. Explica por qué se usa `http.StripPrefix`.

Solución explicada:

```

package main

import (
    "encoding/json"
    "fmt"
    "net/http"
    "os"
)

func main() {
    os.MkdirAll("estatico", 0755)
    os.WriteFile("estatico/hola.txt", []byte("Hola desde un archivo estatico\n"),
0644)

    mux := http.NewServeMux()

    // Archivos estaticos en /archivos/
    archivos := http.FileServer(http.Dir("estatico"))
    mux.Handle("GET /archivos/", http.StripPrefix("/archivos/", archivos))

    // API JSON
    mux.HandleFunc("GET /api/saludo", func(w http.ResponseWriter, r *http.Request)
{
        w.Header().Set("Content-Type", "application/json")
        json.NewEncoder(w).Encode(map[string]string{"mensaje": "hola"})
    })

    fmt.Println("Archivos en http://localhost:8080/archivos/hola.txt")
    http.ListenAndServe(":8080", mux)
}

```

Puntos clave: `http.FileServer` sirve archivos del disco de forma eficiente (maneja rangos, tipos MIME, If-Modified-Since). Sin `StripPrefix`, una petición a `/archivos/hola.txt` buscaría el archivo `estatico/archivos/hola.txt` (conserva el prefijo de la ruta); con `StripPrefix` se quita `/archivos/` y busca `estatico/hola.txt`, que es lo correcto. Este patrón es como se sirven los frontends (HTML/CSS/JS) junto a la API en un solo binario.

Ejercicio 5: Cliente que consume tu API

Escribe un programa cliente (en otro archivo `main`) que use `net/http` como cliente para: crear una tarea en la API del Ejemplo 3 (POST), listarla (GET) y marcarla completada (PUT). Maneja errores y códigos de estado, e imprime la respuesta.

Solución explicada:

```
package main

import (
    "bytes"
    "encoding/json"
    "fmt"
    "io"
    "log"
    "net/http"
)

type Tarea struct {
    ID          int    `json:"id"`
    Titulo      string `json:"titulo"`
    Completada  bool   `json:"completada"`
}

func leerRespuesta(resp *http.Response) []byte {
    defer resp.Body.Close()
    datos, err := io.ReadAll(resp.Body)
    if err != nil {
        log.Fatal("leyendo respuesta:", err)
    }
    return datos
}

func main() {
    base := "http://localhost:8080"

    // 1. Crear
    cuerpo, _ := json.Marshal(map[string]string{"titulo": "Probar el cliente"})
    resp, err := http.Post(base+"/tareas", "application/json",
bytes.NewReader(cuerpo))
    if err != nil {
        log.Fatal("POST fallo:", err)
    }
    datos := leerRespuesta(resp)
    fmt.Println("POST status:", resp.Status)
    var creada Tarea
    json.Unmarshal(datos, &creada)
    fmt.Printf("Creada: %+v\n", creada)

    // 2. Listar
    resp, err = http.Get(base + "/tareas")
}
```

```

if err != nil {
    log.Fatal("GET fallo:", err)
}
fmt.Println("GET /tareas:", string(LeerRespuesta(resp)))

// 3. Marcar completada
cuerpo, _ = json.Marshal(map[string]any{
    "titulo": creada.Titulo, "completada": true,
})
req, _ := http.NewRequest(http.MethodPut,
    fmt.Sprintf("%s/tareas/%d", base, creada.ID), bytes.NewReader(cuerpo))
req.Header.Set("Content-Type", "application/json")
resp, err = http.DefaultClient.Do(req)
if err != nil {
    log.Fatal("PUT fallo:", err)
}
fmt.Println("PUT status:", resp.Status)
fmt.Println("PUT respuesta:", string(LeerRespuesta(resp)))
}

```

Puntos clave: `http.Post` y `http.Get` son atajos para los casos simples; para PUT (o para poner encabezados) se construye la petición con `http.NewRequest` y se ejecuta con `client.Do`. Siempre cierra el cuerpo de la respuesta (`defer resp.Body.Close()`) o agotarás las conexiones. Un buen cliente revisa el código de estado antes de interpretar el cuerpo: un 404 con cuerpo de error no es una tarea válida.

Ejercicio 6: Prueba automatizada de un handler con `httptest`

Usa `net/http/httptest` para probar el handler de ``GET /tareas/{id}`` del Ejemplo 3 sin levantar un servidor real: crea la tienda, registra la ruta en un mux, haz una petición de prueba con `httptest.NewRequest`, captúrala con `httptest.NewRecorder`, y verifica el código 200 y que el JSON contenga la tarea. Incluye también el caso 404.

Solución explicada:

```

package main

import (
    "encoding/json"
    "net/http"
    "net/http/httptest"
    "testing"
)

```



```

func TestObtenerTarea(t *testing.T) {
    tienda := NuevaTienda()
    creada := tienda.crear("Probar httptest")

    mux := http.NewServeMux()
    mux.HandleFunc("GET /tareas/{id}", func(w http.ResponseWriter, r *http.Request) {
        // (mismo handler del Ejemplo 3; en código real estaría en una función)
        id, err := strconv.Atoi(r.PathValue("id"))
        if err != nil {
            http.Error(w, "id invalido", http.StatusBadRequest)
            return
        }
        tarea, ok := tienda.obtener(id)
        if !ok {
            http.Error(w, "tarea no encontrada", http.StatusNotFound)
            return
        }
        responderJSON(w, http.StatusOK, tarea)
    })

    t.Run("existe", func(t *testing.T) {
        req := httptest.NewRequest(http.MethodGet, "/tareas/1", nil)
        rec := httptest.NewRecorder()
        mux.ServeHTTP(rec, req)

        if rec.Code != http.StatusOK {
            t.Fatalf("codigo = %d; se esperaba 200", rec.Code)
        }
        var tarea Tarea
        if err := json.Unmarshal(rec.Body.Bytes(), &tarea); err != nil {
            t.Fatalf("respuesta no es JSON valido: %v", err)
        }
        if tarea.ID != creada.ID || tarea.Titulo != creada.Titulo {
            t.Errorf("tarea = %+v; se esperaba %+v", tarea, creada)
        }
    })

    t.Run("no existe", func(t *testing.T) {
        req := httptest.NewRequest(http.MethodGet, "/tareas/999", nil)
        rec := httptest.NewRecorder()
        mux.ServeHTTP(rec, req)
        if rec.Code != http.StatusNotFound {
            t.Errorf("codigo = %d; se esperaba 404", rec.Code)
        }
    })
}

```

```
}
```

Puntos clave: `httptest.NewRequest` crea una petición sintética y `NewRecorder` captura la respuesta sin red de por medio: las pruebas son rápidas y no dependen de puertos libres. Se prueba contra el mux real con la ruta real, así que también se valida el enrutamiento y `PathValue`. Nota la lección de diseño: para que esto sea cómodo, los handlers deben ser funciones separadas (o métodos) que reciban sus dependencias, no closures gigantes definidos dentro de `main`. Ese es el puente entre la Unidad 9 (testing) y esta: el código testeable se diseña testeable desde el inicio.

Ejercicio 7: Middleware de rate limiting

Un API pública necesita protegerse de clientes abusivos (o con bugs) que hacen miles de peticiones por segundo. Escribe un middleware `conLimite` que permita máximo 5 peticiones por segundo por dirección IP, usando un mapa de IP a contador con ventana de tiempo simple. Si se excede, responde 429 Too Many Requests con JSON. Usa `sync.Mutex` para el mapa.

Solución explicada:

```
package main

import (
    "encoding/json"
    "fmt"
    "net"
    "net/http"
    "sync"
    "time"
)

type Limitador struct {
    mu          sync.Mutex
    ventanas    map[string]*ventana
    max         int
    duracion    time.Duration
}

type ventana struct {
    conteo int
    inicio time.Time
}

func NuevoLimitador(max int, duracion time.Duration) *Limitador {
```

```

    return &Limitador{
        ventanas: map[string]*ventana{},
        max:      max,
        duracion: duracion,
    }
}

func (l *Limitador) permitir(ip string) bool {
    l.mu.Lock()
    defer l.mu.Unlock()
    ahora := time.Now()
    v, ok := l.ventanas[ip]
    if !ok || ahora.Sub(v.inicio) >= l.duracion {
        l.ventanas[ip] = &ventana{conteo: 1, inicio: ahora}
        return true
    }
    if v.conteo >= l.max {
        return false
    }
    v.conteo++
    return true
}

func (l *Limitador) middleware(siguiete http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        ip, _, _ := net.SplitHostPort(r.RemoteAddr)
        if !l.permitir(ip) {
            w.Header().Set("Content-Type", "application/json")
            w.WriteHeader(http.StatusTooManyRequests)
            json.NewEncoder(w).Encode(map[string]string{
                "error": "demasiadas peticiones, intenta en un momento",
            })
            return
        }
        siguiete.ServeHTTP(w, r)
    })
}

func main() {
    limitador := NuevoLimitador(5, time.Second)
    mux := http.NewServeMux()
    mux.HandleFunc("GET /recurso", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "recurso valioso")
    })
    fmt.Println("Servidor en http://localhost:8080 (max 5 peticiones/segundo por IP)")
}

```

```
http.ListenAndServe(":8080", limitador.middleware(mux))
}
```

Puntos clave: el algoritmo es "ventana fija": cada IP tiene un contador que se reinicia cada segundo. Es simple y suficiente para empezar; en producción se usan variantes más finas (token bucket, sliding window) disponibles en bibliotecas como golang.org/x/time/rate. La IP se extrae de `r.RemoteAddr` (que trae "ip:puerto") con `net.SplitHostPort`. El mapa de ventanas crece con cada IP nueva: en un servidor real habría que limpiar entradas viejas periódicamente para no agotar la memoria (una goroutine con ticker que borre ventanas expiradas). Prueba el límite con un ciclo de curl y observa los 429 después de la quinta petición.

Resumen de la unidad

En esta última unidad llevaste Go del lenguaje al producto. Lo esencial:

- `net/http` trae un servidor HTTP completo sin dependencias: `http.HandleFunc` registra handlers, `http.ListenAndServe` abre el puerto, y cada petición se atiende en su propia goroutine automáticamente.
- El handler recibe `*http.Request` (método, ruta, query, encabezados, cuerpo) y `http.ResponseWriter` (encabezados, código de estado, cuerpo), en ese orden: primero encabezados, luego `WriteHeader`, luego el cuerpo.
- Desde Go 1.22 los patrones incluyen el método ("GET /tareas") y segmentos con nombre ("/tareas/{id}" con `r.PathValue("id")`). Usa `http.NewServeMux()` propio en lugar del global.
- `encoding/json` convierte structs a JSON y de regreso: `json.Marshal/Unmarshal` para valores, `NewEncoder/NewDecoder` para streams. Solo campos exportados, tags para nombres, ``json:"-"`` para excluir, ``omitempty`` para omitir vacíos.
- Una API REST se construye con handlers delgados, un almacén con mutex para la concurrencia, validación temprana con 400, y los códigos correctos: 200, 201, 204, 400, 404, 405, 500.
- `html/template` genera páginas HTML escapando datos automáticamente contra XSS; compila las plantillas una vez con `template.Must`.
- El middleware (funciones que envuelven handlers) centraliza logging, autenticación y

recover; el recover en el borde evita que un panic tumbe el servidor.

- os.Getenv inyecta configuración sin recompilar; los secretos nunca van en el código.
- go build produce un binario estático independiente; -ldflags="-s -w" lo adelgaza; GOOS/GOARCH permiten compilar cruzado (desarrollas en Mac/Windows, despliegas en Linux).
- El Dockerfile multietapa compila en la imagen de Go y corre en una imagen mínima (alpine o scratch) con CGO_ENABLED=0: imágenes de ~20-30 MB listas para cualquier nube.
- El camino sigue con Gin/Echo, database/sql y drivers, gRPC para microservicios, y observabilidad (slog, métricas, tracing).

Checklist final antes de decir "mi servidor está listo para producción": ¿los errores se manejan con `if err != nil` y se envuelven con contexto? ¿los datos compartidos están protegidos con mutex? ¿hay middleware de logging y recover? ¿la configuración sale de variables de entorno y ningún secreto está en el código? ¿el servidor tiene timeouts y graceful shutdown? ¿hay tests con httptest y cobertura razonable? ¿el binario se compila cruzado y el Dockerfile es multietapa? ¿los encabezados de seguridad están puestos y el tráfico va por HTTPS? Si respondiste que sí a todo, no solo terminaste el libro: despliegas como un profesional.

Proyecto final integrador: API de biblioteca

Para cerrar el libro, un proyecto que usa todo: una API REST de biblioteca con libros y préstamos. Hazlo paso a paso; cada paso es un commit.

Paso 1: el modelo. Define structs `Libro` {ID, Titulo, Autor, Año, Disponible} y `Prestamo` {ID, LibroID, Usuario, Fecha} con tags JSON. Validaciones: título y autor no vacíos, año entre 1000 y el actual.

Paso 2: la tienda. Un struct `Biblioteca` con mutex, mapas de libros y préstamos, y métodos `crear/obtener/listar/actualizar/eliminar` para libros, más `prestar(id, usuario)` (marca no disponible y crea el préstamo; error si ya está prestado) y `devolver(prestamoID)`.

Paso 3: los handlers. Rutas: GET `/libros` (con query `?autor=` para filtrar), POST `/libros` (201), GET `/libros/{id}`, PUT `/libros/{id}`, DELETE `/libros/{id}` (204), POST `/libros/{id}/prestar` {"usuario": "..."}, POST `/prestamos/{id}/devolver`. Códigos correctos y mensajes de error JSON {"error": "..."}.

Paso 4: middleware. Logging de cada petición con duración, recover que regresa 500 en JSON, y autenticación simple por token para las rutas de escritura (POST/PUT/DELETE), dejando las lecturas públicas.

Paso 5: configuración y main. Puerto y token desde variables de entorno (PUERTO, TOKEN) con valores por defecto. http.Server con timeouts. Endpoint /salud.

Paso 6: testing. Table-driven tests para prestar/devolver (incluye el caso "prestar un libro ya prestado"), tests con httptest para dos handlers (uno feliz y un 404), y go test -cover apuntando a 80%+ en la lógica de la tienda.

Paso 7: despliegue. go build con -ldflags="-s -w", compilación cruzada a linux/amd64, Dockerfile multietapa, y prueba el contenedor localmente con docker run -p 8080:8080 -e PUERTO=8080.

Si completas los siete pasos, tienes un proyecto de portafolio real: súbelo a GitHub con un README que explique cómo correrlo, qué endpoints tiene y qué decisiones tomaste. Eso vale más que cualquier certificado.

Extensiones opcionales si quieres llevarlo más lejos: persiste los datos en SQLite con database/sql (reemplaza los mapas de la tienda sin tocar los handlers, y comprueba que la separación de capas funciona); agrega paginación al GET /libros como en el Ejemplo 7; protege las escrituras con el rate limiter del Ejercicio 7; y agrega un endpoint GET /metricas con expvar para ver el tráfico. Cada extensión es una línea más en tu portafolio y una historia más que contar en una entrevista ("decidí separar la tienda de los handlers para poder cambiar el almacenamiento sin reescribir la API").

Consejos para conseguir trabajo con Go

El mercado de Go es particular: hay menos vacantes que en JavaScript o Python, pero también menos competencia y suelen ser puestos mejor pagados, casi siempre en backend, infraestructura, DevOps o sistemas distribuidos. Cómo entrar:

1. El portafolio manda. Dos o tres proyectos en GitHub bien hechos (como la API de biblioteca de arriba, un CLI útil, un scraper concurrente) valen más que diez tutoriales copiados. Cada proyecto con README, tests y un Dockerfile dice "sé entregar software", no solo "sé escribir código".
2. Domina lo que preguntan en entrevistas: goroutines y canales (y cuándo NO usarlos),

el modelo de memoria básico, interfaces, manejo de errores (por qué no hay excepciones), slices vs maps por dentro, y el ciclo de vida de una petición HTTP en net/http. Todo eso está en este libro: repásalo.

3. Contribuye a un proyecto open source en Go, aunque sea documentación o tests al inicio. Aparecer en el historial de un repo conocido es una señal fuerte y aprendes a leer código ajeno, que es la mitad del trabajo real.

4. Aprende lo adyacente que piden los empleos: Docker y Kubernetes básico, SQL y diseño de bases de datos, Git a fondo, Linux en terminal, y nociones de redes (qué es un load balancer, qué significa un 502). Go rara vez se contrata solo: se contrata "Go + nube + datos".

5. En la entrevista técnica, piensa en voz alta. Los entrevistadores de Go valoran el razonamiento explícito ("aquí reviso el error porque...", "uso un mutex porque los handlers son concurrentes") tanto como la respuesta correcta. Es la misma honestidad que el lenguaje te enseñó con `if err != nil`: haz visible tu pensamiento.

6. Empieza donde hay puerta: freelances pequeños (APIs, automatizaciones, CLIs), startups que usan Go, o roles DevOps/SRE donde Go es el lenguaje de las herramientas. El primer empleo es el más difícil; después de un año con Go en producción, las puertas se abren solas.

7. Escribe sobre lo que aprendes. Un blog corto o hilos explicando algo de Go (cómo funciona un mutex, por qué los errores son valores) demuestra que entiendes a fondo y te hace visible para reclutadores. Enseñar es la prueba final de que aprendiste, y es exactamente lo que hace Lucas EGA Academy: quien enseña, domina dos veces.

8. Prepara el "live coding" practicando problemas clásicos en Go: invertir un string, detectar palíndromos, el problema de las dos sumas, implementar un LRU sencillo. No porque los uses a diario, sino porque en la entrevista necesitas escribir Go fluido bajo presión. Practica en voz alta, explicando cada decisión como lo harías con un colega: esa es la habilidad que evalúan.

Felicidades: terminaste el libro. No todos los que empiezan un libro de programación lo terminan, y tú lo hiciste. Ahora la teoría se acabó y empieza lo importante: construir. Abre tu editor, elige el proyecto final (o inventa el tuyo), y escribe código todos los días. Nos vemos en producción. Bienvenido a la comunidad gopher de Lucas EGA Academy.

EXAMEN FINAL

Llegaste al final del libro. Este examen pone a prueba todo lo que aprendiste en las 10 unidades. Tiene 50 preguntas en tres partes: opción múltiple, verdadero o falso, y preguntas abiertas con código. Intenta responderlas sin mirar atrás; después revisa tus respuestas en la sección de soluciones. Si sacas menos de 40 aciertos, repasa las unidades donde fallaste antes de considerarte listo.

Parte 1: Opción múltiple

Elige la respuesta correcta para cada pregunta.

PREGUNTA 1: ¿Quiénes crearon el lenguaje Go y en qué empresa nació?

a) Guido van Rossum en CWI b) Robert Griesemer, Rob Pike y Ken Thompson en Google c) James Gosling en Sun Microsystems d) Bjarne Stroustrup en Bell Labs

PREGUNTA 2: ¿Cuál es el comando correcto para inicializar un módulo nuevo de Go llamado "mitienda"?

a) go new mitienda b) go init mitienda c) go mod init mitienda d) go module create mitienda

PREGUNTA 3: ¿Qué valor imprime este programa?

```
package main

import "fmt"

func main() {
    var x int
    var s string
    var b bool
    fmt.Println(x, s == "", b)
}
```

a) 0 false false b) 0 true false c) Da error de compilación porque x no está inicializada d) 1 true true

PREGUNTA 4: ¿Qué hace la palabra clave iota en una declaración const?

a) Genera números aleatorios b) Crea una constante que vale el número de línea dentro del bloque const, útil para enumeraciones c) Declara una variable global d) Importa un paquete automáticamente

PREGUNTA 5: ¿Cuál es la diferencia entre `var nombre = "Ana"` y `nombre := "Ana"` dentro de una función?

a) No hay ninguna diferencia b) `var` declara sin tipo y `:=` declara con tipo explícito c) `:=` declara e infiere el tipo y solo puede usarse dentro de funciones; `var` también puede usarse a nivel de paquete d) `:=` es más lento en tiempo de ejecución

PREGUNTA 6: En la expresión `7 / 2` en Go, donde ambos operandos son `int`, el resultado es:

a) 3.5 b) 3 c) 4 d) Error de compilación

PREGUNTA 7: ¿Qué imprime este código?

```
package main

import "fmt"

func main() {
    for i := 0; i < 3; i++ {
        if i == 1 {
            continue
        }
        fmt.Print(i)
    }
}
```

a) 012 b) 02 c) 01 d) 12

PREGUNTA 8: En un `switch` de Go, ¿qué ocurre si un `case` termina sin `break`?

a) Da error de compilación b) Continúa automáticamente al siguiente `case` como en C c) Sale del `switch` automáticamente; solo continúa si se usa `fallthrough` d) Repite el mismo `case` en un bucle

PREGUNTA 9: ¿Qué valores devuelve esta función?

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

a) Solo el cociente b) El cociente y un error (o nil si no hubo error) c) Solo el error d) No compila porque faltan los nombres de los retornos

PREGUNTA 10: ¿En qué orden se ejecutan las llamadas defer dentro de una función?

a) En el orden en que se escribieron (FIFO) b) En orden inverso al que se escribieron (LIFO) c) En orden alfabético d) Todas al mismo tiempo

PREGUNTA 11: Dado `s := []int{1, 2, 3}; s = append(s, 4)`. ¿Qué afirmación es correcta?

a) s sigue siendo {1, 2, 3} porque los slices son inmutables b) append puede devolver un slice nuevo con otro arreglo subyacente; por eso se reasigna a s c) Da error porque el slice tiene tamaño fijo 3 d) append modifica el arreglo original garantizado

PREGUNTA 12: ¿Cómo se verifica si una clave existe en un mapa m?

a) if m["clave"] != nil b) valor, ok := m["clave"]; if ok { ... } c) if exists(m, "clave") d) if m.has("clave")

PREGUNTA 13: ¿Cuál es la diferencia entre un método con receptor de valor y uno con receptor de puntero?

a) No hay diferencia b) El receptor de puntero puede modificar el struct original; el de valor trabaja sobre una copia c) El receptor de valor es más rápido siempre d) Solo los receptores de valor pueden usarse en interfaces

PREGUNTA 14: ¿Cómo se "implementa" una interfaz en Go?

a) Con la palabra clave implements b) Automáticamente: si un tipo tiene todos los métodos de la interfaz, la satisface sin declararlo c) Registrando el tipo con go register d) Heredando de la interfaz con la palabra extends

PREGUNTA 15: ¿Qué es una goroutine?

a) Un hilo del sistema operativo b) Una función que se ejecuta concurrentemente, gestionada por el runtime de Go y muy barata de crear c) Un tipo de canal con buffer d) Una librería para redes

PREGUNTA 16: `ch := make(chan int, 2)`. ¿Qué significa el 2?

a) Que el canal solo puede usarse 2 veces b) Que el canal tiene un buffer con capacidad para 2 valores sin bloquear al emisor c) Que el canal acepta solo el número 2 d) Que hay 2 receptores obligatorios

PREGUNTA 17: ¿Para qué sirve `select` en Go?

a) Para elegir elementos de un slice b) Para esperar en múltiples operaciones de canales y actuar en la primera que esté lista c) Para ordenar mapas d) Para seleccionar el sistema operativo de compilación

PREGUNTA 18: ¿Cuál es la forma idiomática de manejar un error retornado por una función?

a) `try { f() } catch (e) { ... }` b) `if err != nil { manejar el error; return }` c) Ignorarlo siempre con `_` d) `f() or die()`

PREGUNTA 19: ¿Qué hace `go test -cover`?

a) Cubre el código con comentarios b) Ejecuta las pruebas y reporta el porcentaje de código cubierto por los tests c) Compila sin ejecutar pruebas d) Borra los archivos de prueba

PREGUNTA 20: En `net/http`, ¿qué hace `http.HandleFunc("/saludo", miHandler)`?

a) Inicia el servidor en el puerto saludo b) Registra la función `miHandler` para atender peticiones a la ruta `/saludo` c) Envía una petición HTTP a `/saludo` d) Cierra el servidor

Parte 2: Verdadero o falso

Indica si cada afirmación es verdadera (V) o falsa (F).

PREGUNTA 21: Go es un lenguaje interpretado como Python. (V/F)

PREGUNTA 22: Una variable declarada y no usada provoca un error de compilación en Go. (V/F)

PREGUNTA 23: Los arreglos en Go tienen tamaño fijo que forma parte de su tipo: `[3]int` y `[4]int` son tipos distintos. (V/F)

PREGUNTA 24: Se puede comparar dos slices con `==` para saber si tienen los mismos elementos. (V/F)

PREGUNTA 25: En Go, los identificadores que empiezan con mayúscula son exportados (visibles desde otros paquetes). (V/F)

PREGUNTA 26: Una función puede devolver más de un valor en Go. (V/F)

PREGUNTA 27: El operador ++ puede usarse dentro de expresiones, por ejemplo `x := y++`. (V/F)

PREGUNTA 28: Un struct puede "heredar" de otro struct como en Java con `extends`. (V/F)

PREGUNTA 29: Un valor nil de una interfaz es igual a una interfaz que guarda un puntero nil. (V/F)

PREGUNTA 30: Enviar a un canal sin buffer bloquea hasta que otro goroutine reciba el valor. (V/F)

PREGUNTA 31: `close(ch)` se usa para cerrar un canal cuando ya no se enviarán más valores. (V/F)

PREGUNTA 32: En Go se recomienda usar `panic` para errores normales como "archivo no encontrado". (V/F)

PREGUNTA 33: `errors.Is` permite saber si un error envuelto con `%w` contiene un error centinela específico. (V/F)

PREGUNTA 34: Los archivos de prueba en Go deben terminar en `_test.go` y las funciones de prueba empezar con `Test`. (V/F)

PREGUNTA 35: Con `GOOS=windows GOARCH=amd64 go build` se puede compilar en Linux un ejecutable para Windows. (V/F)

Parte 3: Preguntas abiertas con código

Responde por escrito o escribiendo código Go.

PREGUNTA 36: Explica con tus palabras la diferencia entre un arreglo y un slice en Go, y escribe un ejemplo donde un `append` provoque que el slice cambie de arreglo subyacente (usa `len` y `cap` para demostrarlo).

PREGUNTA 37: Escribe una función `EsPrimo(n int) bool` que indique si un número es primo, y un programa que imprima los primos del 1 al 50 usando esa función.

PREGUNTA 38: Define un struct Rectangulo con campos Ancho y Alto (float64) y los métodos Area() y Perimetro(). Luego define la interfaz Figura con esos dos métodos y escribe una función que reciba un slice de Figura e imprima el área total. Incluye también un struct Circulo que implemente la interfaz.

PREGUNTA 39: Escribe un programa que lance 5 goroutines, cada una calculando el cuadrado de un número del 1 al 5, enviando los resultados por un canal. El programa principal debe recibir e imprimir los 5 resultados usando sync.WaitGroup para esperar a que terminen.

PREGUNTA 40: Escribe una función LeerArchivo(ruta string) (string, error) que lea un archivo de texto y devuelva su contenido. Debe envolver los errores con fmt.Errorf usando %w y un mensaje de contexto. Explica por qué se usa %w en lugar de %v.

PREGUNTA 41: Dado el siguiente código con un bug, encuéntralo y corrígelo:

```
package main

import "fmt"

func main() {
    m := make(map[string]int)
    m["a"] = 1
    m["b"] = 2
    for i := 0; i < 3; i++ {
        delete(m, "a")
        fmt.Println(m["a"] + m["c"])
    }
}
```

Explica qué imprime el código original y por qué, y entrega la versión corregida si la intención era sumar los valores de "a" y "b".

PREGUNTA 42: Explica qué es un data race (condición de carrera) y escribe un ejemplo con dos goroutines incrementando un contador, protegido una vez con sync.Mutex y otra vez con un canal. ¿Con qué comando detectarías la carrera en la versión sin protección?

PREGUNTA 43: Escribe un table-driven test para la función Multiplicar(a, b int) int con al menos 4 casos, incluyendo un caso con cero y uno con números negativos.

PREGUNTA 44: Escribe un servidor HTTP mínimo con net/http que tenga dos rutas: GET /saludo que responda "Hola, mundo" y POST /eco que lea el cuerpo de la petición y lo devuelva tal cual. Usa las rutas con método de Go 1.22+ ("GET /saludo").

PREGUNTA 45: Explica la diferencia entre `fmt.Errorf("...: %v", err)` y `fmt.Errorf("...: %w", err)`. ¿En qué caso necesitarías `errors.As`? Da un ejemplo.

PREGUNTA 46: Escribe el struct `Usuario` con campos `Nombre string`, `Edad int` y `Email string`, con las etiquetas JSON correctas (`nombre`, `edad`, `email` en minúsculas). Luego escribe código que convierta un `Usuario` a JSON y código que convierta ese JSON de vuelta a un `Usuario`, manejando errores.

PREGUNTA 47: ¿Qué imprime este programa y por qué?

```
package main

import "fmt"

func main() {
    x := 10
    f := func() {
        x++
        fmt.Println(x)
    }
    f()
    f()
    fmt.Println(x)
}
```

PREGUNTA 48: Diseña (pseudocódigo o Go) un worker pool: 3 workers que leen números de un canal de trabajos, calculan su cubo y envían el resultado a un canal de resultados. El main envía los números del 1 al 10, cierra el canal de trabajos y recoge los 10 resultados.

PREGUNTA 49: Explica con tus palabras por qué en Go se dice "acepta interfaces, devuelve structs" (accept interfaces, return structs). Da un ejemplo de una función que acepte `io.Writer` en lugar de `*os.File` y explica la ventaja.

PREGUNTA 50: Proyecto integrador: describe paso a paso cómo construirías una API REST de una lista de tareas (TODO) en Go con net/http y encoding/json, incluyendo: estructura del proyecto, struct `Tarea`, rutas (GET /tareas, POST /tareas, GET /tareas/{id}, PUT /tareas/{id}, DELETE /tareas/{id}), dónde guardarías los datos en memoria de forma segura para concurrencia y cómo probarías cada endpoint.

Respuestas y soluciones

RESPUESTA 1: b) Robert Griesemer, Rob Pike y Ken Thompson en Google. Go nació en Google en 2007 y se liberó en 2009. Ken Thompson es además uno de los creadores de Unix y del lenguaje C.

RESPUESTA 2: c) `go mod init mitienda`. Crea el archivo `go.mod` que declara el módulo y su ruta. Sin módulo, los imports de tu propio código se complican.

RESPUESTA 3: b) `0 true false`. En Go toda variable tiene un "valor cero": `0` para `int`, `""` para `string` (por eso `s == ""` es `true`) y `false` para `bool`. No da error: el valor cero es una característica, no un descuido.

RESPUESTA 4: b) `iota` vale el número de línea dentro del bloque `const` (empezando en `0`) y se usa para enumeraciones: `const ( Domingo = iota; Lunes; Martes )` genera `0, 1, 2`.

RESPUESTA 5: c) `:=` declara e infiere el tipo y solo funciona dentro de funciones; `var` funciona también a nivel de paquete. Además `:=` exige que al menos una variable a la izquierda sea nueva (permite re-declarar las demás).

RESPUESTA 6: b) `3`. La división entre enteros trunca hacia cero. Si quieres `3.5`, convierte: `float64(7) / 2`.

RESPUESTA 7: b) `02`. Con `i=1` el `continue` salta la impresión y pasa a la siguiente iteración. Se imprimen `0` y `2`.

RESPUESTA 8: c) Go no tiene caída implícita entre `case`s; cada `case` termina automáticamente a menos que escribas `fallthrough` explícito. Es una de las trampas clásicas evitadas respecto a C.

RESPUESTA 9: b) El cociente y un error (o `nil` si no hubo error). Es el patrón `value, err`: el llamador debe revisar `if err != nil` antes de usar el valor.

RESPUESTA 10: b) LIFO: la última llamada `defer` registrada es la primera en ejecutarse. Por eso `defer` se usa para "deshacer" en orden inverso: abrir, luego diferir cerrar.

RESPUESTA 11: b) `append` puede devolver un slice distinto (con otro arreglo subyacente si se agotó la capacidad); por eso el resultado siempre se reasigna: `s = append(s, 4)`. Los slices no son inmutables.

RESPUESTA 12: b) `valor, ok := m["clave"]`. La "coma-ok" distingue entre "la clave no existe" y "existe con el valor cero". Leer una clave inexistente devuelve el valor cero sin error, lo que puede confundir.

RESPUESTA 13: b) El receptor de puntero (func (c *Cuenta) Depositar()) modifica el original; el receptor de valor trabaja sobre una copia. Regla práctica: si un método modifica, usa puntero en todos los métodos del tipo para ser consistente.

RESPUESTA 14: b) Implementación implícita: no se declara nada. Si tu tipo tiene los métodos de la interfaz, la satisface. Esto permite que tipos de paquetes distintos satisfagan interfaces sin conocerse.

RESPUESTA 15: b) Una goroutine es una función ejecutándose concurrentemente, multiplexada por el runtime sobre hilos del SO. Crear miles de goroutines es barato; crear miles de hilos del SO no.

RESPUESTA 16: b) El 2 es la capacidad del buffer: el emisor puede enviar 2 valores sin bloquearse aunque nadie los haya recibido todavía. El tercer envío bloquearía hasta que alguien reciba.

RESPUESTA 17: b) select espera en múltiples canales y ejecuta el case del primero que esté listo. Con time.After se implementan timeouts y con default se evita el bloqueo.

RESPUESTA 18: b) if err != nil { ... return }. En Go los errores son valores que se revisan explícitamente; no hay try/catch. Ignorar errores con _ solo es aceptable cuando estás seguro de que no importan.

RESPUESTA 19: b) Ejecuta las pruebas y muestra el porcentaje de sentencias cubiertas por los tests. Se combina: go test -cover ./...

RESPUESTA 20: b) Registra miHandler para la ruta /saludo. Después hay que iniciar el servidor con http.ListenAndServe(":8080", nil).

RESPUESTA 21: F. Go es compilado: go build genera un ejecutable nativo. Es compilado y de tipado estático.

RESPUESTA 22: V. "declared and not used" es error de compilación. Es una decisión deliberada para mantener el código limpio.

RESPUESTA 23: V. El tamaño es parte del tipo en arreglos. Por eso casi siempre se prefieren slices, cuyo tamaño es dinámico.

RESPUESTA 24: F. Los slices no se pueden comparar con == (solo con nil). Para comparar elementos hay que hacerlo manualmente o con slices.Equal del paquete slices.

RESPUESTA 25: V. La mayúscula inicial exporta el identificador: Nombre es público, nombre es privado del paquete. No hay public/private como palabras clave.

RESPUESTA 26: V. `func dividir(a, b float64) (float64, error)` devuelve dos valores. El patrón (valor, error) es el más común de Go.

RESPUESTA 27: F. En Go `++` y `--` son sentencias, no expresiones: `x++` es válido como línea, pero `x := y++` no compila. Otra diferencia deliberada con C.

RESPUESTA 28: F. Go no tiene herencia. Se usa composición con structs embebidos (embedding): el struct externo "promociona" los métodos del interno. "Go no tiene herencia porque la composición es más flexible".

RESPUESTA 29: F. Una interfaz que guarda un puntero nil NO es nil: la interfaz tiene tipo (el puntero) aunque el valor sea nil. Por eso `if err == nil` puede fallar cuando err guarda un puntero nil. Trampa clásica del examen y de la vida real.

RESPUESTA 30: V. Sin buffer, el envío bloquea hasta que haya un receptor listo (comunicación síncrona). Es el famoso "no te comuniques compartiendo memoria; comparte memoria comunicándote".

RESPUESTA 31: V. `close(ch)` indica que no habrá más envíos. Solo el emisor debe cerrar. Recibir de un canal cerrado devuelve el valor cero inmediatamente; enviar a un canal cerrado provoca panic.

RESPUESTA 32: F. panic es para errores irre recuperables de programador, no para errores esperados. "Archivo no encontrado" se devuelve como error y el llamador decide. Regla: las bibliotecas casi nunca deben hacer panic.

RESPUESTA 33: V. `fmt.Errorf("...: %w", err)` envuelve; `errors.Is(err, centinela)` desempaqueta la cadena buscando el centinela. Con `%v` se pierde la cadena.

RESPUESTA 34: V. Convención estricta: `foo_test.go`, `func TestFoo(t *testing.T)`, y `go test` los descubre automáticamente.

RESPUESTA 35: V. Go compila cruzado sin herramientas extra: basta fijar GOOS y GOARCH. El binario resultante es estático y autocontenido.

RESPUESTA 36: Un arreglo tiene tamaño fijo que es parte de su tipo (`[4]int`), se pasa por valor (se copia) y casi no se usa directamente. Un slice es una vista ligera (puntero al arreglo subyacente, longitud y capacidad) sobre un arreglo; se pasa barato y su tamaño es dinámico. Ejemplo de crecimiento:

```
package main

import "fmt"

func main() {
    s := make([]int, 3, 3)
    fmt.Println(len(s), cap(s)) // 3 3
    s = append(s, 4)
    fmt.Println(len(s), cap(s)) // 4 6 (capacidad duplicada, nuevo arreglo)
    s2 := s[:2]
    s2[0] = 99
    fmt.Println(s[0]) // 99: s2 comparte el arreglo con s
}
```

Al hacer `append` con capacidad llena, Go reserva un arreglo nuevo (normalmente el doble) y copia los elementos: el slice resultante apunta a otro arreglo. La última parte muestra que dos slices pueden compartir el arreglo subyacente: modificar uno afecta al otro.

RESPUESTA 37:

```

package main

import "fmt"

func EsPrimo(n int) bool {
    if n < 2 {
        return false
    }
    for i := 2; i*i <= n; i++ {
        if n%i == 0 {
            return false
        }
    }
    return true
}

func main() {
    for i := 1; i <= 50; i++ {
        if EsPrimo(i) {
            fmt.Print(i, " ")
        }
    }
    fmt.Println()
}

```

Imprime: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47. Nota: basta probar divisores hasta la raíz cuadrada ($i*i \leq n$); probar hasta $n-1$ funciona pero es desperdicio.

RESPUESTA 38:

```

package main

import (
    "fmt"
    "math"
)

type Figura interface {
    Area() float64
    Perimetro() float64
}

type Rectangulo struct {
    Ancho, Alto float64
}

func (r Rectangulo) Area() float64      { return r.Ancho * r.Alto }
func (r Rectangulo) Perimetro() float64 { return 2 * (r.Ancho + r.Alto) }

type Circulo struct {
    Radio float64
}

func (c Circulo) Area() float64      { return math.Pi * c.Radio * c.Radio }
func (c Circulo) Perimetro() float64 { return 2 * math.Pi * c.Radio }

func AreaTotal(figuras []Figura) float64 {
    var total float64
    for _, f := range figuras {
        total += f.Area()
    }
    return total
}

func main() {
    figuras := []Figura{
        Rectangulo{Ancho: 3, Alto: 4},
        Circulo{Radio: 2},
    }
    fmt.Println(AreaTotal(figuras)) // 12 + 12.566... = 24.566...
}

```

Rectangulo y Circulo satisfacen Figura implícitamente, sin declarar nada. AreaTotal funciona con cualquier figura futura: ese es el polimorfismo en Go.

RESPUESTA 39:

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    resultados := make(chan int, 5)
    var wg sync.WaitGroup

    for i := 1; i <= 5; i++ {
        wg.Add(1)
        go func(n int) {
            defer wg.Done()
            resultados <- n * n
        }(i)
    }

    wg.Wait()
    close(resultados)

    for r := range resultados {
        fmt.Println(r)
    }
}
```

Puntos clave: se pasa `i` como parámetro al goroutine para evitar la trampa de la variable de bucle compartida; el canal con buffer evita bloqueos; se cierra el canal después de `wg.Wait()` para que el `for-range` termine. El orden de impresión no está garantizado.

RESPUESTA 40:

```

package main

import (
    "fmt"
    "os"
)

func LeerArchivo(ruta string) (string, error) {
    datos, err := os.ReadFile(ruta)
    if err != nil {
        return "", fmt.Errorf("no se pudo leer el archivo %s: %w", ruta, err)
    }
    return string(datos), nil
}

```

Se usa %w porque envuelve el error original conservando la cadena: el llamador puede usar errors.Is(err, os.ErrNotExist) para saber si el archivo no existía. Con %v el mensaje se vería igual, pero la información estructurada se perdería y errors.Is no funcionaría.

RESPUESTA 41: El bug: m["c"] no existe, así que vale 0 (valor cero), no provoca error. Pero hay algo peor: en la primera iteración se borra "a", así que m["a"] vale 0 desde entonces. El programa imprime "0" tres veces (0+0). Si la intención era sumar "a" y "b", la versión corregida:

```

package main

import "fmt"

func main() {
    m := make(map[string]int)
    m["a"] = 1
    m["b"] = 2
    for i := 0; i < 3; i++ {
        fmt.Println(m["a"] + m["b"]) // 3, tres veces
    }
}

```

Lección: leer una clave inexistente en un mapa no da error, devuelve el valor cero. Para código serio, usa la coma-ok: v, ok := m["c"].

RESPUESTA 42: Un data race ocurre cuando dos goroutines acceden a la misma variable concurrentemente y al menos una escribe, sin sincronización. El resultado es impredecible.

Sin protección (con carrera):

```
var contador int
var wg sync.WaitGroup
for i := 0; i < 2; i++ {
    wg.Add(1)
    go func() {
        defer wg.Done()
        for j := 0; j < 1000; j++ {
            contador++ // carrera: leer-modificar-escribir no es atómico
        }
    }()
}
wg.Wait()
fmt.Println(contador) // probablemente menos de 2000
```

Se detecta con: `go run -race main.go`.

Con mutex:

```
var mu sync.Mutex
// dentro del goroutine:
for j := 0; j < 1000; j++ {
    mu.Lock()
    contador++
    mu.Unlock()
}
```

Con canal:

```
ch := make(chan int)
go func() {
    total := 0
    for j := 0; j < 1000; j++ {
        total++
    }
    ch <- total
}()
// el main suma lo recibido; solo un goroutine toca cada variable
```

RESPUESTA 43:

```

package main

import "testing"

func Multiplicar(a, b int) int { return a * b }

func TestMultiplicar(t *testing.T) {
    casos := []struct {
        nombre    string
        a, b       int
        esperado   int
    }{
        {"positivos", 3, 4, 12},
        {"por cero", 7, 0, 0},
        {"cero por numero", 0, 9, 0},
        {"negativos", -3, -5, 15},
        {"mixto", -2, 6, -12},
    }
    for _, c := range casos {
        t.Run(c.nombre, func(t *testing.T) {
            if got := Multiplicar(c.a, c.b); got != c.esperado {
                t.Errorf("Multiplicar(%d, %d) = %d, se esperaba %d",
                    c.a, c.b, got, c.esperado)
            }
        })
    }
}

```

El table-driven test es el estilo idiomático de Go: una tabla de casos y un solo bucle. `t.Run` da subtests con nombre, visibles con `go test -v`.

RESPUESTA 44:


```

package main

import (
    "io"
    "net/http"
)

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /saludo", func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("Hola, mundo"))
    })
    mux.HandleFunc("POST /eco", func(w http.ResponseWriter, r *http.Request) {
        cuerpo, _ := io.ReadAll(r.Body)
        w.Write(cuerpo)
    })
    http.ListenAndServe(":8080", mux)
}

```

Las rutas con método ("GET /saludo") llegaron en Go 1.22 y eliminan la necesidad de revisar `r.Method` a mano. Se prueba con: `curl localhost:8080/saludo` y `curl -X POST -d "hola" localhost:8080/eco`.

RESPUESTA 45: Con `%v` el error original se convierte en texto y se pierde la cadena de envoltura; con `%w` se conserva, de modo que `errors.Is` y `errors.As` pueden recorrerla. `errors.As` se necesita cuando quieres extraer un tipo de error concreto de la cadena, no solo comparar un centinela. Ejemplo:

```

type ErrorValidacion struct{ Campo string }
func (e *ErrorValidacion) Error() string { return "campo invalido: " + e.Campo }

func procesar() error {
    return fmt.Errorf("fallo al procesar: %w", &ErrorValidacion{Campo: "email"})
}

func main() {
    err := procesar()
    var ev *ErrorValidacion
    if errors.As(err, &ev) {
        fmt.Println("El campo con error fue:", ev.Campo)
    }
}

```

errors.Is compara contra un valor centinela (==); errors.As extrae un tipo de la cadena para inspeccionar sus campos.

RESPUESTA 46:

```
package main

import (
    "encoding/json"
    "fmt"
)

type Usuario struct {
    Nombre string `json:"nombre"`
    Edad   int    `json:"edad"`
    Email  string `json:"email"`
}

func main() {
    u := Usuario{Nombre: "Ana", Edad: 30, Email: "ana@ejemplo.com"}

    datos, err := json.Marshal(u)
    if err != nil {
        panic(err)
    }
    fmt.Println(string(datos))
    // {"nombre":"Ana","edad":30,"email":"ana@ejemplo.com"}

    var u2 Usuario
    if err := json.Unmarshal(datos, &u2); err != nil {
        panic(err)
    }
    fmt.Printf("%+v\n", u2)
}
```

Las etiquetas (tags) json:"nombre" controlan el nombre en el JSON. Sin tags, Go usaría "Nombre", "Edad", "Email". Solo los campos exportados (mayúscula) se serializan; un campo en minúscula se ignora silenciosamente.

RESPUESTA 47: Imprime:

```
11
12
12
```

f es un closure que captura la variable x del ámbito externo (no una copia). Cada llamada a f() incrementa la misma x: 10 -> 11 (imprime 11), 11 -> 12 (imprime 12). El Println final muestra 12. Los closures capturan variables por referencia, y eso es lo que los hace útiles para contadores y generadores.

RESPUESTA 48:

```
package main

import (
    "fmt"
    "sync"
)

func worker(id int, trabajos <-chan int, resultados chan<- int, wg *sync.WaitGroup)
{
    defer wg.Done()
    for n := range trabajos {
        resultados <- n * n * n
    }
}

func main() {
    trabajos := make(chan int)
    resultados := make(chan int, 10)
    var wg sync.WaitGroup

    for w := 1; w <= 3; w++ {
        wg.Add(1)
        go worker(w, trabajos, resultados, &wg)
    }

    for n := 1; n <= 10; n++ {
        trabajos <- n
    }
    close(trabajos)

    wg.Wait()
    close(resultados)

    for r := range resultados {
        fmt.Println(r)
    }
}
```

Los canales direccionales (<-chan de solo lectura, chan<- de solo escritura) documentan el rol de cada canal. Cerrar trabajos hace que los for-range de los workers terminen; wg.Wait() espera a los 3; luego se cierra resultados para que el main termine de recoger. Es el patrón worker pool, la base de casi todo procesamiento concurrente en Go.

RESPUESTA 49: "Acepta interfaces, devuelve structs" significa: los parámetros de tus funciones deben ser lo más generales posible (interfaces pequeñas) y los retornos lo más concretos posible (structs). Ejemplo:

```
// Mal: solo acepta archivos reales
func GuardarLog(f *os.File, msg string)

// Bien: acepta cualquier cosa donde se pueda escribir
func GuardarLog(w io.Writer, msg string) {
    fmt.Fprintln(w, msg)
}
```

Ventajas: la función se puede probar pasando un bytes.Buffer en lugar de crear archivos reales; sirve para archivos, conexiones de red, respuestas HTTP (http.ResponseWriter implementa io.Writer) o cualquier writer futuro sin cambiar la firma. Del lado del retorno, devolver un struct concreto (no una interfaz) le da al llamador acceso a todos sus métodos y evita la trampa del "nil con tipo" en interfaces.

RESPUESTA 50: Paso a paso:

1. Estructura del proyecto:

```
mitodo/
  go.mod
  main.go      (servidor y rutas)
  tareas.go    (struct Tarea y almacenamiento)
  tareas_test.go (pruebas de los handlers)
```

2. El struct:

```
type Tarea struct {
    ID      int    `json:"id"`
    Titulo  string `json:"titulo"`
    Hecha   bool   `json:"hecha"`
}
```

3. Almacenamiento seguro para concurrencia: un struct `Tienda` con un mapa y un `sync.RWMutex` (los handlers de `net/http` corren cada uno en su goroutine, así que el mapa se accede concurrentemente):

```
type Tienda struct {  
    mu      sync.RWMutex  
    tareas  map[int]Tarea  
    siguiente int  
}
```

Métodos: Crear, Obtener, Listar, Actualizar, Borrar, cada uno con `Lock/Unlock` o `RLock/RUnlock`.

4. Rutas con Go 1.22+:

```
mux.HandleFunc("GET /tareas", h.listar)  
mux.HandleFunc("POST /tareas", h.crear)  
mux.HandleFunc("GET /tareas/{id}", h.obtener)  
mux.HandleFunc("PUT /tareas/{id}", h.actualizar)  
mux.HandleFunc("DELETE /tareas/{id}", h.borrar)
```

Cada handler: decodifica JSON del body (`json.NewDecoder(r.Body).Decode`), valida, opera en la tienda y responde con `json.NewEncoder(w).Encode` más el código de estado correcto (201 Created al crear, 404 si no existe, 400 si el JSON es inválido). El id se lee con `r.PathValue("id")` y se convierte con `strconv.Atoi`.

5. Pruebas: table-driven tests para la Tienda (crear/listar/borrar) y tests de handlers con `net/http/httptest` (`httptest.NewRequest` + `httptest.NewRecorder`), verificando códigos de estado y el JSON de respuesta. Se corren con `go test -v -cover`.
6. Extensiones posibles: persistencia en SQLite con `database/sql`, autenticación con middleware, y graceful shutdown con `signal.Notify`.

Cómo calificarte

- 45 a 50 aciertos: nivel profesional. Puedes construir y desplegar proyectos reales en Go.
- 40 a 44: muy bien. Repasa las preguntas que fallaste y sus unidades.
- 30 a 39: base sólida pero con huecos. Relee las unidades de tus errores y haz de nuevo sus prácticas.

- Menos de 30: vuelve a leer el libro con calma, escribiendo cada ejemplo a mano. Go premia la práctica constante.

Sea cual sea tu resultado, recuerda: el examen mide un momento, no tu potencial. Los programadores profesionales consultan documentación todos los días. Lo que importa es que entiendas los conceptos y sepas dónde buscar los detalles.

Gracias por llegar hasta aquí. Nos vemos en lucasegaacademy.com con el siguiente curso.

Curso Completo de Go

Libro de programación de Lucas EGA Academy: explicaciones extensas unidad por unidad, práctica guiada y examen final para dominar el lenguaje desde cero.

Colibrí Editorial EGA

lucasegaacademy.com