Boring Go

K19G

2026-09-13

Table of contents

Boring Go

A linear guide to writing Go the way it actually works in long-lived projects: **simple, explicit, and a little boring**. Clarity beats novelty. Readability beats magic. The boring default is usually the one that still compiles, still tests, and still makes sense six months later.

Baseline: Go **1.27** · toolchain `go` (modules, `gofmt`, `go vet`, `go test`, `govulncheck`). Completely self-contained. You do not need any other title in this library.

Who this is for

- People who have never written Go and want a complete path, not a bag of tricks.
- Working developers who can already `go run` something, but want better defaults.
- Anyone building services, CLIs, or tools who is tired of clever abstractions.

You do not need prior Go. You do need a terminal, a text editor, and a willingness to type the examples.

What "boring" means here

Go is a small language on purpose. Well-written Go looks repetitive. Names are long. Errors are checked. Interfaces are small. Generics appear when they remove duplication, not when they impress a reviewer.

That is not a limitation. It is the product.

Each chapter teaches **the default that lasts**, then shows the trap that looks smarter for a week.

How to read

1. Read a chapter.
2. Copy the program into `main.go` (or the filename in the comment).
3. Run it with `go run .` (or `go test` when the chapter says so).
4. Change one thing. Run it again.
5. Do the **Try this** exercises before moving on.

The book is designed to be read **front to back**. You can jump to errors, testing, or concurrency if you already write Go — but the early chapters define the vocabulary later chapters assume.

How examples work

Every Go listing is a **complete, runnable program** (or a complete test file). Nothing is a fragment that "you should imagine the rest of."

```go
// hello.go
package main

import "fmt"

func main() {
    fmt.Println("hello, desk")
}
```

```
go run hello.go
```

```
hello, desk
```

When a chapter needs more than one file (`go.mod` plus `main.go`, or `_test.go` beside the code), both files are shown **in the chapter**. There is no separate examples directory.

A small **desk** (orders, tickets, shifts) shows up across chapters so types, methods, errors, and tests feel like one codebase — but each listing still runs on its own.

Map of the book

Part	Folder	What you leave with
1	`01-foundations`	Why Go exists, install, packages, a language tour
2	`02-toolchain`	`go` commands, format/vet/doc, modules, workspaces, tool pins
3	`03-types-and-values`	Predeclared types, zeros, constants, scope
4	`04-control-flow`	`if`, `switch`, `for`, `range`
5	`05-composite-types`	Arrays, slices, strings, maps, structs
6	`06-functions-and-methods`	Functions, closures, methods, receivers
7	`07-memory`	Stack/heap, pointers, sharing, GC
8	`08-interfaces`	Implicit interfaces, design rules, assertions
9	`09-generics`	Type parameters and Go 1.27 generic methods, used where they earn their keep
10	`10-errors`	Errors as values, wrapping, panic/recover
11	`11-testing`	Table tests, coverage, fuzz, integration
12	`12-concurrency`	Goroutines, channels, `sync`, `context`

Part	Folder	What you leave with
13	`13-stdlib`	`time`, `io`, encoding, HTTP, `log/slog`, `strings`/`strconv`, `slices`/`sort`, `os`/`flag`, `regexp`, `maps`, `iter`, `signal`, `crypto`, `filepath`, `database/sql`, `json/v2`, `omitzero`, `uuid`
14	`14-shipping`	Vet/vuln, generate/embed, release, docs as API
15	`15-long-term`	Reflect, `unsafe`, cgo, pprof, design that lasts
99	`99-appendices`	Glossary and command cheat sheet

What this book is not

- Not a framework tutorial.
- Not a catalog of every standard-library package.
- Not a collection of interview puzzles.
- Not a runtime-internals encyclopedia.

It is a guide to writing Go that stays understandable when the requirements change.

Formats

HTML, PDF, and EPUB via the library portal.

Foundations

Why Go Exists

Why Go Exists

Go exists because a small group of people at Google were tired of waiting on compiles, tired of languages that rewarded cleverness, and tired of concurrency that looked like an accident. The boring default in this book is the same default Go was built for: **code a stranger can read on Monday morning**.

Mental model

Go is a compiled, statically typed language with garbage collection, a tiny feature set, and a toolchain that is part of the language. You trade inheritance, exceptions, and operator overloading for:

- fast builds
- one obvious way to layout a package
- goroutines that are cheap enough to use as a unit of work
- errors you can see in the function signature

That trade is the point. If you keep reaching for magic, you will dislike Go. If you like software that stays boring as it grows, you will like it.

Worked examples

Case 1: A complete program, no ceremony

Save as `hello.go`. This is the smallest useful Go program: a `package main`, a `main` function, and a print.

```
// hello.go
package main

import "fmt"

func main() {
    fmt.Println("desk is open")
}
```

Run:

```
go run hello.go
```

Output:

```
desk is open
```

`go run` compiles a temporary binary and executes it. There is no separate "interpreter mode." What you run is what you ship, minus the install path.

Case 2: Explicit control flow, explicit failure

Other languages hide failure in exceptions. Go puts it in the return value. The next program takes a list of table numbers and "opens" each one. If a number is nonsense, the function returns an error and the caller prints it. Nothing unwinds the stack in secret.

```
// open_tables.go
package main

import (
    "fmt"
    "os"
)

func openTable(n int) error {
    if n <= 0 {
        return fmt.Errorf("table %d: number must be positive", n)
    }
    fmt.Printf("opened table %d\n", n)
    return nil
}

func main() {
    tables := []int{3, 0, 11}
    for _, n := range tables {
        if err := openTable(n); err != nil {
            fmt.Fprintln(os.Stderr, err)
            os.Exit(1)
        }
    }
}
```

Run:

```
go run open_tables.go
```

Output (stdout then the process exits 1):

```
opened table 3
table 0: number must be positive
```

The `if err != nil` line is repetitive on purpose. You see every failure path when you read the function.

Case 3: Concurrency without a thread pool class

A goroutine is a function the runtime can run alongside others. Channels pass values between them. This program starts two workers that send a line each, then the main goroutine prints what it receives.

```
// two_shifts.go
package main

import "fmt"

func shift(name string, out chan<- string) {
    out <- name + " clocked in"
}

func main() {
    ch := make(chan string, 2)
    go shift("Amina", ch)
    go shift("Bo", ch)

    fmt.Println(<-ch)
    fmt.Println(<-ch)
}
```

Run:

```
go run two_shifts.go
```

Possible output (order of the two lines is not guaranteed):

```
Amina clocked in
Bo clocked in
```

or:

```
Bo clocked in
Amina clocked in
```

You did not import a thread library. `go f()` is the whole launch API. Later chapters will slow this down and make the order deterministic. For now, notice how little machinery there is.

Case 4: What Go refuses to do

This program is valid Go. The version that overloads `+` for tickets, subclasses `Worker`, or catches a `TableException` is **not**. The language is small so the team shares one dialect.

```
// no_magic.go
package main

import "fmt"

type Ticket struct {
    ID    int
    Table int
}

func (t Ticket) Label() string {
    return fmt.Sprintf("ticket %d → table %d", t.ID, t.Table)
}

func main() {
    t := Ticket{ID: 7, Table: 12}
    fmt.Println(t.Label())
}
```

Run:

```
go run no_magic.go
```

Output:

```
ticket 7 → table 12
```

Methods exist. Classes, inheritance, and default arguments do not. Composition (a struct with fields, functions that take those structs) is the whole design story.

The trap

Coming from a language that prizes abstraction, it is tempting to wrap every `fmt.Println` in a framework on day one. That is how a 40-line desk tool becomes a 12-package "platform."

This program does too much for its size: an interface, a constructor, and a type that only exists to print a string.

```
// too_clever.go
package main
```

```go
import "fmt"

type Printer interface {
    Print(string)
}

type stdoutPrinter struct{}

func NewPrinter() Printer { return stdoutPrinter{} }

func (stdoutPrinter) Print(s string) { fmt.Println(s) }

func main() {
    NewPrinter().Print("desk is open")
}
```

Run:

```
go run too_clever.go
```

Output:

```
desk is open
```

It works. It is also a waste. The boring version is Case 1. Introduce an interface when you have **two real implementations** or a test double you cannot avoid — not when you have a slogan about "SOLID."

The boring rule

- Prefer a function and a struct over a hierarchy.
- Return `error`. Do not panic for the caller's mistake (see the errors part).
- Use a goroutine when you have concurrent work, not because it looks modern.
- Keep packages few. Split them when names collide or when a boundary is real.
- If a feature is not in the language, that is usually a hint, not a gap to fill with a library.

Try this

1. Change `hello.go` so it prints the number of tables (an `int`) next to the message. Use `fmt.Printf`.
2. In `open_tables.go`, keep going after a bad table instead of calling `os.Exit`. Print the error and open the rest.
3. In `two_shifts.go`, start a third worker. Receive three times. Confirm the program still exits (it will hang if you forget a receive).

Installing and Running Go

Installing and Running Go

The boring way to get Go is the official toolchain from go.dev/dl. One `go` binary on your `PATH`, **Go 1.27**, modules on by default, and the editor you already use. This chapter stops at a module you can run and a binary you can keep.

Mental model

Go is compiled. The `go` command is the compiler, linker, module manager, and test runner. You install **one** toolchain. Each module pins a language version in `go.mod`. With `GOTOOLCHAIN=auto` (the default), `go` may download a matching compiler. With `GOTOOLCHAIN=local`, it uses only what you installed.

`GOPATH` still names a cache and an install directory (`pkg/mod`, `bin`). It is **not** where new code lives. New code lives in a directory that has a `go.mod`.

A **terminal** is the window where you type commands. `PATH` is the list of directories the shell searches for a program named `go`. A **file path** is a location on disk (`main.go`, `/home/you/desk`). Put the directory that contains the `go` binary on `PATH` so `go version` works from any folder.

Worked examples

Case 1: Prove the toolchain is on PATH

Install from go.dev/dl — the official site, not a random script. Follow that page for your OS. Then:

```
go version
```

Output (OS and architecture match your machine):

```
go version go1.27.0 linux/amd64
```

Anything that starts with `go version go1.27` is fine. If the shell says the command is not found, `PATH` does not include the directory that holds `go`.

This program prints the same facts from inside a running binary:

```
// where.go
package main
```

```
import (
    "fmt"
    "runtime"
)

func main() {
    fmt.Println(runtime.Version())
    fmt.Println(runtime.GOOS + "/" + runtime.GOARCH)
}
```

Run:

```
go run where.go
```

Output (again, OS/arch follow the machine):

```
go1.27.0
linux/amd64
```

`go run` compiles a temporary executable and runs it. There is no interpreter mode.

Case 2: Ask the toolchain about itself

```
go env GOROOT GOPATH GOMODCACHE GOTOOLCHAIN GO111MODULE
```

Typical output:

```
/usr/local/go
/home/you/go
/home/you/go/pkg/mod
auto
on
```

Variable	Meaning
`GOROOT`	Installed toolchain
`GOPATH`	Cache root: module zip cache under `pkg/mod`, `go install` binaries under `bin`
`GOMODCACHE`	Downloaded modules
`GOTOOLCHAIN`	`auto` may fetch a compiler to match `go.mod`; `local` uses only the install
`GO111MODULE`	`on` — modules are the default. Do not turn this off.

You do not create `$GOPATH/src/...` trees for new work.

A `go.mod` can also name a toolchain. The `go` line is the **language** version. The optional `toolchain` line is which compiler `auto` should prefer:

```
module desk

go 1.27

toolchain go1.27.0
```

Leave `GOTOOLCHAIN=auto` unless you are debugging the compiler itself.

Case 3: First module

Pick an empty directory. Create the module metadata, then the program.

```
mkdir desk
cd desk
go mod init desk
```

That writes `go.mod`:

```
module desk

go 1.27
```

Save as `main.go`:

```
// main.go
package main

import "fmt"

func main() {
    fmt.Println("desk module is running")
}
```

Run:

```
go run .
```

Output:

```
desk module is running
```

The `.` means "the package in this directory." That is the habit to keep. `go run main.go` works for a single file; `go run .` is what you want once the folder is a module.

Case 4: Build a binary you can keep

Same module, same `go.mod`. Replace `main.go` with this complete program:

```
// main.go
package main

import "fmt"

func main() {
    fmt.Println("shift board ready")
}
```

Run:

```
go build -o desk .
./desk
```

Output:

```
shift board ready
```

`go build -o desk .` writes an executable named `desk` in the current directory. There is no VM to install on the machine that runs it (this book does not use cgo). Rebuild after you change the source; an old binary will not update itself.

Use whatever editor you already like. A language server (`gopls`) is optional polish. This book never assumes a particular IDE. `gofmt` is not optional; the editor is.

The trap

Installing Go from a distro package that lags a year, from a random `curl | bash`, or from a tutorial that sets `GO111MODULE=off` and dumps code in `$GOPATH/src`. The first file still runs. The project does not exist yet.

This program is fine. The **layout** around it is the bug: no `go.mod`.

```
// orphan.go
package main

import "fmt"

func main() {
    fmt.Println("I have no module")
}
```

Run:

```
go run orphan.go
```

Output:

```
I have no module
```

Single-file `go run` still compiles stdlib-only files outside a module. That convenience hides the missing `go.mod`. The moment you add a second package or a dependency, the folder is not a project. Run `go mod init desk` **before** the second file exists.

The same class of mistake: setting `GO111MODULE=off`, or pinning `GOTOOLCHAIN` to an older compiler "to match a blog post." Pin `go 1.27` in `go.mod`. Install 1.27 from go.dev.

The boring rule

- Install from go.dev/dl. Confirm with `go version`.
- New work starts with `go mod init`.
- Use `go run .` and `go build -o <name> .` from the module root.
- Leave `GOPATH` alone. Do not set `GO111MODULE=off`.
- `GOTOOLCHAIN=auto` is fine. The `go` line in `go.mod` is the language pin.
- The editor is yours. Format with `gofmt`.

Try this

1. Run `go env GOVERSION GOTOOLCHAIN`. Then `GOTOOLCHAIN=local go env GOTOOLCHAIN` and confirm it prints `local`.
2. In the `desk` module, change the printed string and run `go build -o desk .` again. Run `./desk` and confirm it is the new text, not a stale binary.
3. From `desk/`, run `go env GOMOD`. It should print the full path to `go.mod`. From your home directory, run it again and notice it is empty (or `/dev/null`).

How Go Code Is Organized

How Go Code Is Organized

A **module** is the unit you version and download. A **package** is the unit you import and compile. The boring default is one module, one `package main` at the root, until a real boundary appears. Capitals are the public API.

Mental model

`go.mod` names the module (`module desk` or `module example.com/desk`). Every import path starts with that module path, then the directory: `example.com/desk/greet`.

A directory is one package. The `package` clause is the name importers use in code (`greet.Hello`), not the directory name — though they should match.

`package main` plus `func main` is a program. Any other package name is a library. Names that start with a capital letter are **exported** (visible to other packages). The rest are private to the package.

`internal/` is enforced by the toolchain: only code rooted at the parent of `internal` may import it. `cmd/` is a convention, not a rule: one directory per binary when you have more than one, or when the root is a library.

Worked examples

Case 1: Exported means a capital letter

Save as `names.go`. Both functions live in `package main`, so both are callable here. The names still teach the rule you will need the moment a second package exists.

```
// names.go
package main

import "fmt"

func OpenShift(name string) string {
    return name + " is on"
}

func closeShift(name string) string {
    return name + " is off"
```

```
}

func main() {
    fmt.Println(OpenShift("Amina"))
    fmt.Println(closeShift("Amina"))
}
```

Run:

```
go run names.go
```

Output:

```
Amina is on
Amina is off
```

Another package could call `OpenShift`. It could not call `closeShift`. There is no `public` keyword. The first letter is the whole policy.

Case 2: A tiny two-package module

Three files, one module. Create a directory, then save each listing at the path in the comment.

`go.mod`:

```
module example.com/desk

go 1.27
```

Save as `greet/greet.go`:

```
// greet.go
package greet

import "fmt"

func Hello(name string) string {
    return fmt.Sprintf("hello, %s", name)
}

func whisper(name string) string {
    return "(" + name + ")"
}
```

Save as `main.go`:

```go
// main.go
package main

import (
    "fmt"

    "example.com/desk/greet"
)

func main() {
    fmt.Println(greet.Hello("desk"))
}
```

Run from the directory that contains `go.mod`:

```
go run .
```

Output:

```
hello, desk
```

The import path is `example.com/desk` plus `/greet`. The identifier is `greet.Hello`. `greet.whisper` would not compile from `main.go` — try it once and read the error.

`whisper` is not dead code. Other files in `package greet` could call it. Privacy is per package, not per file.

Case 3: `cmd/` and `internal/`

Same module path. Layout:

```
.
   go.mod
   cmd/desk/main.go
   internal/ticket/ticket.go
```

`go.mod` is unchanged:

```
module example.com/desk

go 1.27
```

Save as `internal/ticket/ticket.go`:

```go
// ticket.go
package ticket

import "fmt"

type Ticket struct {
    ID    int
    Table int
}

func (t Ticket) Label() string {
    return fmt.Sprintf("ticket %d → table %d", t.ID, t.Table)
}
```

Save as `cmd/desk/main.go`:

```go
// main.go
package main

import (
    "fmt"

    "example.com/desk/internal/ticket"
)

func main() {
    t := ticket.Ticket{ID: 9, Table: 2}
    fmt.Println(t.Label())
}
```

Run:

```
go run ./cmd/desk
```

Output:

```
ticket 9 → table 2
```

`example.com/desk/internal/ticket` is importable from `example.com/desk/...`. A different module cannot import it. That is the whole point of `internal/`.

You do not need `cmd/` for a single binary. Root `package main` (Case 2) is enough. Add `cmd/desk` when the repository also holds a library, or a second command.

The trap

Splitting a 60-line desk tool into `pkg/`, `internal/`, `cmd/`, and four packages named `util`, `helpers`, `common`, and `types`. Import paths get longer. Names get shorter. Nobody can find `OpenShift`.

This complete program is the whole application. It does not need a directory tree.

```
// one_package.go
package main

import "fmt"

type Shift struct {
    Name string
    On   bool
}

func (s Shift) Line() string {
    state := "off"
    if s.On {
        state = "on"
    }
    return s.Name + " is " + state
}

func main() {
    fmt.Println(Shift{Name: "Bo", On: true}.Line())
}
```

Run:

```
go run one_package.go
```

Output:

```
Bo is on
```

The other trap is the mirror image: exporting every name "so tests can see it," or putting two packages in one directory (`package greet` next to `package main`). One directory, one package. Unexported tests use `package greet` in `greet_test.go`; that is a later chapter.

The boring rule

- One module per project until you publish two versioned units.
- Import path = module path + directory.
- Capital letter = API. Lowercase = package private.

- Start with `package main` at the root. Add `greet/` when a name is reused.
- Add `internal/` when you need a wall against other modules. Add `cmd/<name>` when you have two binaries or a library plus a binary.
- Do not invent `pkg/` or `utils/` on day one.

Try this

1. In Case 2, change `main.go` to call `greet.whisper("desk")`. Read the compiler error. Change `whisper` to `Whisper` and fix the call.
2. In Case 3, add `cmd/board/main.go` that also imports `internal/ticket` and prints a different ticket. Run `go run ./cmd/board`.
3. Rename the module path in `go.mod` to `example.com/cafe` and update every import. Run `go run .` (or `go run ./cmd/desk`) until it builds.

Go Tour

Go Tour

A short walk through the pieces you will type every day: variables, `if` and `for`, a slice, a map, a struct, a function, a method, and an `error` return. Each listing is a complete program. The desk is the same; the files still run alone.

Mental model

Go is statically typed. Every name has a type at compile time. The zero value is usable: `0`, `""`, `nil`, `false`. Control flow is explicit: `if err != nil`, a single `for`, no `while`. Composite data is a slice (sequence), a map (lookup), or a struct (named fields). Behaviour is functions, plus methods attached to a type. Failure is a value you return, not an exception you throw.

This chapter does not cover interfaces or generics. You do not need them to write the programs here.

Worked examples

Case 1: Variables and types

Save as `vars.go`. `var` names a type. `:=` infers it from the right-hand side. Both are ordinary.

```
// vars.go
package main

import "fmt"

func main() {
    var tables int = 8
    open := true
    var title string
    price := 3.50

    fmt.Printf("tables=%d open=%t title=%q price=%.2f\n", tables, open, title, price)
}
```

Run:

```
go run vars.go
```

Output:

```
tables=8 open=true title="" price=3.50
```

`title` was never assigned, so it is `""`. `int`, `bool`, `string`, and `float64` are the types you will see first. `Printf` verbs: `%d` integer, `%t` bool, `%q` quoted string, `%.2f` two decimal places.

Case 2: `if` and `for`

Save as `shifts.go`. There is one loop keyword. A three-clause `for` counts. `range` walks a slice.

```go
// shifts.go
package main

import "fmt"

func main() {
    names := []string{"Amina", "Bo", "Cara"}
    for i := 0; i < len(names); i++ {
        if names[i] == "Bo" {
            fmt.Println("skip", names[i])
            continue
        }
        fmt.Println("shift", names[i])
    }
    for _, name := range names {
        fmt.Println("listed", name)
    }
}
```

Run:

```
go run shifts.go
```

Output:

```
shift Amina
skip Bo
shift Cara
listed Amina
listed Bo
listed Cara
```

`continue` skips the rest of that iteration. `_` discards the index from `range`. There is no `while`; a condition-only `for` is the equivalent, shown in later chapters.

Case 3: A slice of orders

Save as `orders.go`. A slice is a view over an array: length, capacity, and a pointer you do not manage by hand. `append` may return a new backing array — always keep the result.

```
// orders.go
package main

import "fmt"

func main() {
    orders := []string{"tea"}
    orders = append(orders, "toast", "soup")
    fmt.Println("len", len(orders), "cap", cap(orders))
    fmt.Println("first", orders[0])
    fmt.Println("rest", orders[1:])
    for i, item := range orders {
        fmt.Printf("%d %s\n", i, item)
    }
}
```

Run:

```
go run orders.go
```

Output:

```
len 3 cap 3
first tea
rest [toast soup]
0 tea
1 toast
2 soup
```

Capacity can be larger than length after later appends. Do not index past `len-1`. `orders[1:]` is a new slice heading, same backing array.

Case 4: A map of tables

Save as `tables.go`. A map keys a value. The zero value of a missing key is usable. The comma-ok form tells presence apart from zero.

```
// tables.go
package main

import "fmt"
```

```
func main() {
    seats := map[int]string{1: "Amina", 2: "Bo"}
    seats[3] = "Cara"
    fmt.Println("table 2", seats[2])
    name, ok := seats[9]
    fmt.Printf("table 9 name=%q ok=%t\n", name, ok)
    delete(seats, 2)
    fmt.Println("len", len(seats))
    for table, who := range seats {
        fmt.Printf("%d:%s\n", table, who)
    }
}
```

Run:

```
go run tables.go
```

Possible output (`range` over a map is unordered):

```
table 2 Bo
table 9 name="" ok=false
len 2
1:Amina
3:Cara
```

The pair `1:Amina` and `3:Cara` may swap. Never depend on map iteration order.

Case 5: Struct, function, method, error

Save as `ticket.go`. A struct groups fields. A function returns `(Ticket, error)`. A method has a receiver. Check `err` before using the value.

```
// ticket.go
package main

import (
    "fmt"
    "os"
)

type Ticket struct {
    ID    int
    Table int
}

func NewTicket(id, table int) (Ticket, error) {
```

```
    if id <= 0 {
        return Ticket{}, fmt.Errorf("id must be positive")
    }
    if table <= 0 {
        return Ticket{}, fmt.Errorf("table must be positive")
    }
    return Ticket{ID: id, Table: table}, nil
}

func (t Ticket) Label() string {
    return fmt.Sprintf("ticket %d → table %d", t.ID, t.Table)
}

func main() {
    t, err := NewTicket(12, 4)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    fmt.Println(t.Label())

    _, err = NewTicket(0, 4)
    if err != nil {
        fmt.Println("rejected:", err)
    }
}
```

Run:

```
go run ticket.go
```

Output:

```
ticket 12 → table 4
rejected: id must be positive
```

`nil` means success. The empty `Ticket{}` on failure is the zero value; callers who skip the error check will see `id 0`, which is why the check is not optional.

The trap

Treating this tour as a pile of independent tricks, then inventing a mini-framework that wraps each one. The next program "improves" Case 5 by hiding the error and panicking. It is shorter. It is worse.

```go
// panic_ticket.go
package main

import "fmt"

type Ticket struct {
    ID    int
    Table int
}

func MustTicket(id, table int) Ticket {
    if id <= 0 || table <= 0 {
        panic("bad ticket")
    }
    return Ticket{ID: id, Table: table}
}

func main() {
    t := MustTicket(12, 4)
    fmt.Printf("ticket %d → table %d\n", t.ID, t.Table)
}
```

Run:

```
go run panic_ticket.go
```

Output:

```
ticket 12 → table 4
```

Change `12` to `0` and the process dies. `Must` helpers belong in tests and `main` wiring, not in the function every caller uses. Return `error`. Check it.

The boring rule

- Declare types you care about. Let `:=` infer the rest in short functions.
- One `for`. Use `range` for slices; use the comma-ok form for maps.
- Keep `append`'s result. Do not index past `len`.
- Maps are unordered. Presence is `value, ok`.
- Structs hold data. Functions and methods do work. Errors come back as values.
- Skip interfaces and generics until a second implementation or a real type parameter earns them.

Try this

1. In `vars.go`, add a `const` for the table count and use it in the `Printf`.
2. In `orders.go`, append four more items in a loop. Print `len` and `cap` after each append. Watch capacity jump.
3. In `ticket.go`, reject table numbers greater than 20 with a new error. Call `NewTicket(1, 21)` from `main` and print the result.

Toolchain

Core Go Commands

Core Go Commands

The `go` command is the whole toolchain: run, build, test, format, vet, document, list, env, clean. Learn these nine well and you can ignore the rest for a long time.

Mental model

You type `go <verb>` in a module directory. The verb looks at `go.mod`, the `.go` files in the package, and (for tests) the `*_test.go` files. Output is deterministic enough to script. None of these verbs need an IDE.

`go run` is for trying a program. `go build` is for keeping a binary. `go test` is how you know the program still works. `gofmt` / `go fmt` are how the team shares one layout. `go vet` catches mistakes the compiler allows. `go doc` and `go list` answer questions. `go env` prints the machine. `go clean` removes build debris.

Worked examples

Case 1: `go run` and `go build`

Save as `open.go` in a module (`go mod init desk` if you do not have one):

```
// open.go
package main

import "fmt"

func main() {
    fmt.Println("desk open")
}
```

Run:

```
go run .
```

Output:

```
desk open
```

Build and run the binary:

```
go build -o desk .
./desk
```

Output:

```
desk open
```

`go run` throws the executable away. `go build -o desk .` keeps it. Ship the build, not the run.

Case 2: `go test`

Two files in the same package. `hours.go` is the program. `hours_test.go` is the test. `go test` compiles them together.

Save as `hours.go`:

```
// hours.go
package main

import "fmt"

func openHours() string {
    return "09:00-17:00"
}

func main() {
    fmt.Println(openHours())
}
```

Save as `hours_test.go`:

```
// hours_test.go
package main

import "testing"

func TestOpenHours(t *testing.T) {
    got := openHours()
    want := "09:00-17:00"
    if got != want {
        t.Fatalf("openHours() = %q, want %q", got, want)
    }
}
```

Run:

```
go test
```

Output:

```
PASS
ok      desk    0.001s
```

The module path in the `ok` line matches your `go.mod`. The time varies. `FAIL` plus a `Fatalf` message means the strings did not match. `go test` does not run `main`.

Case 3: `go fmt` and `gofmt`

`gofmt` is the formatter. `go fmt` is a thin wrapper that runs it on packages. Save this as `messy.go` (it is deliberately ugly, and it still compiles):

```
// messy.go
package main

import "fmt"

func main() {
    fmt.Println("aligned")
}
```

The listing above is already formatted. To see the tool work, type a messy copy yourself: extra spaces around `Println`, tabs mixed with spaces, braces on their own strange lines. Then:

```
gofmt -w messy.go
```

`-w` writes the file in place. Without `-w`, `gofmt` prints the formatted source to stdout. `go fmt ./...` formats every package under the current directory. There is no style debate. The output of `gofmt` is the style.

Case 4: `go vet`

Save as `badprint.go`. The compiler accepts it. `go vet` does not.

```
// badprint.go
package main

import "fmt"

func main() {
    ticket := "12"
    fmt.Printf("ticket %d\n", ticket)
}
```

Run:

```
go vet badprint.go
```

Output:

```
badprint.go:8:21: fmt.Printf format %d has arg ticket of wrong type string
```

Run it anyway:

```
go run badprint.go
```

Output:

```
ticket %!d(string=12)
```

The program "works." The line is garbage. Change `%d` to `%s` (or `ticket` to an `int`) and `go vet` is silent. Case 5 of the next chapter writes the fix in full.

Case 5: `go doc`, `go list`, `go env`, `go clean`

Documentation for a stdlib function:

```
go doc fmt.Println
```

Output starts with:

```
package fmt // import "fmt"

func Println(a ...any) (n int, err error)
    Println formats using the default formats for its operands and writes to
    standard output.
```

List the package and the module:

```
go list .
go list -m
```

Output (module name from `go.mod`):

```
desk
desk
```

Environment:

```
go env GOVERSION GOMOD GOCACHE
```

Typical output:

```
go1.27.0
/home/you/desk/go.mod
/home/you/.cache/go-build
```

Clean the default binary name after `go build` (directory `desk`, no `-o`):

```
go build
go clean
```

`go clean` removes the executable `go build` wrote in this directory. It does not delete your `.go` files. `go clean -cache` wipes the build cache — leave that alone unless a cache bug is the problem.

The trap

Using `go run` as the production start command, or wrapping every verb in a Makefile before you can recite them. This program is three lines. The trap is the process around it.

```
// serve.go
package main

import "fmt"

func main() {
    fmt.Println("would listen on :8080")
}
```

Run:

```
go run serve.go
```

Output:

```
would listen on :8080
```

In a container or a service unit, run **the binary** from `go build` (or `go install`). `go run` rebuilds from source every time, hides compile errors behind a convenience wrapper, and is not how you pin what you shipped.

The other trap: skipping `go vet` because the compiler was green. Case 4 is the exhibit.

The boring rule

- `go run .` while learning. `go build -o <name> .` when you keep a binary.
- `go test` in the package directory. Tests are `*_test.go`.
- `gofmt -w` (or `go fmt ./...`) on every save. No personal layout.
- `go vet ./...` before you call the change done.
- `go doc`, `go list`, `go env` instead of guessing paths.
- `go clean` for leftover binaries. Not `-cache` as a ritual.

Try this

1. In the `desk` module, run `go build -o desk .` then `go run .`. Confirm both print the same line.
2. Break `TestOpenHours` by changing `want`. Run `go test` and read the `Fatalf` line. Fix it.
3. Run `go doc -all fmt` and find `Printf`. Then `go list -f '{{.GoFiles}}' .` in a package that has more than one `.go` file.

Formatting, Vetting, and Documentation

Formatting, Vetting, and Documentation

`gofmt` is not a preference. `go vet` is not optional CI garnish. Doc comments are the API. The boring team runs all three on every change because arguments about spaces, printf verbs, and "what does this function do" are more expensive than the tools.

Mental model

`gofmt` takes Go source and prints Go source with one layout: tabs for indentation, a canonical import block, a standard take on line breaks. You do not configure it.

`go vet` runs checks the compiler skips: wrong `Printf` verbs, unreachable code, common lock mistakes. A vet finding is a bug even when the binary runs.

A **doc comment** is a complete sentence immediately above a package, type, function, or method, starting with the name. `go doc` prints those comments. If `go doc` is empty, you did not write documentation — you wrote a note somewhere else.

Worked examples

Case 1: `gofmt` is the style

Save as `before.go`. This is legal Go with noisy spacing. `gofmt` will rewrite it.

```
// before.go
package main

import "fmt"

func main() {
fmt.Println(  "tea",   2)
}
```

Run:

```
gofmt before.go
```

Output (stdout, file unchanged without `-w`):

```
// before.go
package main

import "fmt"

func main() {
    fmt.Println("tea", 2)
}
```

Write it back:

```
gofmt -w before.go
```

`go fmt ./...` does the same for every package under `.`. Editors should run `gofmt` on save. A pull request that bikesheds alignment is a pull request that forgot the tool.

The first-line `// before.go` filename comment is for this book. `gofmt` leaves comments alone.

Case 2: A real `go vet` finding, then the fix

Save as `board.go`. The compiler is happy. The output is not.

```
// board.go
package main

import "fmt"

func main() {
    table := "4"
    fmt.Printf("open table %d\n", table)
}
```

Run:

```
go vet board.go
```

Output:

```
board.go:8:25: fmt.Printf format %d has arg table of wrong type string
```

Run anyway:

```
go run board.go
```

Output:

```
open table %!d(string=4)
```

`%d` expects an integer. `table` is a `string`. Vet saw it; the runtime printed a placeholder.

Save as `board.go` again, with a matching verb:

```
// board.go
package main

import "fmt"

func main() {
    table := "4"
    fmt.Printf("open table %s\n", table)
}
```

Run:

```
go vet board.go
go run board.go
```

Output:

```
open table 4
```

`go vet` printed nothing. That is success. `go vet ./...` in a module is the form to keep.

Case 3: Doc comments and `go doc`

Save `go.mod`:

```
module example.com/desk

go 1.27
```

Save as `shift.go`:

```
// shift.go

// Package main is a tiny shift board for the front desk.
package main

import "fmt"

// Shift is one person's time on the desk.
type Shift struct {
    Name string
    Hour int
```

```
}

// Label returns a single-line description of the shift.
func (s Shift) Label() string {
    return fmt.Sprintf("%s at %02d:00", s.Name, s.Hour)
}

func main() {
    s := Shift{Name: "Amina", Hour: 9}
    fmt.Println(s.Label())
}
```

Run the program:

```
go run .
```

Output:

```
Amina at 09:00
```

Ask `go doc` about the type and the method (from the module directory):

```
go doc Shift
go doc Shift.Label
```

Output of `go doc Shift`:

```
type Shift struct {
    Name string
    Hour int
}
    Shift is one person's time on the desk.

func (s Shift) Label() string
```

Output of `go doc Shift.Label`:

```
func (s Shift) Label() string
    Label returns a single-line description of the shift.
```

The comment starts with the name (`Shift is…`, `Label returns…`). `// Package main …` documents the package. `// TODO` above a function is not a doc comment if it does not describe the function.

`go doc fmt.Printf` works the same for the standard library. You already have that documentation locally; you do not need a browser.

The trap

Turning `gofmt` off because "the team prefers spaces," or ignoring vet because "it still prints." This program documents nothing and formats nothing until you run the tools. The silent failure is the printf.

```
// quiet.go
package main

import "fmt"

// this helper prints a ticket id
func show(id int) {
    fmt.Printf("ticket %s\n", id)
}

func main() {
    show(7)
}
```

Run:

```
go vet quiet.go
go run quiet.go
```

Output:

```
quiet.go:8:21: fmt.Printf format %s has arg id of wrong type int
ticket %!s(int=7)
```

Two mistakes: `%s` on an `int`, and a comment that does not start with `show` so `go doc show` has nothing useful to say. Fix the verb (`%d`), rewrite the comment to `// Show prints a ticket id.` (and export it if it is API), run `gofmt -w quiet.go`.

A private formatter, a disabled vet check, or documentation in a wiki nobody updates are the same decision: the source is no longer the source of truth.

The boring rule

- `gofmt -w` on every file. Tabs. No config file.
- `go vet ./...` must be clean before you merge.
- Doc comment: full sentence, starts with the name, sits on the line above the declaration.
- `// Package name …` on the package clause.
- `go doc` locally. Do not paste stdlib docs into your tree.
- Do not disable a vet check to silence a real printf bug.

Try this

1. Add a space before every `(` in `shift.go`. Run `gofmt -w shift.go` and confirm the file snaps back.
2. In the fixed `board.go`, change `%s` to `%q` and run the program. Then run `go vet`. Decide whether vet is right to stay silent (`%q` is valid for strings).
3. Export `show` as `Show` in a fixed `quiet.go`, give it a proper doc comment, and run `go doc Show`.

Dependency Management

Dependency Management

A module lists what it needs in `go.mod` and records checksums in `go.sum`. The boring default is the module cache plus a proxy, not a `vendor/` tree. This chapter's runnable programs use only the standard library so they work offline. `go get` is shown as a command you will type when you do add a module.

Mental model

`go.mod` has a module path, a language version (`go 1.27`), and `require` lines for other modules. `go.sum` is a lock of cryptographic hashes. Commit both.

The **module cache** lives under `GOMODCACHE` (usually `$GOPATH/pkg/mod`). Builds read from there. `GOPROXY` (default `https://proxy.golang.org,direct`) is how the toolchain fetches code you do not have yet.

`go get` adds or upgrades a require. `go mod tidy` adds what the source imports and drops what it does not. Versions are semantic (`v1.2.3`). `retract` in an upstream `go.mod` marks a version you should not use. `vendor/` is an optional snapshot for networks that cannot reach a proxy — not the everyday layout.

Worked examples

Case 1: A module that only uses the standard library

Empty directory. Save `go.mod`:

```
module example.com/desk

go 1.27
```

Save as `main.go`:

```go
// main.go
package main

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

```
func main() {
    fmt.Fprintf(os.Stdout, "desk module %s\n", "example.com/desk")
}
```

Run:

```
go run .
go list -m
```

Output:

```
desk module example.com/desk
example.com/desk
```

There is no `require` block. There may be no `go.sum`, or an empty one. That is correct: the standard library is not a module you download. `go.sum` grows when you add a **module** dependency.

Case 2: Stdlib packages still belong in the import block, not in `go.mod`

Save as `menu.go` next to the same `go.mod`. JSON is in the standard library. It does not get a `require` line.

```
// menu.go
package main

import (
    "encoding/json"
    "fmt"
    "os"
)

type Item struct {
    Name  string `json:"name"`
    Price int    `json:"price"`
}

func main() {
    item := Item{Name: "tea", Price: 3}
    err := json.NewEncoder(os.Stdout).Encode(item)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}
```

This file is a second `package main` in the same directory as Case 1. Keep **one** `main.go` when you run `..` For this case, run the file:

```
go run menu.go
```

Output:

```
{"name":"tea","price":3}
```

`encoding/json` is part of the toolchain. If you find yourself `go get`-ing a JSON library on day one, stop. Read the stdlib first.

Case 3: `go get`, `go mod tidy`, versions (commands)

When you do need an external module, the commands look like this. Do not run them for this chapter's programs; they need the network and a real module.

```
go get example.com/mod@v1.2.3
go get example.com/mod@latest
go get example.com/mod@none
go mod tidy
```

`@v1.2.3` pins a version. `@latest` takes the highest release the proxy knows (not a pseudo-version from the default branch unless there are no tags). `@none` drops the dependency from `go.mod`.

`go mod tidy` is the closer: it reads your imports, adds missing `require` lines, removes unused ones, and updates `go.sum`. Run it after you add or delete an import.

A `go.mod` with a dependency looks like this (illustrative — this module is not fetched here):

```
module example.com/desk

go 1.27

require example.com/mod v1.2.3
```

`go.sum` then has hash lines for `example.com/mod` and its `go.mod`. Never edit `go.sum` by hand. If it is dirty, run `go mod tidy` or `go get` again.

Case 4: `retract`, and why not to vendor by default

Upstream authors mark bad versions in **their** `go.mod`:

```
retract v1.0.2 // published with a broken zip
```

or:

```
retract [v1.0.0, v1.0.2] // all of these builds are wrong
```

`go get example.com/mod@v1.0.2` then warns. You pick a version that is not retracted. You do not copy retract lines into your own module unless you are the author who shipped the bad tag.

`go mod vendor` copies required modules into `vendor/`. Builds can use `-mod=vendor`. That is useful when CI has **no network** and **no module cache**. It is not useful as the default: the directory bloats the repository, every upgrade is a huge diff, and `go.sum` already pins content. Prefer a proxy and a warm cache. Vendor when a real constraint forces you to.

Case 5: The `replace` directive for local development

When developing a library alongside an application, or patching a bug in a dependency before upstream merges your fix, add a `replace` directive to `go.mod`:

```
module example.com/desk

go 1.27

require example.com/store v1.2.0

replace example.com/store => ../store
```

The toolchain ignores the version in the module cache and compiles directly against the files in `../store`. You can edit both without publishing intermediate git tags.

When done testing, drop the `replace` directive before pushing your branch:

```
go mod edit -dropreplace example.com/store
```

Case 6: Major version upgrades (`/v2`, `/v3`)

In Go's module system, Semantic Versioning is part of the import path. A major version bump (from `v1` to `v2`) represents a breaking API change. Go enforces the **Import Compatibility Rule**: if an old API and a new API have the same import path, the new API must be backwards-compatible with the old one.

Therefore, breaking releases require a major version suffix in the module declaration:

```
module example.com/desk/v2

go 1.27
```

Callers import the new major version explicitly:

```
import "example.com/desk/v2/orders"
```

Because `example.com/desk` and `example.com/desk/v2` are distinct module paths, a project can import both simultaneously during migration without version conflicts or dependency collisions.

The trap

Hand-editing `go.mod` versions until the build "seems" to work, deleting `go.sum` because it looks noisy, or vendoring on day one "for reproducibility." Reproducibility is `go.sum` plus the proxy. This program needs no extra module. The trap is adding one anyway.

```
// leftover.go
package main

import "fmt"

func main() {
    fmt.Println("still stdlib")
}
```

Run:

```
go run leftover.go
```

Output:

```
still stdlib
```

If `go.mod` still `require`s a module this file does not import, `go mod tidy` will drop it. That is the point. A leftover require is a leftover attack surface and a leftover upgrade. Do not keep dependencies "in case we need them."

The other trap: `replace` lines pointed at a coworker's laptop path, committed forever. `replace` is a local override. Workspaces (next chapter) are the cleaner way to develop two modules at once.

The boring rule

- `go.mod` and `go.sum` are source. Commit them.
- Stdlib is not a `require`. Import it; do not `go get` it.
- `go get pkg@version` to add or upgrade. `go mod tidy` to reconcile.
- Major breaking changes require updating the module path suffix (`/v2`).
- Use `replace` only for local temporary debugging or offline forks; never commit a replace targeting local filesystem paths.

- Pin versions you care about. Read retract warnings; do not ignore them.
- Do not vendor by default. Do not delete `go.sum`. Do not leave unused requires.

Try this

1. In Case 1, run `go env GOMODCACHE GOPROXY`. Read the two paths.
2. Run `go list -m all` in that module. You should see only `example.com/desk`.
3. Add an unused import of `strings` to `main.go`, run `go run .` (it should fail), then remove the import and run `go mod tidy`. Confirm `go.mod` still has no `require` block.
4. Run `go list -m -u all` in any project with dependencies to see available minor and patch updates.

Workspaces and Multi-Module Development

Workspaces and Multi-Module Development

Most desk tools are **one module**. A workspace (`go.work`) is how you develop two or more modules on disk as one build without committing `replace` lines. The boring default is still a single `go.mod`. Split modules when they have different versions or different consumers — not because the folder feels busy.

Mental model

A **workspace** is a `go.work` file that lists local module directories:

```
go 1.27

use (
    ./desk
    ./prices
)
```

Inside that workspace, an import of `example.com/prices` resolves to `./prices`, not to a downloaded zip. You do **not** need a `replace` in `desk/go.mod`.

`replace` in `go.mod` is the older hammer: it rewrites an import path for everyone who builds that module. Fine for a fork you control. Bad as "please use my cousin's checkout." A workspace is local to the developer. Commit `go.work` only when the repository is the workspace (a small number of private modules that always move together). Do not commit `go.work` into a public library that strangers will `go get`.

Worked examples

Case 1: One module is enough

This is the default. One `go.mod`, one `main.go`, prices inlined. No workspace.

`go.mod`:

```
module example.com/desk

go 1.27
```

Save as `main.go`:

```
// main.go
package main

import "fmt"

func price(item string) int {
    switch item {
    case "tea":
        return 3
    case "toast":
        return 4
    default:
        return 0
    }
}

func main() {
    fmt.Println("tea", price("tea"))
}
```

Run:

```
go run .
```

Output:

```
tea 3
```

Stop here if `price` is only used by this program. A second module would be ceremony.

Case 2: Two modules and a workspace

When `prices` is a separate module (another repo, another version, another binary that also imports it), check both out as siblings and put a workspace **above** them. Create the two `go.mod` files **before** `go work init` — an empty directory is not a module, and init will not add it.

```
mkdir -p work/desk work/prices
cd work
```

`prices/go.mod`:

```
module example.com/prices

go 1.27
```

Save as `prices/prices.go`:

```go
// prices.go
package prices

func For(item string) int {
    switch item {
    case "tea":
        return 3
    case "toast":
        return 4
    default:
        return 0
    }
}
```

`desk/go.mod` — **no** `replace`. The require is a version name only; the workspace maps it to `./prices`:

```
module example.com/desk

go 1.27

require example.com/prices v0.0.0
```

Save as `desk/main.go`:

```go
// main.go
package main

import (
    "fmt"

    "example.com/prices"
)

func main() {
    fmt.Println("tea", prices.For("tea"))
    fmt.Println("toast", prices.For("toast"))
}
```

Now, with both `go.mod` files on disk:

```
go work init ./desk ./prices
```

That writes `go.work` (the `go` line may show a patch version such as `1.27.1`):

```
go 1.27
```

```
use (
    ./desk
    ./prices
)
```

Run from `work/` (the directory that contains `go.work`):

```
go run ./desk
```

Output:

```
tea 3
toast 4
```

The `require example.com/prices v0.0.0` is a version name only. You did not publish `v0.0.0`. In the workspace it still resolves to `./prices`. There is often no `go.sum` line for that unpublished path.

`go work use ./other` adds another module later. `go work sync` copies the workspace's selected versions into each module's `go.mod` requires.

Case 3: `replace` instead of a workspace

Same two modules, no `go.work`. `desk/go.mod` must rewrite the path:

```
module example.com/desk

go 1.27

require example.com/prices v0.0.0

replace example.com/prices => ../prices
```

`desk/main.go` and `prices/` are the same as Case 2. From `desk/`:

```
go run .
```

Output:

```
tea 3
toast 4
```

It works. The `replace` is now part of `desk`'s module file. Anyone who clones **only** `desk` gets a broken `../prices`. That is why workspaces exist: the override lives in `go.work`, which you can keep uncommitted (add it to `.gitignore`) while `desk/go.mod` stays publishable.

Use `replace` when you permanently fork a module to a path you do publish (`replace example.com/mod => example.com/you/mod v1.2.3`). Use a workspace when you are editing two checkouts on your machine.

The trap

Creating `go.work` because a blog said "monorepo," then splitting a 200-line desk app into five modules that cannot be built unless the workspace file is present. This program is the whole product. It does not need a sibling module.

```
// solo.go
package main

import "fmt"

func main() {
    fmt.Println("one module")
}
```

Run:

```
go run solo.go
```

Output:

```
one module
```

The other trap is committing `go.work` with absolute paths (`use /Users/you/src/prices`). `use` directories should be relative to `go.work`. Absolute paths fail on every other machine.

A third trap: leaving a `replace => ../prices` in a module you tag as `v1.0.0`. Downstream `go get` cannot follow your laptop.

The boring rule

- One module per program until a library has its own consumers or its own versions.
- `go work init ./a ./b` for local multi-module work. Relative `use` paths.
- Prefer a workspace over a committed `replace` to a relative folder.
- Commit `go.work` only when the repo *is* the workspace. Do not commit it into a library strangers import.
- `replace` is for forks you publish, or a temporary patch you will delete.
- `go run ./desk` from the workspace root. Do not copy source between modules by hand.

Try this

1. In Case 1, add a `soup` price and print it. Do not create a second module.
2. In Case 2, add `prices.For("soup")` returning `5`. Run `go run ./desk` from `work/` and from `desk/` (with `GOWORK=off` from `desk/` it should fail to find `example.com/prices` unless you added a real require+replace).

3. Move `go.work` aside and try Case 3's `replace`. Then delete the `replace` and restore `go.work`. Notice which file you would be willing to commit.

Tool Dependencies

Tool Dependencies

Libraries belong in `require`. **Generators and linters belong in `tool`.** The boring default since Go 1.24 is to pin those executables in `go.mod` with `go get -tool`, then run them with `go tool`. Same version on your laptop and in CI. No blank-import `tools.go`. No "works on my machine" `go install` of whatever was latest last Tuesday.

Mental model

A `require` line is a module your packages import at build time. A `tool` line is a package path of a **main** program your project needs to develop, generate, or check code. The module still records the tool's module version in `require` (usually with `// indirect`). `tool` is the explicit list of "run this with `go tool`."

```
go get -tool path@version   # add or pin a tool
go get -tool path@none      # remove it
go get tool                 # upgrade every tool in this module
go tool name [args...]      # build and run the pinned tool
```

`go tool stringer` works when the last path segment is unique among tools in the module. Full package paths work too. Built-in tools (`go tool compile`, `go tool nm`) still win their names.

If tool modules would bloat the application graph, put them in a separate `go.tool.mod` and pass `-modfile=go.tool.mod`. Most desks start with tools in the main `go.mod`.

Worked examples

Case 1: Pin `stringer` in `go.mod`

Empty directory. Save `go.mod`:

```
module example.com/desk

go 1.27
```

Add the tool (needs the network once):

```
go get -tool golang.org/x/tools/cmd/stringer@v0.34.0
```

`go.mod` now looks like this (exact indirect versions follow the pin you asked for):

```
module example.com/desk

go 1.27

tool golang.org/x/tools/cmd/stringer

require (
    golang.org/x/mod v0.25.0 // indirect
    golang.org/x/sync v0.15.0 // indirect
    golang.org/x/tools v0.34.0 // indirect
)
```

`go.sum` grows. Commit both files. The tool is not imported by your packages. It is still a dependency of the module graph, recorded on purpose.

List tools:

```
go list tool
```

Output:

```
golang.org/x/tools/cmd/stringer
```

Case 2: Generate `String` for a desk status

Keep the `go.mod` from Case 1. Save as `status.go` (package `desk`, not `main` — `stringer` loads types from a normal package):

```
// status.go
package desk

type Status int

const (
    StatusOpen Status = iota
    StatusClosed
)
```

Run the pinned tool:

```
go tool stringer -type=Status
```

That writes `status_string.go`. The important parts look like this (generated; do not hand-edit):

```
// status_string.go
// Code generated by "stringer -type=Status"; DO NOT EDIT.

package desk

import "strconv"

func _() {
    var x [1]struct{}
    _ = x[StatusOpen-0]
    _ = x[StatusClosed-1]
}

const _Status_name = "StatusOpenStatusClosed"

var _Status_index = [...]uint8{0, 10, 22}

func (i Status) String() string {
    if i < 0 || i >= Status(len(_Status_index)-1) {
        return "Status(" + strconv.FormatInt(int64(i), 10) + ")"
    }
    return _Status_name[_Status_index[i]:_Status_index[i+1]]
}
```

Save as `cmd/print/main.go`:

```
// cmd/print/main.go
package main

import (
    "fmt"

    "example.com/desk"
)

func main() {
    fmt.Println(desk.StatusOpen)
    fmt.Println(desk.StatusClosed)
}
```

Run:

```
go run ./cmd/print
```

Output:

```
StatusOpen
StatusClosed
```

Without `String`, `fmt` would print `0` and `1`. The tool keeps names and values aligned when you add a constant later and re-run `go tool stringer`.

Case 3: Wire `go generate`

Put the generate directive in the file that owns the type. Save as `status.go`:

```
// status.go
package desk

//go:generate go tool stringer -type=Status

type Status int

const (
    StatusOpen Status = iota
    StatusClosed
    StatusHeld
)
```

From the module root:

```
go generate ./...
go run ./cmd/print
```

Update `cmd/print/main.go` to print the new constant too:

```
// cmd/print/main.go
package main

import (
    "fmt"

    "example.com/desk"
)

func main() {
    fmt.Println(desk.StatusOpen)
    fmt.Println(desk.StatusClosed)
    fmt.Println(desk.StatusHeld)
}
```

Output:

```
StatusOpen
StatusClosed
StatusHeld
```

The directive says `go tool stringer`, not `go run golang.org/x/tools/cmd/stringer@v0.34.0`, and not a bare `stringer` from `$PATH`. CI runs the same command and gets the same binary.

Case 4: Upgrade and remove

Upgrade every tool listed in this module:

```
go get tool
```

Pin one tool to a newer version:

```
go get -tool golang.org/x/tools/cmd/stringer@v0.34.0
```

Remove it entirely:

```
go get -tool golang.org/x/tools/cmd/stringer@none
go mod tidy
```

After `@none`, `go list tool` prints nothing for that path, and `go tool stringer` fails until you add it back. That failure is good: the module no longer pretends it owns a generator.

The trap

The old pattern was a file that existed only to pull tools into the module graph:

```
// tools.go
//go:build tools

package tools

import (
    _ "golang.org/x/tools/cmd/stringer"
)
```

Then everyone ran a **different** global install:

```
go install golang.org/x/tools/cmd/stringer@latest
stringer -type=Status
```

Two problems. The blank import is a lie — you never call that package. And `@latest` on one laptop is not the version CI used last month. Generated files drift. Reviews turn into "re-run stringer" noise.

The fix is the Case 1 `tool` block plus `go tool stringer` (or `go generate`). Delete `tools.go`. Put the same `go tool` / `go generate` lines in the Makefile and the workflow file.

A related trap: pinning a huge unrelated Go program as a `tool` "because we can." Prefer project generators and analyzers. Install one-off CLIs another way if they only waste module download time.

The boring rule

- Pin tools with `go get -tool path@version`. Commit `go.mod` and `go.sum`.
- Run them with `go tool` or `//go:generate go tool ...`, never with an unpinned global binary.
- Use `go get tool` to bump the whole tool set on purpose.
- Use `@none` to remove a tool; then `go mod tidy`.
- Reach for a separate `go.tool.mod` only when tool requires pollute the app graph.
- Do not keep a `tools.go` blank-import file in a Go 1.27 module.

Try this

1. In Case 1's module, run `go list -m -f '{{.Path}} {{.Version}}' all | head` and find `golang.org/x/tools`.
2. Add `StatusHeld` to the const block, re-run `go tool stringer -type=Status`, and confirm `status_string.go` grew a new name.
3. Run `go get -tool golang.org/x/tools/cmd/stringer@none`, then `go tool stringer -type=Status`. Read the error. Add the tool back.
4. Replace a Makefile line that says `go run golang.org/x/tools/cmd/stringer@...` with `go tool stringer` and commit the `go.mod` that makes that safe.

Types and Values

Predeclared Types

Predeclared Types

Go gives you a short list of types with no import. The boring default is `bool`, `int`, and `string` for local work, `int64` (or another sized type) when the width is part of the contract, and an **explicit conversion** whenever two names differ.

Mental model

A **predeclared type** is a named type that exists in every package. `int` and `int64` are different types even when both hold eight bytes on a 64-bit machine. `byte` is an alias for `uint8`. `rune` is an alias for `int32` and means "Unicode code point," not "UTF-8 byte."

A **literal** such as `4`, `true`, or `"toast"` is *untyped* until you use it in a typed place. A variable always has a type. Conversions are written: `int64(n)`, never implied.

Worked examples

Case 1: The three types you actually type all day

Save as `desk_types.go`. A desk is open or not (`bool`), it has a name (`string`), and it has a table count (`int`).

```
// desk_types.go
package main

import "fmt"

func main() {
    open := true
    name := "front"
    tables := 12
    fmt.Printf("open=%v (%T)\n", open, open)
    fmt.Printf("name=%q (%T)\n", name, name)
    fmt.Printf("tables=%d (%T)\n", tables, tables)
}
```

Run:

```
go run desk_types.go
```

Output:

```
open=true (bool)
name="front" (string)
tables=12 (int)
```

`int` is the default integer. Its size is the size of a register: 8 bytes on the 64-bit machine this book was checked on, 4 bytes on 32-bit. Use it for counts, indexes, and anything that never leaves this process.

Case 2: `int` vs `int64` — convert on the boundary

Save as `ticket_id.go`. Ticket IDs are stored as `int64` because they are a contract with another system. Table numbers stay `int`. You cannot add them until one side converts.

```
// ticket_id.go
package main

import "fmt"

func main() {
    var table int = 4
    var ticket int64 = 1_000_001
    fmt.Println(int64(table) + ticket)
    fmt.Printf("%T + %T → %T\n", int64(table), ticket, int64(table)+ticket)
}
```

Run:

```
go run ticket_id.go
```

Output:

```
1000005
int64 + int64 → int64
```

If you write `table + ticket`, the compile fails: `invalid operation: table + ticket (mismatched types int and int64)`. That failure is the feature. Money in this book is **integer cents** (`int` or `int64`), not `float64`.

Case 3: `byte` and `rune`

Save as `rune_byte.go`. A `byte` is a small unsigned integer. A `rune` is a code point. The euro sign is one rune and three UTF-8 bytes.

```
// rune_byte.go
package main

import "fmt"

func main() {
    var mark byte = 'A'
    var euro rune = '€'
    fmt.Printf("mark %q %d %T\n", mark, mark, mark)
    fmt.Printf("euro %q %d %T %U\n", euro, euro, euro, euro)
}
```

Run:

```
go run rune_byte.go
```

Output:

```
mark 'A' 65 uint8
euro '€' 8364 int32 U+20AC
```

`%T` prints `uint8` and `int32` because those are the real types. The aliases exist so you can say what you mean.

Case 4: Untyped constants, typed variables

Save as `untyped.go`. An untyped constant can land in more than one type. A typed constant cannot.

```
// untyped.go
package main

import "fmt"

func main() {
    const seats = 4
    var asInt int = seats
    var asInt64 int64 = seats
    fmt.Println(asInt, asInt64)

    const typed int = 4
    var also int64 = int64(typed)
    fmt.Println(also)

    fmt.Printf("%T %T %T\n", true, 12, "toast")
}
```

Run:

```
go run untyped.go
```

Output:

```
4 4
4
bool int string
```

`seats` has no type of its own, so both assignments work. `typed` is an `int`; it needs `int64(typed)` to become `int64`. Passing `12` to `fmt.Printf` gives it the **default type** `int` — that is why `%T` prints `int`, not "untyped integer."

Case 5: Sizes, once, so the names stop being magic

Save as `sizes.go`. `unsafe.Sizeof` reports how many bytes a variable occupies. For a `string` that is the header (pointer plus length), not the characters. This is a measuring tape, not a desk tool — do not sprinkle `unsafe` through business code.

```
// sizes.go
package main

import (
    "fmt"
    "unsafe"
)

func main() {
    fmt.Println("bool", unsafe.Sizeof(false))
    fmt.Println("int", unsafe.Sizeof(int(0)))
    fmt.Println("int32", unsafe.Sizeof(int32(0)))
    fmt.Println("int64", unsafe.Sizeof(int64(0)))
    fmt.Println("rune", unsafe.Sizeof(rune(0)))
    fmt.Println("byte", unsafe.Sizeof(byte(0)))
    fmt.Println("string header", unsafe.Sizeof(""))
    fmt.Println("float64", unsafe.Sizeof(float64(0)))
}
```

Run:

```
go run sizes.go
```

Output on 64-bit:

```
bool 1
int 8
int32 4
int64 8
rune 4
byte 1
string header 16
float64 8
```

`rune` matches `int32`. `byte` matches `uint8`. `int` matches the machine. `float64` exists; this book still keeps prices in cents.

The trap

Conversions compile whenever the types are numeric (or string/`[]byte`). They do not check that the value still means the same thing. Save as `truncate.go`:

```
// truncate.go
package main

import "fmt"

func main() {
    euro := '€'
    b := byte(euro)
    fmt.Printf("rune %U %d\n", euro, euro)
    fmt.Printf("as byte %d\n", b)

    minutes := 2.9
    fmt.Println("int(2.9) =", int(minutes))
}
```

Run:

```
go run truncate.go
```

Output:

```
rune U+20AC 8364
as byte 172
int(2.9) = 2
```

`8364` does not fit in a `byte`, so you get `8364 % 256`. `int` of a float **truncates toward zero**; it does not round. Convert when you know the value fits, or when truncation is the specified behaviour. Otherwise keep the wider type.

The boring rule

- Use `int` for counts and indexes in this process.
- Use `int64` (or `int32`) when a file, a network, or another language must agree on width.
- Store money as integer cents. Do not use `float64` for prices.
- Write conversions. If the compiler complains about mismatched types, do not silence it with a random cast — pick the type you meant.
- Use `rune` for characters, `byte` for raw bytes. They are not interchangeable just because both are "small integers."
- Leave `unsafe` for diagnostics and the rare low-level package.

Try this

1. In `ticket_id.go`, delete `int64(table)` and run. Read the compile error. Put the conversion back on the other side (`int(ticket)`) and decide which width you want.
2. In `untyped.go`, try `var also int64 = typed` without the conversion. Confirm only the typed constant fails.
3. In `truncate.go`, convert `'A'` to `byte` and print it. Then convert a rune that does not fit (`' '` or `'€'`) and compare.
4. Add `fmt.Printf("%T\n", 1.5)` to `untyped.go`. The default type of an untyped float is `float64`.

Zero Values and Initialization

Zero Values and Initialization

Every variable in Go has a value from the moment it exists. The boring default is: declare with `var` when the zero is what you want, assign with `:=` when you have a real value, call `make` only for slices, maps, and channels you are about to fill.

Mental model

The **zero value** is the language's "empty but usable" bit pattern for a type: `0`, `""`, `false`, `nil`. You never read uninitialised memory. That is why a missed assignment often looks like a quiet `0` instead of a crash — and why zeros are part of the design, not an accident.

How	What you get
`var x T`	`x` is type `T`, value is zero
`x := v`	type of `x` is the type of `v`
`new(T)`	`*T` pointing at a zero `T`
`make(slice/map/chan)`	a live header, not nil (for the length you asked)

`new` is rarely needed: `&Ticket{}` is the same idea and names the fields. `make` is not optional for maps you will write.

Worked examples

Case 1: Zeros you can print

Save as `zeros.go`. One `var` per kind, no assignment.

```
// zeros.go
package main

import "fmt"

type Ticket struct {
    ID    int
    Table int
}

func main() {
```

```
    var open bool
    var tables int
    var name string
    var ticket *Ticket
    var items []string
    var prices map[string]int
    var t Ticket

    fmt.Printf("bool %v\n", open)
    fmt.Printf("int %d\n", tables)
    fmt.Printf("string %q\n", name)
    fmt.Printf("pointer nil=%v\n", ticket == nil)
    fmt.Printf("slice nil=%v len=%d\n", items == nil, len(items))
    fmt.Printf("map nil=%v len=%d\n", prices == nil, len(prices))
    fmt.Printf("struct %+v\n", t)
}
```

Run:

```
go run zeros.go
```

Output:

```
bool false
int 0
string ""
pointer nil=true
slice nil=true len=0
map nil=true len=0
struct {ID:0 Table:0}
```

A nil slice has length 0. A nil map has length 0. They are not the same as empty-but-allocated, which is what `make` gives you.

Case 2: := when you already know the value

Save as `short_decl.go`. Short declaration infers the type from the right-hand side and must introduce at least one new name.

```
// short_decl.go
package main

import "fmt"

func main() {
    name := "Amina"
    tables := 3
```

```
    open := true
    fmt.Printf("%s %d open=%v\n", name, tables, open)
}
```

Run:

```
go run short_decl.go
```

Output:

```
Amina 3 open=true
```

Use `var tables int = 3` when you want the type on the left for a reader. Use `:=` when the type is obvious from the value.

Case 3: `make` for a map and a slice you will fill

Save as `make_desk.go`. `append` on a nil slice is fine. A write to a nil map panics (next chapter on maps shows that crash). `make` the map before the first write.

```
// make_desk.go
package main

import "fmt"

func main() {
    var items []string
    items = append(items, "toast")
    items = append(items, "tea")

    prices := make(map[string]int)
    prices["toast"] = 350
    prices["tea"] = 250

    fmt.Println(items)
    fmt.Println(prices["toast"], prices["tea"])
}
```

Run:

```
go run make_desk.go
```

Output:

```
[toast tea]
350 250
```

`make([]string, 0, 8)` is the same story with a hint: length 0, capacity 8, so the first few `append`s do not allocate again.

Case 4: `new` vs a pointer to a composite literal

Save as `new_ticket.go`. `new(Ticket)` returns a pointer to a zero ticket. `&Ticket{ID: 7}` does the same job and lets you set fields.

```
// new_ticket.go
package main

import "fmt"

type Ticket struct {
    ID    int
    Table int
}

func main() {
    a := new(Ticket)
    b := &Ticket{}
    c := &Ticket{ID: 7, Table: 12}
    fmt.Printf("new  %#v nil=%v\n", a, a == nil)
    fmt.Printf("&{}  %#v\n", b)
    fmt.Printf("lit  %#v\n", c)
}
```

Run:

```
go run new_ticket.go
```

Output:

```
new  &main.Ticket{ID:0, Table:0} nil=false
&{}  &main.Ticket{ID:0, Table:0}
lit  &main.Ticket{ID:7, Table:12}
```

`a` is not nil. It points at zeros. Prefer `c` when you have data; prefer `var t Ticket` when a value (not a pointer) is enough.

The trap

Zeros look like data. A function that returns `int` cannot tell "table 0" from "I forgot to set the table" unless you add an `error` or a separate ok flag. Save as `zero_table.go`:

```
// zero_table.go
package main

import "fmt"

func tableFor(name string) int {
    if name == "window" {
        return 4
    }
    return 0
}

func main() {
    fmt.Println("window", tableFor("window"))
    fmt.Println("missing", tableFor("patio"))
}
```

Run:

```
go run zero_table.go
```

Output:

```
window 4
missing 0
```

The second line is not a table. It is the zero. The fix is the comma-ok style you will use on maps, or an `error`: `func tableFor(name string) (int, error)`. Do not invent a sentinel like `-1` unless the domain already uses it.

The boring rule

- `var x T` when the zero is the right start (counters, flags, buffers you `append` to).
- `x := v` when `v` is the real value.
- `make` maps before write. `make` slices when you know length or capacity.
- `new(T)` and `&T{}` both yield a non-nil `*T`. Prefer the literal.
- Treat `0`, `""`, and `nil` as values you will see in production, not as "unset" unless you also return `error`.
- Nil slice: `append` is safe. Nil map: write is not.

Try this

1. In `zeros.go`, `append` one string onto `items` and print `items == nil` again. After `append` it is no longer nil.

2. In `make_desk.go`, replace `make(map[string]int)` with `var prices map[string]int` and run. Read the panic.
3. In `new_ticket.go`, add `var t Ticket` and `p := &t`. Print `p`. Same shape as `new`, different way to get there.
4. Change `tableFor` to return `(int, bool)` and make `main` print `"none"` when the bool is false.

Constants and Literals

Constants and Literals

A constant is a value the compiler knows. The boring default is `const` for names that must not change (prices, bit sizes, shift labels), untyped until a type is useful, and `iota` only for a dense list of related integers.

Mental model

`const n = 4` is **untyped**. It can become `int` or `int64` at the use site. `const n int = 4` is **typed** and stays `int`.

`iota` is a counter that starts at 0 in each `const` block and goes up by one per line. A line with no expression repeats the previous expression, which is how `iota` "fills down."

Numeric literals may use `_` as a digit separator. Rune literals are code points in single quotes: `'A'`, `'€'`, `'\n'`.

Worked examples

Case 1: Named prices that cannot drift

Save as `menu_const.go`. Cents are integers. The names are the documentation.

```
// menu_const.go
package main

import "fmt"

const (
    toastCents = 350
    teaCents   = 250
    soupCents  = 600
)

func main() {
    total := toastCents + teaCents
    fmt.Println(total)
    fmt.Printf("%T\n", toastCents)
}
```

Run:

```
go run menu_const.go
```

Output:

```
600
int
```

`toastCents` is untyped until `fmt.Printf("%T")` gives it the default type `int`. You still cannot assign to it: there is no `toastCents = 1`.

Case 2: Untyped vs typed constants

Save as `typed_const.go`. The untyped seat count fits both widths. The typed one needs a conversion.

```
// typed_const.go
package main

import "fmt"

func main() {
    const seats = 4
    var i int = seats
    var w int64 = seats

    const typedSeats int = 4
    var also int64 = int64(typedSeats)

    fmt.Println(i, w, also)
}
```

Run:

```
go run typed_const.go
```

Output:

```
4 4 4
```

`var also int64 = typedSeats` does not compile. Typed constants follow the same conversion rules as typed variables.

Case 3: `iota` for a small set of states

Save as `ticket_state.go`. Consecutive integers with names beat magic `0` and `1` in a switch.

```
// ticket_state.go
package main

import "fmt"

const (
    StateOpen = iota
    StateSeated
    StateOrdered
    StatePaid
)

func label(s int) string {
    switch s {
    case StateOpen:
        return "open"
    case StateSeated:
        return "seated"
    case StateOrdered:
        return "ordered"
    case StatePaid:
        return "paid"
    default:
        return "unknown"
    }
}

func main() {
    fmt.Println(StateOpen, label(StateOpen))
    fmt.Println(StatePaid, label(StatePaid))
}
```

Run:

```
go run ticket_state.go
```

Output:

```
0 open
3 paid
```

Skip a value with `_`: `const ( _ = iota; Morning; Evening )` gives Morning = 1, Evening = 2.

Case 4: Numeric literals and rune literals

Save as `literals.go`. Underscores are for readers. Bases are `0b`, `0o`, `0x`. A rune is an integer.

```
// literals.go
package main

import "fmt"

func main() {
    cents := 1_250
    flags := 0b_0000_0101
    table := 0x0C
    euro := '€'
    fmt.Println(cents)
    fmt.Println(flags)
    fmt.Println(table)
    fmt.Printf("%q %d %U\n", euro, euro, euro)
}
```

Run:

```
go run literals.go
```

Output:

```
1250
5
12
'€' 8364 U+20AC
```

`1_250` is one thousand two hundred fifty, not "a tuple." The underscores are not digits.

Case 5: Bit flags with `1 << iota`

Save as `bit_flags.go`. When modeling permissions, roles, or operational modes, shifting `1` by `iota` creates distinct power-of-two bit flags that can be combined with bitwise OR (`|`) and tested with bitwise AND (`&`).

```
// bit_flags.go
package main

import "fmt"

const (
    PermRead  = 1 << iota // 1 << 0 == 1
    PermWrite             // 1 << 1 == 2
```

```
    PermAudit             // 1 << 2 == 4
)

func main() {
    role := PermRead | PermWrite
    hasRead := role&PermRead != 0
    hasAudit := role&PermAudit != 0

    fmt.Println("role flags:", role)
    fmt.Println("can read:", hasRead)
    fmt.Println("can audit:", hasAudit)
}
```

Run:

```
go run bit_flags.go
```

Output:

```
role flags: 3
can read: true
can audit: false
```

Bitmasks let you represent multi-attribute status in a single integer without slices or maps.

Case 6: Arbitrary precision in constant math

Save as `big_const.go`. Untyped constants in Go are evaluated at compile time with arbitrary precision (at least 256 bits of precision). Intermediate expressions can exceed the limits of `int64` without overflowing, as long as the final assigned value fits the target type.

```
// big_const.go
package main

import "fmt"

const (
    _   = 1 << (10 * iota)
    KiB // 1024
    MiB // 1048576
    GiB // 1073741824
)

func main() {
    fmt.Println("KiB:", KiB)
    fmt.Println("MiB:", MiB)
```

```
    fmt.Println("GiB:", GiB)

    // Intermediate calculation exceeds int64, but result fits in int
    const huge = (1 << 100) / (1 << 90)
    fmt.Println("huge ratio:", huge)
}
```

Run:

```
go run big_const.go
```

Output:

```
KiB: 1024
MiB: 1048576
GiB: 1073741824
huge ratio: 1024
```

`1 << 100` would immediately overflow a 64-bit variable, but in constant arithmetic it evaluates cleanly.

The trap

`iota` keeps counting even when you write a different expression. The next line **repeats that expression**, not "the next iota." Save as `iota_repeat.go`:

```
// iota_repeat.go
package main

import "fmt"

func main() {
    const (
        open    = iota // 0
        seated         // 1
        hold    = 10
        ordered        // 10 again - repeats 10, iota is unused
        paid    = iota // 4 - iota still counted the lines
    )
    fmt.Println(open, seated, hold, ordered, paid)
}
```

Run:

```
go run iota_repeat.go
```

Output:

```
0 1 10 10 4
```

`ordered` is not 3. It is 10. If you need a hole, use `_ = iota` or write every value. If the numbers must be stable on the wire, **write the numbers** (`StatePaid = 3`) and stop using `iota` for that block.

The boring rule

- `const` for values that are part of the program, not the request.
- Leave constants untyped unless a typed const documents a width.
- `iota` for dense, internal enumerations and bit flags (`1 << iota`).
- Do not use `iota` for wire protocol or database codes that must remain immutable across versions.
- Rely on compile-time arbitrary precision for byte size calculations (`1 << 20` for MiB).
- Use `_` in long numbers: `1_000_000`, not `1000000`.
- Rune literals for code points. String literals for text.
- One `const` block per family of names.

Try this

1. In `menu_const.go`, try `toastCents = 1` inside `main`. Read the compile error.
2. In `bit_flags.go`, add `PermAdmin = 1 << iota` and check if a user with `PermAdmin | PermRead` has admin privileges.
3. In `big_const.go`, calculate `TiB` as the next line in the `iota` block.
4. Add a typed `const soupCents int64 = 600` and add it to an `int` total. Convert one side.
5. In `ticket_state.go`, insert `_ = iota` as the first line so `StateOpen` becomes 1.

Variables and Scope

Variables and Scope

A name is visible in the block that declared it, and in nested blocks, until a new declaration **hides** it. The boring default is: declare at the smallest block that needs the name, and treat `:=` as a new variable, not an assignment.

Mental model

Go has three places a variable lives:

- **Package** — `var` or `const` at file top. Visible through the package.
- **Function** — parameters and names declared in the function body.
- **Block** — anything inside `{ ... }`, including `if`, `for`, and `switch`.

`=` assigns to an existing name. `:=` declares at least one new name in the **current** block. If that name already exists in an outer block, `:=` still declares a **new** inner variable. That is shadowing. It compiles. It is the trap.

Worked examples

Case 1: Package, function, block

Save as `three_scopes.go`. Prints prove which name is live.

```
// three_scopes.go
package main

import "fmt"

var desk = "front"

func main() {
    fmt.Println("package", desk)

    shift := "Amina"
    fmt.Println("function", shift)

    {
        table := 4
```

```
        fmt.Println("block", desk, shift, table)
    }
}
```

Run:

```
go run three_scopes.go
```

Output:

```
package front
function Amina
block front Amina 4
```

`table` does not exist after the inner braces. `desk` and `shift` still do.

Case 2: Inner block ends, outer name remains

Save as `block_ends.go`. The inner `n` dies with its braces.

```
// block_ends.go
package main

import "fmt"

func main() {
    n := 1
    {
        n := 2
        fmt.Println("inner", n)
    }
    fmt.Println("outer", n)
}
```

Run:

```
go run block_ends.go
```

Output:

```
inner 2
outer 1
```

Two variables named `n`. The inner one is not an assignment to the outer one.

Case 3: `:=` in `if` is a new block

Save as `if_init.go`. The short statement on `if` belongs to the `if` (and its `else`), not to the rest of `main`.

```
// if_init.go
package main

import "fmt"

func tableFor(name string) (int, bool) {
    if name == "window" {
        return 4, true
    }
    return 0, false
}

func main() {
    if n, ok := tableFor("window"); ok {
        fmt.Println("seated at", n)
    }
    if n, ok := tableFor("patio"); ok {
        fmt.Println("seated at", n)
    } else {
        fmt.Println("no table", n, ok)
    }
}
```

Run:

```
go run if_init.go
```

Output:

```
seated at 4
no table 0 false
```

`n` and `ok` in the `else` are the same inner pair from that `if`. They are gone on the next line of `main`. That is why you can write two `if n, ok := ...` in a row.

Case 4: `:=` vs `=` in the same function

Save as `assign_or_declare.go`. After `err` exists, `=` updates it. A later `:=` with a **new** name on the left still redeclares `err` if you are in a new block.

```
// assign_or_declare.go
package main

import (
    "fmt"
    "strconv"
)

func main() {
    n, err := strconv.Atoi("12")
    fmt.Println("first", n, err)

    n, err = strconv.Atoi("x")
    fmt.Println("assign", n, err)

    if n, err := strconv.Atoi("4"); err == nil {
        fmt.Println("if", n, err)
    }
    fmt.Println("after if", n, err)
}
```

Run:

```
go run assign_or_declare.go
```

Output:

```
first 12 <nil>
assign 0 strconv.Atoi: parsing "x": invalid syntax
if 4 <nil>
after if 0 strconv.Atoi: parsing "x": invalid syntax
```

The `if` parsed `"4"` successfully. `main`'s `n` and `err` are still the failed `"x"` parse. Look at `after if`.

The trap

Short declaration in a nested block is the Monday-morning bug. You think you updated `err`. You declared a second `err`. Save as `shadow_err.go`:

```
// shadow_err.go
package main

import (
    "fmt"
    "strconv"
```

```
)

func main() {
    table, err := strconv.Atoi("4")
    if err != nil {
        fmt.Println("bad table")
        return
    }

    if cents, err := strconv.Atoi("not-a-price"); err != nil {
        fmt.Println("inner saw", err)
    } else {
        fmt.Println("price", cents)
    }

    if err != nil {
        fmt.Println("outer still set:", err)
        return
    }
    fmt.Println("open table", table)
}
```

Run:

```
go run shadow_err.go
```

Output:

```
inner saw strconv.Atoi: parsing "not-a-price": invalid syntax
open table 4
```

The inner `err` is not `main`'s `err`. The outer `if err != nil` is false, so the desk "opens" after a failed price parse. The fix: use `=` when the name already exists in this block (`cents, err = strconv.Atoi(...)` after declaring `cents`), or return when the inner call fails without introducing a new `err`.

The boring rule

- Smallest block that needs the name.
- `:=` means "new variable in this block." `=` means "update."
- Never `:=` an `err` you already have in the same function unless you are sure you want a second one.
- `if v := ...; cond` keeps `v` out of the rest of the function — that is usually what you want.
- Package-level `var` for process-wide state only. Prefer passing values.
- If a print would surprise you, the name is shadowed. Print it.

Try this

1. In `block_ends.go`, change the inner line to `n = 2` (assignment). Both prints should show `2`.
2. In `if_init.go`, try `fmt.Println(n)` after the second `if`. It must not compile.
3. In `shadow_err.go`, declare `var cents int` before the inner `if` and use `cents, err = strconv.Atoi(...)` so the failed parse is visible to the outer `if`.
4. Add a package-level `var shift = "desk"` and a `shift := "Bo"` in `main`. Print both by moving the inner name (`s := "Bo"`) so you can still see the package value.

Control Flow

Conditional Logic

Conditional Logic

Go has `if` and `switch`. There is no ternary operator. The boring default is a **guard clause** at the top of a function (bail out on the bad case) and a `switch` when you have more than two named states.

Mental model

`if` can start with a short statement: `if v, err := f(); err != nil`. That statement's names live in the `if` and `else` only.

`switch` compares one value to a list of cases (**tagged**), or lists boolean conditions with no value (**tagless**). Cases do **not** fall through unless you write `fallthrough`. There is no `?:`. Write `if` and assign.

Worked examples

Case 1: `if` with an init statement

Save as `open_if.go`. Look up a table; only seat when the lookup worked.

```
// open_if.go
package main

import "fmt"

func tableFor(name string) (int, bool) {
    if name == "window" {
        return 4, true
    }
    if name == "counter" {
        return 1, true
    }
    return 0, false
}

func main() {
    if n, ok := tableFor("window"); ok {
        fmt.Println("seat at", n)
```

```
    } else {
        fmt.Println("no table")
    }
}
```

Run:

```
go run open_if.go
```

Output:

```
seat at 4
```

The init form keeps `n` and `ok` out of the rest of `main`. That is the same idea as the scope chapter, now used as the normal way to write `if`.

Case 2: Tagged `switch` on a state

Save as `ticket_switch.go`. One value, several names. `default` is the leftover.

```
// ticket_switch.go
package main

import "fmt"

func next(state string) string {
    switch state {
    case "open":
        return "seat the guest"
    case "seated":
        return "take the order"
    case "ordered":
        return "send the kitchen"
    case "paid":
        return "clear the table"
    default:
        return "unknown state: " + state
    }
}

func main() {
    fmt.Println(next("seated"))
    fmt.Println(next("lost"))
}
```

Run:

```
go run ticket_switch.go
```

Output:

```
take the order
unknown state: lost
```

Several values can share a case: `case "open", "seated":`. No extra `break` is required.

Case 3: Tagless `switch`

Save as `shift_switch.go`. Each case is a condition. First match wins.

```
// shift_switch.go
package main

import "fmt"

func pay(hours int, night bool) int {
    switch {
    case hours <= 0:
        return 0
    case night:
        return hours * 1800
    case hours > 8:
        return 8*1500 + (hours-8)*2250
    default:
        return hours * 1500
    }
}

func main() {
    fmt.Println(pay(6, false))
    fmt.Println(pay(10, false))
    fmt.Println(pay(6, true))
}
```

Run:

```
go run shift_switch.go
```

Output:

```
9000
16500
10800
```

This is the replacement for a chain of `else if` when the conditions are the story.

Case 4: Guard clauses, not a triangle of `if`

Save as `charge.go`. Bad input returns first. The happy path stays left-aligned.

```
// charge.go
package main

import (
    "fmt"
    "os"
)

func charge(item string, cents int) error {
    if item == "" {
        return fmt.Errorf("item is empty")
    }
    if cents <= 0 {
        return fmt.Errorf("%s: cents must be positive", item)
    }
    fmt.Printf("charged %s %d cents\n", item, cents)
    return nil
}

func main() {
    if err := charge("toast", 350); err != nil {
        fmt.Fprintln(os.Stderr, err)
    }
    if err := charge("tea", 0); err != nil {
        fmt.Fprintln(os.Stderr, err)
    }
}
```

Run:

```
go run charge.go
```

Output:

```
charged toast 350 cents
tea: cents must be positive
```

The nested version (`if item != "" { if cents > 0 { ... } }`) does the same work and hides the success path. Prefer the guards.

Go has no `cents > 0 ? cents : 0`. Write `if cents < 0 { cents = 0 }`.

The trap

`fallthrough` goes to the **next case body**, even if that case's match would have failed. Save as `fallthrough.go`:

```
// fallthrough.go
package main

import "fmt"

func main() {
    state := "open"
    switch state {
    case "open":
        fmt.Println("seat")
        fallthrough
    case "seated":
        fmt.Println("hand menu")
    case "paid":
        fmt.Println("should not print for open")
    }
}
```

Run:

```
go run fallthrough.go
```

Output:

```
seat
hand menu
```

`"open"` is not `"seated"`. `fallthrough` still ran the seated body. It did not run `paid` because that case has no `fallthrough` above it. If you want two states to share work, list them on one `case` or call a function. Leave `fallthrough` for the rare C-style dispatch you can name in a comment.

The boring rule

- Guard first. Return the error. Keep the happy path flat.
- `if init; cond` when the value is only for that branch.
- `switch` on a name when the list is known. Tagless `switch` when the list is conditions.
- No ternary. An `if` is two lines and reads in one pass.
- Do not write `fallthrough` unless the next body must always run.
- `default` for the unexpected state, not for the common one.

Try this

1. In `open_if.go`, look up `"counter"` and `"patio"`. Print both.
2. In `ticket_switch.go`, combine `"open"` and `"seated"` on one case and return `"on the floor"`.
3. In `charge.go`, add a guard that rejects `cents > 100_000` as "too large."
4. Delete `fallthrough` from `fallthrough.go` and confirm only `seat` prints.

Loops and Iteration

Loops and Iteration

Go has one loop: `for`. The boring default is a C-style `for` when you need an index you control, `for i := range n` when you need `0..n-1`, and `for _, v := range s` when you need the elements. Do not range a map if the **order** of keys is the behaviour.

Mental model

Every loop is `for`. There is no `while` and no `do`. The condition-only form is `for cond { }`. The infinite form is `for { }`.

Since Go 1.22, `for i := range n` walks `0, 1, …, n-1` for an integer `n`. `range` over a slice gives index and value. `range` over a string gives **byte index** and **rune**. `range` over a map gives key and value in an order the runtime does not promise.

`break` leaves the inner-most loop or switch. A **label** names which loop to leave. `continue` skips to the next iteration of that loop.

Worked examples

Case 1: The three `for` shapes

Save as `for_shapes.go`. A counter, a condition, and an infinite loop that `break`s.

```
// for_shapes.go
package main

import "fmt"

func main() {
    for i := 1; i <= 3; i++ {
        fmt.Println("table", i)
    }

    n := 3
    for n > 0 {
        fmt.Println("left", n)
        n--
    }
```

```
    k := 0
    for {
        k++
        if k == 2 {
            continue
        }
        fmt.Println("tick", k)
        if k >= 3 {
            break
        }
    }
}
```

Run:

```
go run for_shapes.go
```

Output:

```
table 1
table 2
table 3
left 3
left 2
left 1
tick 1
tick 3
```

`continue` skipped `tick 2`. `break` stopped at 3.

Case 2: `for i := range n`

Save as `range_n.go`. Integer range is the boring way to say "three times" without inventing a dummy slice.

```
// range_n.go
package main

import "fmt"

func main() {
    for i := range 3 {
        fmt.Println("cover", i)
    }
}
```

Run:

```
go run range_n.go
```

Output:

```
cover 0
cover 1
cover 2
```

`range 0` runs zero times. A negative integer panics.

Case 3: Range a slice, then a string of runes

Save as `range_slice.go`. The string is `"café"`: four runes, five bytes. The index on `é` is 3, not 4.

```
// range_slice.go
package main

import "fmt"

func main() {
    items := []string{"toast", "tea", "soup"}
    for i, item := range items {
        fmt.Println(i, item)
    }

    for i, r := range "café" {
        fmt.Printf("byte %d rune %q\n", i, r)
    }
}
```

Run:

```
go run range_slice.go
```

Output:

```
0 toast
1 tea
2 soup
byte 0 rune 'c'
byte 1 rune 'a'
byte 2 rune 'f'
byte 3 rune 'é'
```

If you only want values, write `for _, item := range items`. If you only want indexes, `for i := range items`.

Case 4: Labels when `break` is not enough

Save as `label_break.go`. Two nested loops: stop the **outer** walk when the kitchen hits a stop ticket.

```
// label_break.go
package main

import "fmt"

func main() {
    tables := []int{4, 7, 9}
    items := []string{"toast", "STOP", "tea"}

outer:
    for _, table := range tables {
        for _, item := range items {
            if item == "STOP" {
                fmt.Println("stop at table", table)
                break outer
            }
            fmt.Println("table", table, item)
        }
    }
    fmt.Println("done")
}
```

Run:

```
go run label_break.go
```

Output:

```
table 4 toast
stop at table 4
done
```

Without the label, `break` would only leave the inner loop and table 7 would still print. Labels are rare. Nested loops that need them are a hint to pull a function out.

The trap

Ranging a map is fine for "visit every key." It is wrong for "print in the order we inserted" or "first key wins" if that order is a business rule. Save as `map_order.go`:

```
// map_order.go
package main

import (
    "fmt"
    "sort"
)

func main() {
    prices := map[string]int{
        "tea":   250,
        "toast": 350,
        "soup":  600,
    }

    fmt.Println("one walk:")
    for item, cents := range prices {
        fmt.Println(item, cents)
    }

    names := make([]string, 0, len(prices))
    for item := range prices {
        names = append(names, item)
    }
    sort.Strings(names)
    fmt.Println("sorted:")
    for _, item := range names {
        fmt.Println(item, prices[item])
    }
}
```

Run:

```
go run map_order.go
```

Possible output (the first block may shuffle; the second block is stable):

```
one walk:
soup 600
tea 250
toast 350
sorted:
```

```
soup 600
tea 250
toast 350
```

If a test asserts the first block's order, it will flake. Sort the keys when order matters. Do not "fix" it by ranging until the order looks nice.

The boring rule

- `for` is the only loop. Pick the form that names what you are walking.
- `for i := range n` for a count (Go 1.22+; this book is Go 1.27).
- `for _, v := range s` for elements. Keep the index when you need it.
- Range strings for runes. Index strings for bytes (next part).
- Never depend on map range order. Sort keys, or use a slice of keys you own.
- Labels last. A function is clearer than `break outer` if the inner body is more than a few lines.

Try this

1. In `range_n.go`, print `i+1` so covers are 1, 2, 3.
2. In `range_slice.go`, add a `for i := range items` loop that prints only indexes.
3. In `label_break.go`, replace `break outer` with `continue outer` and explain the new output (table 7 and 9 still run until STOP).
4. In `map_order.go`, run twice. Confirm the sorted block is identical and the first block may not be.

Defer and Cleanup

Defer and Cleanup

`defer` schedules a function call to run when the surrounding function returns — no matter how it returns. The boring default is one line: `defer f.Close()` right after `f` is opened. That one line makes early returns, error paths, and normal returns all safe. Clever uses of `defer` exist; most of them are not worth it.

Mental model

A `defer` statement pushes a call onto a **defer stack**. When the surrounding function exits — by `return`, by falling off the end, or by a panicking runtime — the calls pop off last-in first-out (LIFO). Multiple defers run in reverse order of their appearance.

Arguments are evaluated immediately, at the defer statement, not at call time. The deferred call receives the value that the argument had when `defer` ran, even if the variable changes later.

```
x := 1
defer fmt.Println(x)   // captures 1 right now
x = 99
// prints: 1, not 99
```

Named return values are the exception. A deferred function can read and write the names declared in the result list. That is the one way a deferred call can change what the caller receives. Everything else about named results is optional style; this interaction is their real job.

The defer stack is per-function, not per-block. A `defer` inside a `for` loop is attached to the enclosing function, not to the loop body. Files deferred in a loop do not close until the whole function returns.

Worked examples

Case 1: Closing a file — without defer, then with

The leak first. Save as `ticket_log_leak.go`. `writeTicket` opens a temp file and writes a line. If the write fails, the early return leaks the open file handle.

```
// ticket_log_leak.go
package main

import (
```

```
    "fmt"
    "os"
)

// writeTicketLeaky opens a file and may leak it on error.
func writeTicketLeaky(id int, note string) error {
    f, err := os.CreateTemp("", "ticket-*.log")
    if err != nil {
        return fmt.Errorf("open: %w", err)
    }
    // BUG: if Fprintf fails, we return without closing f.
    if _, err := fmt.Fprintf(f, "ticket %d: %s\n", id, note); err != nil {
        return fmt.Errorf("write ticket %d: %w", id, err)
    }
    fmt.Println("wrote", f.Name())
    return f.Close()
}

func main() {
    if err := writeTicketLeaky(42, "printer jam"); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}
```

Run:

```
go run ticket_log_leak.go
```

Output:

```
wrote /tmp/ticket-3421760645.log
```

It works here — but the file handle leaks whenever the write fails. On a busy desk creating thousands of ticket logs, the process runs out of file descriptors.

Now the fix. Save as `ticket_log.go`. One `defer` right after the file opens. Every return path is covered.

```
// ticket_log.go
package main

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

```
func writeTicket(id int, note string) error {
    f, err := os.CreateTemp("", "ticket-*.log")
    if err != nil {
        return fmt.Errorf("open: %w", err)
    }
    defer f.Close() // runs on every exit path from here

    if _, err := fmt.Fprintf(f, "ticket %d: %s\n", id, note); err != nil {
        return fmt.Errorf("write ticket %d: %w", id, err)
    }
    fmt.Println("wrote", f.Name())
    return nil
}

func main() {
    if err := writeTicket(42, "printer jam"); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}
```

Run:

```
go run ticket_log.go
```

Output:

```
wrote /tmp/ticket-2089451234.log
```

The rule: open a resource, check the error, then immediately defer its cleanup before doing anything else. Never let code grow between the open and the defer.

Case 2: Unlock after lock — a shared ticket counter

The desk tracks open tickets in a shared counter. Multiple goroutines update it. Save as `ticket_counter.go`. `defer mu.Unlock()` right after `mu.Lock()` guarantees the lock releases on every path, including panics that recover elsewhere.

```
// ticket_counter.go
package main

import (
    "fmt"
    "sync"
)
```

```
type Desk struct {
    mu     sync.Mutex
    open   int
    closed int
}

func (d *Desk) Open() {
    d.mu.Lock()
    defer d.mu.Unlock()
    d.open++
}

func (d *Desk) Close() {
    d.mu.Lock()
    defer d.mu.Unlock()
    if d.open > 0 {
        d.open--
    }
    d.closed++
}

func (d *Desk) Stats() (open, closed int) {
    d.mu.Lock()
    defer d.mu.Unlock()
    return d.open, d.closed
}

func main() {
    var d Desk
    var wg sync.WaitGroup

    for range 50 {
        wg.Add(1)
        wg.Go(func() {
            defer wg.Done()
            d.Open()
            d.Close()
        })
    }
    wg.Wait()

    open, closed := d.Stats()
    fmt.Printf("open: %d  closed: %d\n", open, closed)
}
```

Run:

```
go run ticket_counter.go
```

Output:

```
open: 0  closed: 50
```

Writing `mu.Lock(); defer mu.Unlock()` on two adjacent lines is a pattern: never put anything between them. If you forget `defer` and an early return skips the `Unlock`, every subsequent `Lock` blocks forever.

Case 3: defer in a loop — the fix

Processing a batch of shift reports means opening each file, reading it, and closing it before moving to the next. If you defer inside the loop, none of the files close until the function returns. On a large batch that exhausts the process file descriptor limit. Save as `shift_report.go`.

```
// shift_report.go
package main

import (
    "fmt"
    "os"
)

// processShifts opens each report file and reads its size.
// The loop body is an inner function so defer closes the file
// after each iteration, not after the whole batch.
func processShifts(paths []string) error {
    for _, path := range paths {
        if err := processOne(path); err != nil {
            return err
        }
    }
    return nil
}

func processOne(path string) error {
    f, err := os.Open(path)
    if err != nil {
        return fmt.Errorf("open shift report: %w", err)
    }
    defer f.Close() // closes after processOne returns, i.e. each iteration

    info, err := f.Stat()
    if err != nil {
        return fmt.Errorf("stat %s: %w", path, err)
```

```
    }
    fmt.Printf("shift report %s: %d bytes\n", path, info.Size())
    return nil
}

func main() {
    // Create two temp files to act as shift reports.
    a, _ := os.CreateTemp("", "shift-*.txt")
    fmt.Fprintln(a, "shift A: 8h")
    a.Close()

    b, _ := os.CreateTemp("", "shift-*.txt")
    fmt.Fprintln(b, "shift B: 6h")
    b.Close()

    if err := processShifts([]string{a.Name(), b.Name()}); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }

    os.Remove(a.Name())
    os.Remove(b.Name())
}
```

Run:

```
go run shift_report.go
```

Output:

```
shift report /tmp/shift-1027364518.txt: 12 bytes
shift report /tmp/shift-2093847610.txt: 10 bytes
```

The pattern: extract the per-iteration work into its own named function (`processOne`). The `defer` inside that function closes the file after each call. The loop itself stays clean.

Case 4: Named results and defer — adding context to errors

A deferred function that modifies a named return value changes what the caller receives. This is the one place where named returns earn their keep: annotating errors at the exit point without repeating `fmt.Errorf` on every path. Save as `open_ticket.go`.

```
// open_ticket.go
package main

import (
```

```
    "errors"
    "fmt"
    "os"
)

var ErrFull = errors.New("desk is full")

// openTicket returns a named error so the deferred annotator can wrap it.
func openTicket(id int, deskSize int) (err error) {
    defer func() {
        if err != nil {
            // err is the named result; writing to it changes what the caller sees.
            err = fmt.Errorf("openTicket(%d): %w", id, err)
        }
    }()

    if id <= 0 {
        return fmt.Errorf("id must be positive, got %d", id)
    }
    if deskSize <= 0 {
        return ErrFull
    }

    fmt.Printf("ticket %d opened, desk has %d slots\n", id, deskSize)
    return nil
}

func main() {
    // Happy path.
    if err := openTicket(7, 3); err != nil {
        fmt.Fprintln(os.Stderr, err)
    }

    // Bad id.
    if err := openTicket(-1, 3); err != nil {
        fmt.Fprintln(os.Stderr, err)
    }

    // Desk full - wrapping preserves the sentinel.
    err := openTicket(8, 0)
    fmt.Println("is ErrFull:", errors.Is(err, ErrFull))
    fmt.Fprintln(os.Stderr, err)
}
```

Run:

```
go run open_ticket.go
```

Output:

```
ticket 7 opened, desk has 3 slots
openTicket(-1): id must be positive, got -1
is ErrFull: true
openTicket(8): desk is full
```

Three things to notice:

1. The function signature is `(err error)` — named result.
2. The deferred closure reads `err` after the `return` statement has set it.
3. `errors.Is` still works because the annotator uses `%w`, which wraps rather than replaces.

Wrapping preserves the sentinel so callers can still test `errors.Is(err, ErrFull)`. If you used `fmt.Errorf("... %v", err)` instead of `%w`, the wrapping would be lost.

When to use this pattern. Only when every error from the function needs the same context prefix. If only some errors need it, add `fmt.Errorf` at each return.

The trap

Deferring inside a loop without an inner function. The deferred calls pile up and execute only when the enclosing function returns. Save as `loop_leak.go` to see the problem:

```go
// loop_leak.go
package main

import (
    "fmt"
    "os"
)

// BUG: all files stay open until dumpReports returns.
func dumpReports(paths []string) {
    for _, path := range paths {
        f, err := os.Open(path)
        if err != nil {
            fmt.Fprintln(os.Stderr, err)
            continue
        }
        defer f.Close() // deferred to dumpReports, not to the loop iteration
        info, _ := f.Stat()
        fmt.Printf("%s: %d bytes\n", path, info.Size())
    }
    // All f.Close() calls run here, after all files are already open at once.
}
```

```
func main() {
    a, _ := os.CreateTemp("", "report-*.txt")
    fmt.Fprintln(a, "data")
    a.Close()

    b, _ := os.CreateTemp("", "report-*.txt")
    fmt.Fprintln(b, "data")
    b.Close()

    dumpReports([]string{a.Name(), b.Name()})
    os.Remove(a.Name())
    os.Remove(b.Name())
}
```

Run:

```
go run loop_leak.go
```

Output:

```
/tmp/report-1234567890.txt: 5 bytes
/tmp/report-9876543210.txt: 5 bytes
```

The output looks correct, but both files are open simultaneously throughout `dumpReports`. With a thousand shift reports that is a thousand open file descriptors at once. The fix from Case 3 applies: move the body into a separate function and call it from the loop.

The boring rule

- `defer` for cleanup: `Close`, `Unlock`, `cancel`. Place it immediately after the resource opens, before any other logic.
- Never put code between the open and the `defer`.
- Do not `defer` inside a tight loop. Handles pile up until the enclosing function returns. Extract the loop body into a named function instead.
- Use the named-result + defer pattern only when every error from a function needs the same annotation prefix.
- Prefer a flat `return f.Close()` at the end of a function when there is only one exit path and no error to annotate — it is simpler and the close error is not silently discarded.
- Arguments to `defer` are captured at the defer line. If you need the value at exit time, use a closure (`defer func() { use(x) }()`).
- No clever logic in deferred calls. A deferred function should do one obvious thing.

Try this

1. `ticket_log.go`: Add a second deferred call that prints `"cleanup done for ticket N"` before `defer f.Close()`. Confirm which prints first — the close or the message — and explain why.
2. `ticket_counter.go`: Add a `Reset` method that sets both counters to zero behind the mutex. Run with `go run -race ticket_counter.go`. Confirm no data race is reported.
3. `shift_report.go`: Change `processOne` to return the file size as a second result `(int64, error)`. Accumulate the total bytes across all files in `processShifts` and print a summary line.
4. `open_ticket.go`: Add a condition `id > 9999` that returns `fmt.Errorf("id %d exceeds max", id)`. Confirm the annotator wraps it. Then change the annotator from `%w` to `%v` and show that `errors.Is(err, ErrFull)` now returns `false`.

Composite Types

Arrays and Slices

Arrays and Slices

An array is a value with a fixed length in its type. A slice is a view: a pointer, a length, and a capacity. The boring default is slices everywhere, `make` when you know the length, `append` when you do not, and `copy` when the new slice must not share memory.

Mental model

`[3]int` and `[4]int` are different types. Assigning an array copies every element.

A slice header is small. Two slices can point at the **same backing array**. A change through one is visible through the other. `append` writes into leftover capacity when it can; only when capacity is full does it allocate a new array.

`len` is how many elements you may index. `cap` is how far `append` can go before it reallocates.

Worked examples

Case 1: Arrays copy, slices share

Save as `array_copy.go`. The array keeps its old first slot. The slice does not.

```
// array_copy.go
package main

import "fmt"

func main() {
    var seats [3]int
    seats[0] = 2
    seats[1] = 4
    seats[2] = 2
    other := seats
    other[0] = 8
    fmt.Println("array", seats, other)

    open := []int{2, 4, 2}
    alias := open
    alias[0] = 8
    fmt.Println("slice", open, alias)
```

```
}
```

Run:

```
go run array_copy.go
```

Output:

```
array [2 4 2] [8 4 2]
slice [8 4 2] [8 4 2]
```

You will almost never put an array in desk code. You will put slices in desk code, and you will forget they share.

Case 2: `make`, `len`, `cap`

Save as `make_slice.go`. Length 0, capacity 2: room for two `append`s before growth.

```
// make_slice.go
package main

import "fmt"

func main() {
    items := make([]string, 0, 2)
    fmt.Printf("len=%d cap=%d %v\n", len(items), cap(items), items)
    items = append(items, "toast")
    fmt.Printf("len=%d cap=%d %v\n", len(items), cap(items), items)
    items = append(items, "tea")
    fmt.Printf("len=%d cap=%d %v\n", len(items), cap(items), items)
    items = append(items, "soup")
    fmt.Printf("len=%d cap=%d %v\n", len(items), cap(items), items)
}
```

Run:

```
go run make_slice.go
```

Output:

```
len=0 cap=2 []
len=1 cap=2 [toast]
len=2 cap=2 [toast tea]
len=3 cap=4 [toast tea soup]
```

The third `append` needed a new backing array. Capacity doubled here. Do not hard-code growth numbers; they are an implementation detail. Do depend on `len`.

Case 3: Slicing is a window, not a copy

Save as `window.go`. `tickets[1:3]` is length 2. Index 0 of the window is the original index 1.

```
// window.go
package main

import "fmt"

func main() {
    tickets := []int{10, 20, 30, 40}
    mid := tickets[1:3]
    fmt.Println("mid", mid, "len", len(mid), "cap", cap(mid))
    fmt.Println("mid[0]", mid[0])
}
```

Run:

```
go run window.go
```

Output:

```
mid [20 30] len 2 cap 3
mid[0] 20
```

`cap` is 3 because the backing array still has `40` after the window. `mid[2]` panics (`len` is 2), but `append(mid, 99)` would write into that leftover slot.

Case 4: `copy` owns its own array

Save as `copy_tickets.go`. `copy` returns how many elements moved: the min of the two lengths.

```
// copy_tickets.go
package main

import "fmt"

func main() {
    tickets := []int{10, 20, 30}
    out := make([]int, len(tickets))
    n := copy(out, tickets)
    out[0] = 99
    fmt.Println("copied", n)
    fmt.Println("orig", tickets)
    fmt.Println("out", out)
}
```

Run:

```
go run copy_tickets.go
```

Output:

```
copied 3
orig [10 20 30]
out [99 20 30]
```

`append([]int(nil), tickets...)` also copies. Use whichever you can read at speed.

Case 5: The 3-index slice expression (`s[low:high:max]`)

Save as `full_slice.go`. Slicing normally keeps the capacity of the original array up to its end. A **full slice expression** adds a third index: `s[low:high:max]`. This sets capacity explicitly to `max - low`.

```
// full_slice.go
package main

import "fmt"

func main() {
    tickets := []int{10, 20, 30, 40}

    // 3-index slice: tickets[low:high:max]
    // low=0, high=2, max=2 -> len=2, cap=2
    window := tickets[0:2:2]
    fmt.Printf("window: len=%d cap=%d\n", len(window), cap(window))

    // Because cap is reached, append MUST allocate a new backing array!
    window = append(window, 99)
    fmt.Println("window after append:", window)
    fmt.Println("original tickets untouched:", tickets)
}
```

Run:

```
go run full_slice.go
```

Output:

```
window: len=2 cap=2
window after append: [10 20 99]
original tickets untouched: [10 20 30 40]
```

By clamping capacity to length with `tickets[0:2:2]`, any subsequent `append` is forced to allocate a fresh backing array. You protect `tickets[2]` from accidental overwrite without copying eagerly.

Case 6: Deleting elements with `slices.Delete`

Save as `delete_slice.go`. Go does not have a built-in `delete` keyword for slices. The standard library provides `slices.Delete(s, i, j)`, which shifts trailing elements left and zeroes out the discarded slots.

```
// delete_slice.go
package main

import (
    "fmt"
    "slices"
)

func main() {
    orders := []string{"toast", "tea", "soup", "coffee"}

    // Delete index 1 ("tea"): removes elements in orders[1:2]
    orders = slices.Delete(orders, 1, 2)
    fmt.Printf("after delete: %v (len=%d cap=%d)\n", orders, len(orders), cap(orders))
}
```

Run:

```
go run delete_slice.go
```

Output:

```
after delete: [toast soup coffee] (len=3 cap=4)
```

`slices.Delete` zeroes out the vacated elements at the end of the slice before truncating `len`, ensuring pointers or references do not linger in memory.

The trap

A subslice plus `append` can overwrite the original. This is the slice bug that survives code review. Save as `alias_append.go`:

```
// alias_append.go
package main

import "fmt"
```

```
func main() {
    tickets := []int{10, 20, 30, 40}
    window := tickets[0:2]
    window[0] = 99
    fmt.Println("after index", tickets, window)

    window = append(window, 70)
    fmt.Println("after append", tickets, window)
}
```

Run:

```
go run alias_append.go
```

Output:

```
after index [99 20 30 40] [99 20]
after append [99 20 70 40] [99 20 70]
```

`window` had capacity left, so `append` wrote `70` into `tickets[2]`. The kitchen ticket `30` is gone.

The fix is a copy before you append, `append` onto nil, or the 3-index slice expression from Case 5:

```
// alias_copy.go
package main

import "fmt"

func main() {
    tickets := []int{10, 20, 30, 40}
    window := append([]int(nil), tickets[0:2]...)
    window = append(window, 70)
    fmt.Println("tickets", tickets)
    fmt.Println("window", window)
}
```

Run:

```
go run alias_copy.go
```

Output:

```
tickets [10 20 30 40]
window [10 20 70]
```

`tickets[2]` is still `30`.

The boring rule

- Use slices. Use arrays only when the length is the type (a hash, a pixel).
- `make([]T, n)` when you will fill `0..n-1`. `make([]T, 0, n)` when you will `append` up to `n`.
- Always keep the result of `append`: `s = append(s, v)`.
- A slice expression shares. `copy`, `append` onto nil, or a 3-index slice `s[i:j:j]` when passing sub-slices downstream.
- Use `slices.Delete` to remove elements safely without leaking trailing pointer references.
- Never `append` to a subslice you still consider a window onto the original.
- `len` is the contract. `cap` is a hint.

Try this

1. In `array_copy.go`, pass the array to a function that sets `seats[0] = 0`. Print after the call: the caller's array is unchanged (the parameter was a copy).
2. In `full_slice.go`, change `tickets[0:2:2]` to `tickets[0:2:4]` and observe that `append` overwrites `tickets[2]`.
3. In `delete_slice.go`, delete the first element (`orders[0:1]`) and print `orders`.
4. In `make_slice.go`, start with `make([]string, 2)` (length 2, zeros) and assign `items[0] = "toast"` instead of `append`.
5. In `alias_append.go`, replace `tickets[0:2]` with `tickets[0:2:2]` and run again. Notice that `tickets[2]` remains untouched.

Working with Strings

Working with Strings

A Go string is an immutable sequence of bytes, usually UTF-8. The boring default is: range for runes, `len` for bytes, `unicode/utf8` when you need a count of characters, and `strings.Builder` when you are assembling text in a loop.

Mental model

`len(s)` is the number of **bytes**. Indexing `s[i]` is a `byte`. `for i, r := range s` walks **runes** and gives the byte index of each.

You cannot write `s[0] = 'x'`. Conversion `[]byte(s)` copies. Conversion `string(bytes)` copies. `strings.Builder` is the standard library's append-only buffer that becomes a string once.

Worked examples

Case 1: Bytes, not letters

Save as `string_len.go`. `"café"` is four letters and five bytes because `é` is `C3 A9` in UTF-8.

```
// string_len.go
package main

import (
    "fmt"
    "unicode/utf8"
)

func main() {
    s := "café"
    fmt.Println("bytes", len(s))
    fmt.Println("runes", utf8.RuneCountInString(s))
    fmt.Printf("s[3] byte %d\n", s[3])
}
```

Run:

```
go run string_len.go
```

Output:

```
bytes 5
runes 4
s[3] byte 195
```

`s[3]` is the first byte of `é` (0xC3 = 195), not the letter `é`. Do not index a string to get "the fourth character" unless you have proven the text is ASCII.

Case 2: `range` walks runes

Save as `range_runes.go`. The index jumps by the size of each encoding.

```
// range_runes.go
package main

import "fmt"

func main() {
    for i, r := range "café" {
        fmt.Printf("%d %q %U\n", i, r, r)
    }
}
```

Run:

```
go run range_runes.go
```

Output:

```
0 'c' U+0063
1 'a' U+0061
2 'f' U+0066
3 'é' U+00E9
```

There is no index 4 in this loop. The last rune starts at byte 3 and occupies two bytes.

Case 3: The `strings` package

Save as `strings_menu.go`. Split, trim, join, contains. These copy; they do not edit in place.

```
// strings_menu.go
package main

import (
    "fmt"
    "strings"
)
```

```
func main() {
    line := "  toast, tea, soup  "
    line = strings.TrimSpace(line)
    parts := strings.Split(line, ", ")
    fmt.Println(parts)
    fmt.Println(strings.Join(parts, " | "))
    fmt.Println("has tea", strings.Contains(line, "tea"))
    fmt.Println("upper", strings.ToUpper(parts[0]))
}
```

Run:

```
go run strings_menu.go
```

Output:

```
[toast tea soup]
toast | tea | soup
has tea true
upper TOAST
```

`TrimSpace` returns a new string. The original literal is still the original literal.

Case 4: `strings.Builder` in a loop

Save as `builder.go`. Do not `s = s + piece` inside a hot loop; each `+` copies the whole string. A builder grows a byte buffer.

```
// builder.go
package main

import (
    "fmt"
    "strings"
)

func main() {
    items := []string{"toast", "tea", "soup"}
    var b strings.Builder
    for i, item := range items {
        if i > 0 {
            b.WriteString(", ")
        }
        fmt.Fprintf(&b, "%d:%s", i+1, item)
    }
    fmt.Println(b.String())
```

```
}
```

Run:

```
go run builder.go
```

Output:

```
1:toast, 2:tea, 3:soup
```

`fmt.Fprintf(&b, ...)` works because `Builder` implements `io.Writer`. `b.String()` is the snapshot.

Case 5: Conversions copy

Save as `bytes_copy.go`. Mutating the slice does not change the string.

```
// bytes_copy.go
package main

import "fmt"

func main() {
    s := "tea"
    buf := []byte(s)
    buf[0] = 'T'
    fmt.Println("string", s)
    fmt.Println("bytes", string(buf))
}
```

Run:

```
go run bytes_copy.go
```

Output:

```
string tea
bytes Tea
```

`string(buf)` is another copy. After that line, more writes to `buf` would not change the new string either.

The trap

Indexing through a UTF-8 string as if it were an array of letters cuts a rune in half. Save as `cut_rune.go`:

```
// cut_rune.go
package main

import "fmt"

func main() {
    s := "café"
    fmt.Printf("s[:3] = %q\n", s[:3])
    fmt.Printf("s[:4] = %q\n", s[:4])
    fmt.Printf("range last = ")
    var last rune
    for _, r := range s {
        last = r
    }
    fmt.Printf("%q\n", last)
}
```

Run:

```
go run cut_rune.go
```

Output:

```
s[:3] = "caf"
s[:4] = "caf\xc3"
range last = 'é'
```

`s[:4]` ends in the middle of `é`. The printed `Ã` is the stray `C3` byte shown as Latin-1. The fix: range, or `utf8.DecodeLastRuneInString`, or convert to `[]rune` when you truly need character indexes (`[]rune(s)[3]` is `é`, and it copies).

The boring rule

- Treat strings as UTF-8 bytes. `len` is bytes.
- Range for characters. Index for bytes you have measured.
- `strings` for search, split, join, trim. Do not write those loops.
- `strings.Builder` (or `bytes.Buffer`) when building in a loop.
- `[]byte(s)` and `string(buf)` copy. Do not expect aliasing.
- Keep prices and IDs out of strings until you print them.

Try this

1. In `string_len.go`, use `"tea 🍵"` and print bytes vs runes (the emoji is four UTF-8 bytes).
2. In `strings_menu.go`, `strings.ReplaceAll(line, "tea", "chai")` and print.
3. In `builder.go`, call `b.Grow(64)` before the loop. Behaviour stays the same; one allocation is reserved up front.
4. Convert `"café"` to `[]rune`, change index `3`, convert back, and print. Compare with `s[:4]`.

Maps

Maps

A map is a hash table: keys to values, no promised order. The boring default is `make` before the first write, the **comma-ok** form on every lookup that might miss, and a slice of keys when you need a stable walk.

Mental model

`map[K]V` needs a comparable key (`string`, `int`, a struct of comparable fields — not a slice). The zero map is `nil`. **Read** from nil is fine (you get the zero `V`). **Write** to nil panics.

Lookup returns the value, or the zero value if the key is missing. That is why `prices["ghost"]` looks like `0`. Comma-ok tells the two apart: `cents, ok := prices["tea"]`.

`delete(m, k)` is a no-op if the key is absent or `m` is nil. `range` visits each key once per walk, in an order you must not depend on.

Worked examples

Case 1: `make`, write, read

Save as `menu_map.go`. One map of item to cents.

```
// menu_map.go
package main

import "fmt"

func main() {
    prices := make(map[string]int)
    prices["toast"] = 350
    prices["tea"] = 250
    fmt.Println(prices["toast"])
    fmt.Println("len", len(prices))
}
```

Run:

```
go run menu_map.go
```

Output:

```
350
len 2
```

A composite literal also allocates: `prices := map[string]int{"toast": 350, "tea": 250}`. Same type, already filled.

Case 2: Comma-ok

Save as `comma_ok.go`. Missing keys must not look like free tea.

```go
// comma_ok.go
package main

import "fmt"

func main() {
    prices := map[string]int{
        "toast": 350,
        "tea":   0, // comped
    }

    for _, name := range []string{"toast", "tea", "soup"} {
        cents, ok := prices[name]
        if !ok {
            fmt.Println(name, "not on the menu")
            continue
        }
        fmt.Println(name, cents)
    }
}
```

Run:

```
go run comma_ok.go
```

Output:

```
toast 350
tea 0
soup not on the menu
```

`tea` is on the menu at 0 cents. `soup` is not. Only `ok` distinguishes them.

Case 3: `delete` and `range`

Save as `delete_item.go`. After delete, the key is gone. Range prints the rest — order not guaranteed.

```
// delete_item.go
package main

import "fmt"

func main() {
    prices := map[string]int{
        "toast": 350,
        "tea":   250,
        "soup":  600,
    }
    delete(prices, "tea")
    delete(prices, "ghost")

    fmt.Println("len", len(prices))
    for item, cents := range prices {
        fmt.Println(item, cents)
    }
}
```

Run:

```
go run delete_item.go
```

Possible output:

```
len 2
soup 600
toast 350
```

`delete` of `"ghost"` did nothing. Do not write tests that assert the order of the two remaining lines.

Case 4: Equality is not deep

Save as `map_nil.go`. You may compare a map to `nil`. You may not compare two maps to each other.

```
// map_nil.go
package main

import "fmt"
```

```
func main() {
    var unset map[string]int
    empty := map[string]int{}
    fmt.Println("unset nil", unset == nil)
    fmt.Println("empty nil", empty == nil)
    fmt.Println("len unset", len(unset), "len empty", len(empty))
}
```

Run:

```
go run map_nil.go
```

Output:

```
unset nil true
empty nil false
len unset 0 len empty 0
```

`empty == unset` does not compile (`invalid operation: empty == unset (map can only be compared to nil)`). Walk both maps if you need deep equality, or use a helper in tests. A struct that contains a map is also not comparable.

Case 5: The tally pattern (zero-value advantage)

Save as `tally.go`. Because a missing key evaluates to the zero value of its value type (`0` for `int`), you do not need to check if a key exists before incrementing it.

```
// tally.go
package main

import "fmt"

func main() {
    orders := []string{"toast", "tea", "toast", "tea", "toast", "soup"}
    counts := make(map[string]int)

    for _, item := range orders {
        counts[item]++
    }

    fmt.Println("toast:", counts["toast"])
    fmt.Println("tea:", counts["tea"])
    fmt.Println("soup:", counts["soup"])
}
```

Run:

```
go run tally.go
```

Output:

```
toast: 3
tea: 2
soup: 1
```

On the first `"toast"`, `counts["toast"]` evaluates to `0`, gets incremented to `1`, and is written back. No `if _, ok := counts[item]; !ok` boilerplate is needed.

Case 6: Deterministic sorted iteration

Save as `sort_keys.go`. Map iteration order is deliberately randomized by the Go runtime to prevent programs from relying on hash order. When you print a bill or generate a report, collect keys into a slice and sort them.

```
// sort_keys.go
package main

import (
    "fmt"
    "slices"
)

func main() {
    prices := map[string]int{
        "toast": 350,
        "tea":   250,
        "soup":  600,
    }

    keys := make([]string, 0, len(prices))
    for k := range prices {
        keys = append(keys, k)
    }
    slices.Sort(keys)

    for _, k := range keys {
        fmt.Printf("%-6s %d\n", k, prices[k])
    }
}
```

Run:

```
go run sort_keys.go
```

Output:

```
soup   600
tea    250
toast  350
```

The output order is alphabetical every single run.

The trap

Trap 1: A nil map panics on write

`var` is not `make`. Save as `nil_map.go`:

```
// nil_map.go
package main

import "fmt"

func main() {
    var prices map[string]int
    fmt.Println("read missing", prices["tea"])
    prices["tea"] = 250
}
```

Run:

```
go run nil_map.go
```

Output:

```
read missing 0
panic: assignment to entry in nil map

goroutine 1 [running]:
main.main()
    nil_map.go:9 +0xa5
exit status 2
```

The read looked polite. The write killed the process. The fix is one line before any assignment: `prices = make(map[string]int)`, or declare with a literal. Returning a nil map from a function is fine **if callers only range and look up**. Document that, or return an empty `map[string]int{}` so writes succeed.

Trap 2: Direct mutation of struct fields in a map

If a map stores struct values directly, writing to a field of that struct (`m[k].Field = val`) does not compile:

```
cannot assign to struct field m["T1"].Table in map
```

Map elements are not addressable because map hash tables grow and reallocate buckets dynamically as entries are added. Storing an address to an internal map slot would create dangling pointers when the map resizes.

Save as `map_struct_fix.go` to see the two correct patterns:

```go
// map_struct_fix.go
package main

import "fmt"

type Ticket struct {
    ID    int
    Table int
}

func main() {
    // Pattern A: Reassign the modified value struct
    byID := map[string]Ticket{
        "T1": {ID: 1, Table: 4},
    }
    t := byID["T1"]
    t.Table = 12
    byID["T1"] = t
    fmt.Println("Pattern A table:", byID["T1"].Table)

    // Pattern B: Use pointers as map values
    byPtr := map[string]*Ticket{
        "T2": {ID: 2, Table: 8},
    }
    byPtr["T2"].Table = 15
    fmt.Println("Pattern B table:", byPtr["T2"].Table)
}
```

Run:

```
go run map_struct_fix.go
```

Output:

```
Pattern A table: 12
Pattern B table: 15
```

If the struct is small and values are rarely updated in place, reassign the struct (Pattern A). If the struct is frequently updated or passed to methods, use pointers as values (Pattern B).

The boring rule

- `make` (or a literal) before write.
- Comma-ok on lookups that can miss. Do not treat `0` or `""` as "absent."
- Rely on zero values for counters (`counts[k]++`) and append accumulators (`groups[k] = append(groups[k], item)`).
- `delete` is safe on missing keys and on nil maps.
- Do not range a map for order. Extract keys and sort them with `slices.Sort`.
- Do not compare maps except to `nil`.
- When mutating structs stored in a map, store pointers (`map[K]*V`) or copy, mutate, and reassign the value.
- Keys must be comparable. If you want a slice as a key, you wanted a `string` (join it) or a different structure.

Try this

1. In `comma_ok.go`, add `"soup": 600` and confirm the miss path no longer runs.
2. In `tally.go`, add a group accumulator `grouped := make(map[string][]int)` and group ticket IDs under item names using `append`.
3. In `sort_keys.go`, sort keys in reverse order using `slices.Reverse(keys)`.
4. In `map_struct_fix.go`, write `byID["T1"].Table = 99` without copying first. Observe the compiler rejection and explain why map elements are not addressable.

Structs and Data Modeling

Structs and Data Modeling

A struct is a named bag of fields. The boring default is a few structs for the desk (ticket, order, shift), composite literals that name their fields, and **embedding** only when promotion is the point — not as a substitute for inheritance, which Go does not have.

Mental model

`type Ticket struct { ... }` defines a type. Values are copied on assignment. A field name that starts with an upper-case letter is **exported** (visible from another package). Lower-case is package-private.

A **composite literal** `Ticket{ID: 7, Table: 12}` sets the named fields; the rest are zero. Embedding writes another type as a field without a name. Methods and fields of the inner type are **promoted** to the outer type. The outer type is still not "a kind of" the inner type.

A **tag** is a string on a field (`json:"id"`). Encoding packages read tags. Your code usually ignores them.

Worked examples

Case 1: Ticket and order literals

Save as `desk_structs.go`. Named fields, then a print.

```
// desk_structs.go
package main

import "fmt"

type Ticket struct {
    ID    int
    Table int
}

type Order struct {
    Ticket Ticket
    Item   string
    Cents  int
}
```

```
func main() {
    t := Ticket{ID: 7, Table: 12}
    o := Order{Ticket: t, Item: "toast", Cents: 350}
    fmt.Printf("ticket %d table %d\n", o.Ticket.ID, o.Ticket.Table)
    fmt.Printf("%s %d cents\n", o.Item, o.Cents)
}
```

Run:

```
go run desk_structs.go
```

Output:

```
ticket 7 table 12
toast 350 cents
```

`Order{Ticket: t, ...}` is clearer than positional `Order{t, "toast", 350}`. Use names.

Case 2: Exported fields and a JSON tag

Save as `ticket_json.go`. `note` is lower-case, so `encoding/json` skips it. The tags rename the exported fields in the JSON.

```
// ticket_json.go
package main

import (
    "encoding/json"
    "fmt"
)

type Ticket struct {
    ID    int    `json:"id"`
    Table int    `json:"table"`
    note  string `json:"note"`
}

func main() {
    t := Ticket{ID: 7, Table: 12, note: "window"}
    b, err := json.Marshal(t)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(string(b))
}
```

Run:

```
go run ticket_json.go
```

Output:

```
{"id":7,"table":12}
```

`note` is in the struct and gone from the JSON. Other packages cannot write `t.note` either. Export the field (`Note`) when callers and encoders should see it.

Case 3: Embedding promotes fields and methods

Save as `embed_table.go`. `Order` embeds `Table`. `o.Number` is `o.Table.Number`. `o.Label()` is `o.Table.Label()`.

```
// embed_table.go
package main

import "fmt"

type Table struct {
    Number int
    Seats  int
}

func (t Table) Label() string {
    return fmt.Sprintf("table %d", t.Number)
}

type Order struct {
    Table
    Item  string
    Cents int
}

func main() {
    o := Order{
        Table: Table{Number: 4, Seats: 2},
        Item:  "toast",
        Cents: 350,
    }
    fmt.Println(o.Number, o.Seats, o.Label())
    fmt.Println(o.Item, o.Cents)
}
```

Run:

```
go run embed_table.go
```

Output:

```
4 2 table 4
toast 350
```

Embedding is a shorter path to the inner fields. It is still a field. `o.Table` exists.

Case 4: Embedding is not inheritance

Save as `not_a_table.go`. A function that wants a `Table` will not take an `Order`. You pass `o.Table`.

```
// not_a_table.go
package main

import "fmt"

type Table struct {
    Number int
    Seats  int
}

type Order struct {
    Table
    Item string
}

func freeSeats(t Table) int {
    return t.Seats
}

func main() {
    o := Order{Table: Table{Number: 4, Seats: 2}, Item: "tea"}
    fmt.Println(freeSeats(o.Table))
}
```

Run:

```
go run not_a_table.go
```

Output:

```
2
```

`freeSeats(o)` does not compile: `cannot use o (variable of struct type Order) as Table value`. There is no subclass. There is a field.

The trap

A struct value is a copy. A method or function that takes `Ticket` (not `*Ticket`) cannot update the caller's fields. Save as `copy_update.go`:

```
// copy_update.go
package main

import "fmt"

type Ticket struct {
    ID    int
    Table int
}

func (t Ticket) move(table int) {
    t.Table = table
}

func (t *Ticket) movePtr(table int) {
    t.Table = table
}

func main() {
    t := Ticket{ID: 7, Table: 12}
    t.move(4)
    fmt.Println("after value", t.Table)
    t.movePtr(4)
    fmt.Println("after pointer", t.Table)
}
```

Run:

```
go run copy_update.go
```

Output:

```
after value 12
after pointer 4
```

`move` changed a copy. The desk still thinks table 12. Use a pointer receiver when the method's job is to change the struct. Use a value receiver when the method only reads (see methods later). Until then: if you mutate, pass `*Ticket`.

The boring rule

- Name fields in literals. Leave unused fields zero.
- Export fields that other packages (and JSON) must see. Keep the rest lower-case.
- Tags are for encoders. Add them when you encode, not “just in case.”
- Embed when the inner type *is* a part of the outer value and promotion helps. Otherwise use a named field (`Ticket Ticket`).
- Do not pretend embedding is inheritance. Pass `o.Table`, not `o`, into functions that want `Table`.
- Mutating methods take a pointer. Assignment copies the struct.

Try this

1. In `desk_structs.go`, add a `Shift` struct (`Name string`, `Tables int`) and print one literal.
2. In `ticket_json.go`, rename `note` to `Note` and run. The JSON should include `"Note":"window"` unless you add a tag.
3. In `embed_table.go`, print `o.Table.Number` next to `o.Number`. They are the same slot.
4. In `not_a_table.go`, uncomment a call `freeSeats(o)` in your head, then try it and read the compile error.

Promoted Field Literals

Promoted Field Literals

Embedding already lets you *read* `o.Number` when `Order` embeds `Table`. Go **1.27** closes the other half of that story for construction: a key in a composite literal may be any valid **field selector** for the type, so you can write `Number: 4` on the outer literal when that name promotes unambiguously. Nested `Table: Table{...}` is still legal and often clearer. Prefer the promoted key when the desk value is one shape and the inner type is just plumbing.

Mental model

- Embedding still creates a real field. Promotion is a selector path, not inheritance.
- Before 1.27, a composite-literal key had to be a **top-level** field name of that struct type. Setting an embedded field meant nesting: `Table: Table{Number: 4}`.
- On 1.27+, a key may be a promoted field name (`Number`) or another valid field selector the type accepts. The compiler fills the same slot `o.Number` would read.
- If two embeddings promote the **same** name, the promoted key is ambiguous. Nest (or name a field) so the path is unique.
- Reading and writing stay aligned: if `o.Street` compiles, `Street:` in an `Order{...}` literal is the boring twin — when unambiguous.

Worked examples

Case 1: Address fields on a ticket

Save as `ticket_address.go`. `Ticket` embeds `Address`. On Go 1.27 you can set `Street` and `City` as keys on the outer literal.

```
// ticket_address.go
package main

import "fmt"

type Address struct {
    Street string
    City   string
}

type Ticket struct {
    ID int
```

```
    Address
}

func main() {
    t := Ticket{
        ID:     7,
        Street: "12 Desk Lane",
        City:   "Portland",
    }
    fmt.Printf("ticket %d\n", t.ID)
    fmt.Printf("%s, %s\n", t.Street, t.City)
    fmt.Printf("via embed: %s\n", t.Address.Street)
}
```

Run:

```
go run ticket_address.go
```

Output:

```
ticket 7
12 Desk Lane, Portland
via embed: 12 Desk Lane
```

`t.Street` and `t.Address.Street` are the same slot. The literal just stopped making you nest `Address: Address{...}` for the common case.

Case 2: Nested form still works

Save as `ticket_address_nested.go`. Same type, old construction. Use this when you want the inner type name visible, or when you are matching older code.

```
// ticket_address_nested.go
package main

import "fmt"

type Address struct {
    Street string
    City   string
}

type Ticket struct {
    ID int
    Address
}
```

```
func main() {
    t := Ticket{
        ID: 7,
        Address: Address{
            Street: "12 Desk Lane",
            City:   "Portland",
        },
    }
    fmt.Printf("%s / %s\n", t.Street, t.Address.City)
}
```

Run:

```
go run ticket_address_nested.go
```

Output:

```
12 Desk Lane / Portland
```

Both forms build the same value. Pick the one a tired reader will parse faster.

Case 3: Equality of both constructions

Save as `same_order.go`. Build one `Order` with promoted keys and one with a nested `Table` literal. Compare field by field.

```
// same_order.go
package main

import "fmt"

type Table struct {
    Number int
    Seats  int
}

type Order struct {
    Table
    Item  string
    Cents int
}

func main() {
    promoted := Order{
        Number: 4,
        Seats:  2,
```

```
        Item:   "toast",
        Cents:  350,
    }
    nested := Order{
        Table: Table{Number: 4, Seats: 2},
        Item:  "toast",
        Cents: 350,
    }
    fmt.Printf("equal? %v\n", promoted == nested)
    fmt.Printf("promoted table=%d seats=%d\n", promoted.Number, promoted.Seats)
    fmt.Printf("nested   table=%d seats=%d\n", nested.Table.Number, nested.Table.Seats)
}
```

Run:

```
go run same_order.go
```

Output:

```
equal? true
promoted table=4 seats=2
nested   table=4 seats=2
```

Structs compare field by field. Promotion only changes how you *spell* the initialization.

Case 4: Ambiguous promotion — nest on purpose

Save as `ambiguous_ids.go`. `Shift` and `Badge` both have `ID`. A promoted `ID:` key on `Assignment` would be ambiguous, so the program uses nested literals (the form that always compiles).

```
// ambiguous_ids.go
package main

import "fmt"

type Shift struct {
    ID   int
    Day  string
}

type Badge struct {
    ID     int
    Holder string
}

type Assignment struct {
```

```
    Shift
    Badge
    Station string
}

func main() {
    a := Assignment{
        Shift:   Shift{ID: 3, Day: "Monday"},
        Badge:   Badge{ID: 9001, Holder: "Sam"},
        Station: "front desk",
    }
    fmt.Printf("shift=%d badge=%d station=%s\n", a.Shift.ID, a.Badge.ID, a.Station)
    fmt.Printf("holder=%s day=%s\n", a.Holder, a.Day)
}
```

Run:

```
go run ambiguous_ids.go
```

Output:

```
shift=3 badge=9001 station=front desk
holder=Sam day=Monday
```

`a.Holder` and `a.Day` promote cleanly. `a.ID` does **not** — two `ID` fields compete. Writing `ID: 3` in the `Assignment{...}` literal fails for the same reason. Nest (or stop embedding one of them) when names collide.

The trap

Save as `promote_not_subtype.go`. Promoted keys make construction look flatter. They do not make `Order` a `Table`. Functions that want `Table` still need `o.Table`.

```
// promote_not_subtype.go
package main

import "fmt"

type Table struct {
    Number int
    Seats  int
}

type Order struct {
    Table
    Item string
```

```
}

func freeSeats(t Table) int {
    return t.Seats
}

func main() {
    o := Order{
        Number: 4,
        Seats:  2,
        Item:   "tea",
    }
    fmt.Println(freeSeats(o.Table))
    fmt.Printf("item=%s number=%d\n", o.Item, o.Number)
}
```

Run:

```
go run promote_not_subtype.go
```

Output:

```
2
item=tea number=4
```

`freeSeats(o)` still does not compile. Flatter literals are sugar over the same embedding rules you already learned. If a review wants "is-a Table," use an interface or pass `o.Table` — do not reach for embedding tricks.

The boring rule

- On Go **1.27+**, use promoted field keys when the embedded name is unambiguous and the flatter literal reads better.
- Keep the nested `Inner: Inner{...}` form when you want the inner type name on the page, when porting older snippets, or when two embeddings share a field name.
- Embedding is still for "this value *has* that part," not for inheritance and not merely to shorten literals.
- If `outer.Field` is ambiguous at use sites, it is ambiguous in literals too — nest or rename.
- Mutating methods still need pointer receivers; promotion does not change copying.

Try this

1. In `ticket_address.go`, add `Zip string` to `Address` and set it with a promoted key. Print `t.Zip` and `t.Address.Zip`.

2. In `same_order.go`, change only the nested form's `Seats` to `4`. Confirm `equal?` becomes `false`.
3. In `ambiguous_ids.go`, try adding `ID: 3` to the `Assignment{...}` literal and read the compiler error. Then remove it.
4. Rewrite `Case 1` with a *named* field `Addr Address` instead of embedding. Which literal keys still work? Why?

Functions and Methods

Functions in Depth

Functions in Depth

A Go function is a named piece of work with a type you can read in one line: parameters in, results out. The boring default is a short function, explicit results, and a caller that looks at every error. Named results, `defer`, and `...` exist because a few jobs need them — not because every signature should look clever.

Mental model

A **signature** is the function's contract: name, parameter types, result types. Callers depend on that line, not on anything you do inside.

Values are copied into parameters. A `[]int` parameter is a copy of the slice header (pointer, length, capacity), not a deep copy of the elements — that sharing is a later chapter. For now: integers, structs, and strings you receive are yours to use; writing to a local parameter does not write back to the caller.

Results can be more than one. The usual pair is `(T, error)`. The **blank identifier** `_` throws a result away. `defer` schedules a call to run when the surrounding function returns, in last-in first-out order. A **variadic** parameter (`...T`) is a `[]T` built from the arguments.

Worked examples

Case 1: A signature you can read aloud

Save as `total.go`. One parameter, one result, no hidden state.

```
// total.go
package main

import "fmt"

func total(prices []int) int {
    sum := 0
    for _, p := range prices {
        sum += p
    }
    return sum
}
```

```
func main() {
    fmt.Println(total([]int{12, 8, 20}))
}
```

Run:

```
go run total.go
```

Output:

```
40
```

`prices` is the parameter name. `[]int` is its type. `int` after the list is the result type. Names in the signature are for humans; the type is what the compiler checks.

Case 2: Multiple results, error last

Save as `split_ticket.go`. The desk prints tickets as `id: note`. The function returns the pieces or an error. Zeros on the failure path are intentional: the caller must check `err` before using the other results.

```
// split_ticket.go
package main

import (
    "fmt"
    "strings"
)

func splitTicket(label string) (int, string, error) {
    idStr, rest, ok := strings.Cut(label, ":")
    if !ok {
        return 0, "", fmt.Errorf("ticket %q: missing colon", label)
    }
    var id int
    _, err := fmt.Sscanf(idStr, "%d", &id)
    if err != nil {
        return 0, "", fmt.Errorf("ticket %q: bad id", label)
    }
    return id, strings.TrimSpace(rest), nil
}

func main() {
    id, note, err := splitTicket("41: extra napkins")
    if err != nil {
        fmt.Println("err:", err)
```

```
        return
    }
    fmt.Printf("id=%d note=%q\n", id, note)
}
```

Run:

```
go run split_ticket.go
```

Output:

```
id=41 note="extra napkins"
```

The `_` on the `Sscanf` line discards the count of scanned items. That is a fair use of blank: you already have `err`. Do not blank-out `err` itself.

Case 3: Named results, used sparingly

Save as `share.go`. Named result parameters are extra locals, pre-zeroed, that a bare `return` will send back. They help when the names *are* the documentation (`each`, `leftover`) and the function is short. They hurt when you mix them with `:=` and `defer`.

```
// share.go
package main

import "fmt"

func share(cents int, n int) (each int, leftover int) {
    if n <= 0 {
        return 0, cents
    }
    each = cents / n
    leftover = cents % n
    return
}

func main() {
    each, leftover := share(1000, 3)
    fmt.Printf("each=%d leftover=%d\n", each, leftover)
}
```

Run:

```
go run share.go
```

Output:

```
each=333 leftover=1
```

The early `return 0, cents` is still explicit. Naked `return` is the part to keep rare. Prefer `return each, leftover` if anyone on the desk might miss what goes back.

Case 4: `defer` runs on the way out

Save as `close_shift.go`. Open the till, count, then close — even if you add more returns later. `defer` is for cleanup that must not be skipped.

```
// close_shift.go
package main

import "fmt"

func closeShift(name string) {
    fmt.Println("open", name)
    defer fmt.Println("close", name)
    fmt.Println("count till")
}

func main() {
    closeShift("Amina")
}
```

Run:

```
go run close_shift.go
```

Output:

```
open Amina
count till
close Amina
```

Deferred calls run last-in first-out. Two defers stack like plates:

```
// defer_order.go
package main

import "fmt"

func main() {
    defer fmt.Println("unlock drawer")
    defer fmt.Println("write log")
    fmt.Println("count cash")
```

```
}
```

Run:

```
go run defer_order.go
```

Output:

```
count cash
write log
unlock drawer
```

Arguments to a deferred call are evaluated when `defer` runs, not when the function returns. Keep deferred calls simple: `f.Close()`, `unlock()`, `cancel()`.

Case 5: Variadic arguments

Save as `sum.go`. `cents ...int` means zero or more `int` arguments. Inside the function it is a `[]int`. Pass an existing slice with `order...`.

```
// sum.go
package main

import "fmt"

func sum(cents ...int) int {
    total := 0
    for _, c := range cents {
        total += c
    }
    return total
}

func main() {
    fmt.Println(sum())
    fmt.Println(sum(250, 400))
    order := []int{120, 80, 50}
    fmt.Println(sum(order...))
}
```

Run:

```
go run sum.go
```

Output:

```
0
650
250
```

A function may have at most one variadic parameter, and it must be last. `fmt.Println` is the standard-library version of the same idea.

The trap

Named results plus `:=` will happily create a *new* `err` that dies at the end of the `if`. The named `err` stays `nil`. The function "succeeds."

Save as `named_shadow.go`:

```
// named_shadow.go
package main

import "fmt"

func priceOf(item string) (cents int, err error) {
    if item == "" {
        err := fmt.Errorf("empty item")
        _ = err
    }
    return
}

func main() {
    cents, err := priceOf("")
    fmt.Printf("cents=%d err=%v\n", cents, err)
}
```

Run:

```
go run named_shadow.go
```

Output:

```
cents=0 err=<nil>
```

The inner `err :=` is a different variable. The `_ = err` only exists so the program compiles. The caller sees no error.

The boring fix is to stop naming results and return values you can see:

```go
// named_shadow_fix.go
package main

import "fmt"

func priceOf(item string) (int, error) {
    if item == "" {
        return 0, fmt.Errorf("empty item")
    }
    return 250, nil
}

func main() {
    cents, err := priceOf("")
    fmt.Printf("cents=%d err=%v\n", cents, err)
}
```

Run:

```
go run named_shadow_fix.go
```

Output:

```
cents=0 err=empty item
```

If you keep named results, assign with `=` (`err = fmt.Errorf(...)`), never `:=`, and write `return cents, err`.

The boring rule

- Put `error` last. Check it before you use the other results.
- Name parameters for the reader. Keep the function small enough that named *results* are optional.
- Prefer `return v, err` over a naked `return`.
- Use `defer` for cleanup (close, unlock, cancel), not for clever control flow.
- Use `...T` when the natural call is a list of values. Use a slice parameter when the caller already has a slice and always will.
- `_` is for results you have already accounted for. Never `_ = err` at a desk that cares about money.

Try this

1. Change `split_ticket.go` so `main` also calls `splitTicket("no-colon")` and prints the error. Do not crash.

2. In `close_shift.go`, add a second `defer` that prints `log shift done`. Confirm it runs *before* `close` (last-in first-out).
3. In `sum.go`, add a parameter `label string` in front of `cents ...int` and print the label with the total. Variadic still has to be last.
4. In `named_shadow.go`, replace `err :=` with `err =` and drop `_ = err`. Confirm the caller sees `empty item`.

Functions as Values

Functions as Values

A function value is data. You can pass it, store it, and return it. The boring default is a named function or a tiny literal at the call site — the same idea as an HTTP handler, without a server.

Mental model

Every function has a **type**: the parameter types and the result types. `func(int) bool` is a type, the way `int` is a type. A variable of that type holds a function, or it is `nil`.

A **closure** is a function literal that uses variables from the surrounding function. Those variables are shared, not copied: later assignments are visible when the closure runs. That is the power and the leak.

Since Go 1.22 (and in this book's Go 1.27), each iteration of a `for` loop has its **own** per-iteration variables. Closures that capture `name` inside `for _, name := range names` see that iteration's `name`, not the last one. You still have to think when you close over a variable *outside* the loop.

Worked examples

Case 1: A function type as a parameter

Save as `keep.go`. `keep` does not know what "interesting" means. The caller passes a predicate.

```
// keep.go
package main

import "fmt"

func keep(orders []int, ok func(int) bool) []int {
    var out []int
    for _, o := range orders {
        if ok(o) {
            out = append(out, o)
        }
    }
    return out
}

func paid(cents int) bool { return cents > 0 }
```

```
func main() {
    orders := []int{0, 450, 120, 0, 90}
    fmt.Println(keep(orders, paid))
}
```

Run:

```
go run keep.go
```

Output:

```
[450 120 90]
```

`paid` matches `func(int) bool`, so it is a valid argument. You can also pass a literal: `keep(orders, func(c int) bool { return c >= 100 })`.

Case 2: A named function type, HTTP-style

Save as `route.go`. An HTTP server is this pattern with extra types: a path in, a function that handles it. Keep the same shape at the desk — no port, no `net/http` yet.

```
// route.go
package main

import "fmt"

type Handler func(table int, note string)

func route(table int, note string, h Handler) {
    h(table, note)
}

func kitchen(table int, note string) {
    fmt.Printf("kitchen table %d: %s\n", table, note)
}

func bar(table int, note string) {
    fmt.Printf("bar table %d: %s\n", table, note)
}

func main() {
    route(4, "two coffees", bar)
    route(11, "soup", kitchen)
    route(7, "hold the onions", func(table int, note string) {
        fmt.Printf("note on table %d: %s\n", table, note)
    })
```

```
}
```

Run:

```
go run route.go
```

Output:

```
bar table 4: two coffees
kitchen table 11: soup
note on table 7: hold the onions
```

`type Handler func(...)` is a name for a function type. It is not a class. `kitchen` and `bar` are ordinary functions that happen to match. The last call is a one-off literal — fine when it is three lines.

Case 3: A closure that keeps state

Save as `counter.go`. `makeCounter` returns a function. The local `n` lives as long as that function does.

```
// counter.go
package main

import "fmt"

func makeCounter(start int) func() int {
    n := start
    return func() int {
        n++
        return n
    }
}

func main() {
    next := makeCounter(40)
    fmt.Println(next())
    fmt.Println(next())
    fmt.Println(next())
}
```

Run:

```
go run counter.go
```

Output:

```
41
42
43
```

Two counters are two `n`s. `makeCounter(40)` and `makeCounter(40)` again do not share memory. That is the usual way to hide a little state without a struct — until the state grows, at which point you want a struct.

Case 4: Loop variables, modern Go

Save as `greet_loop.go`. Each closure prints the name from *its* iteration. In Go 1.21 and earlier this program would print `Chen` three times. It does not, on Go 1.27.

```
// greet_loop.go
package main

import "fmt"

func main() {
    names := []string{"Amina", "Bo", "Chen"}
    var greet []func()
    for _, name := range names {
        greet = append(greet, func() {
            fmt.Println("hello", name)
        })
    }
    for _, g := range greet {
        g()
    }
}
```

Run:

```
go run greet_loop.go
```

Output:

```
hello Amina
hello Bo
hello Chen
```

If you must target an old compiler, the old workaround was `name := name` inside the loop. You do not need that here. You still need to be careful with variables declared *outside* the loop — next section.

The trap

A closure captures the variable, not a snapshot of its value. Assign after you build the function, and the function sees the new value.

Save as `shift_greet.go`:

```
// shift_greet.go
package main

import "fmt"

func main() {
    shift := "Amina"
    greet := func() {
        fmt.Println("hello", shift)
    }
    greet()
    shift = "Bo"
    greet()
}
```

Run:

```
go run shift_greet.go
```

Output:

```
hello Amina
hello Bo
```

That is correct Go and a surprise if you expected a copy. The fix is to pass the value in, so the closure's parameter is its own:

```
// shift_greet_fix.go
package main

import "fmt"

func main() {
    shift := "Amina"
    greet := func(name string) {
        fmt.Println("hello", name)
    }
    greet(shift)
    shift = "Bo"
    greet(shift)
}
```

Run:

```
go run shift_greet_fix.go
```

Output:

```
hello Amina
hello Bo
```

Same output, clearer contract: `greet` takes a name. If you truly want a snapshot, copy into a new variable before the literal (`name := shift`) and close over `name`.

The boring rule

- Give function types a name when they appear more than once (`Handler`, `Predicate`). Leave them inline when they appear once.
- Prefer a named function (`paid`, `kitchen`) over an anonymous blob that hides in a call.
- Closures are for small captured state. When you have three captured variables, use a struct and a method.
- On Go 1.22+, loop vars are per-iteration. Still pass values into callbacks that outlive the function if the intent is "this value," not "whatever that variable is later."
- `nil` function values panic when called. Check, or do not store `nil`.

Try this

1. In `keep.go`, pass a literal that keeps orders of at least 100 cents. Do not add a new named function.
2. In `route.go`, add a `Handler` that prints `void table N` and route table `3` with an empty note.
3. In `counter.go`, make two counters from `makeCounter(0)` and call each twice. Confirm they do not share `n`.
4. In `shift_greet.go`, snapshot with `name := shift` before the literal and close over `name`. Change `shift` after. What prints?

Methods and Receivers

Methods and Receivers

A method is a function with one extra parameter written in front: the **receiver**. The boring default is a value receiver when the method only reads, and a pointer receiver when it writes. Pick one style per type and keep it.

Mental model

```
func (t Ticket) Label() string    // value receiver: works on a copy of t
func (t *Ticket) Seat(n int)      // pointer receiver: t points at the original
```

The receiver is still pass-by-value. A value receiver copies the struct. A pointer receiver copies the pointer (the address), so the method can change fields.

Go will take the address of an **addressable** variable for you: if `bump` has a `*Table` receiver, `t.bump()` means `(&t).bump()`. That convenience does **not** apply when you put the value in an interface. Interfaces use **method sets**:

- Type `T` has the methods with receiver `T`.
- Type `*T` has the methods with receiver `T` and the methods with receiver `*T`.

So a value of type `Drawer` does not satisfy an interface that needs `Close()` if `Close` is on `*Drawer`.

Worked examples

Case 1: A value receiver, called two ways

Save as `ticket_label.go`. `Label` only reads. A copy is fine. Calling it on a pointer still works: Go copies the pointed-to struct into the value receiver.

```
// ticket_label.go
package main

import "fmt"

type Ticket struct {
    ID    int
    Table int
}
```

```
func (t Ticket) Label() string {
    return fmt.Sprintf("ticket %d → table %d", t.ID, t.Table)
}

func main() {
    t := Ticket{ID: 7, Table: 12}
    fmt.Println(t.Label())
    p := &t
    fmt.Println(p.Label())
}
```

Run:

```
go run ticket_label.go
```

Output:

```
ticket 7 → table 12
ticket 7 → table 12
```

There is no `this`. The receiver is an argument with a slightly special place in the syntax.

Case 2: Mutation needs a pointer

Save as `seats.go`. The value-receiver version increments a copy. The pointer-receiver version increments the table you care about. `t.bump()` is allowed because `t` is a variable (addressable).

```
// seats.go
package main

import "fmt"

type Table struct {
    Number int
    Seats  int
}

func (t Table) bumpValue() {
    t.Seats++
}

func (t *Table) bump() {
    t.Seats++
}

func main() {
```

```
    t := Table{Number: 4, Seats: 2}
    t.bumpValue()
    fmt.Println("after value:", t.Seats)
    t.bump()
    fmt.Println("after pointer:", t.Seats)
}
```

Run:

```
go run seats.go
```

Output:

```
after value: 2
after pointer: 3
```

If a method must change the receiver, or must share one struct with other methods that change it, use `*T`. Mixing a pointer `Close` with a value `Label` on the same type is allowed; mixing them at random is how method-set errors show up on Friday.

Case 3: Method sets and interfaces

Save as `drawer.go`. `Closer` needs `Close()`. `Close` has a pointer receiver, so you pass `*Drawer`.

```
// drawer.go
package main

import "fmt"

type Drawer struct {
    Name string
    Open bool
}

func (d *Drawer) Close() {
    d.Open = false
}

type Closer interface {
    Close()
}

func shut(c Closer) {
    c.Close()
}
```

```
func main() {
    d := Drawer{Name: "till", Open: true}
    shut(&d)
    fmt.Println(d.Open)
}
```

Run:

```
go run drawer.go
```

Output:

```
false
```

`&d` has type `*Drawer`. `*Drawer` includes `Close`. The field flips on the original `d`.

Case 4: Defensive methods on nil receivers

In many object-oriented languages, invoking a method on a `nil` or `null` reference immediately causes a crash. In Go, calling a method on a `nil` pointer receiver is valid and executes the function body. The receiver is simply passed as a `nil` argument. The method can check `t == nil` and return a sensible default.

Save as `nil_receiver.go`:

```
// nil_receiver.go
package main

import "fmt"

type Table struct {
    Number int
}

func (t *Table) Description() string {
    if t == nil {
        return "no table assigned"
    }
    return fmt.Sprintf("table %d", t.Number)
}

func main() {
    var t1 *Table
    t2 := &Table{Number: 12}

    fmt.Println(t1.Description())
    fmt.Println(t2.Description())
```

```
}
```

Run:

```
go run nil_receiver.go
```

Output:

```
no table assigned
table 12
```

This pattern is widely used in standard library types (like `*bytes.Buffer.String()` or error types) to avoid panics on uninitialized pointers.

Case 5: Method values and method expressions

A method can be extracted as a first-class function value:

1. **Method value**: `t.Summary` binds the method to the specific instance `t`. It produces a function with signature `func() string`.
2. **Method expression**: `Ticket.Summary` yields an unbound function with signature `func(Ticket) string`, where the receiver is passed explicitly as the first argument.

Save as `method_values.go`:

```
// method_values.go
package main

import "fmt"

type Ticket struct {
    ID    int
    Table int
}

func (t Ticket) Summary() string {
    return fmt.Sprintf("ticket %d (table %d)", t.ID, t.Table)
}

func main() {
    t := Ticket{ID: 41, Table: 8}

    // Method value: bound to instance t
    summaryFunc := t.Summary
    fmt.Println("method value:", summaryFunc())

    // Method expression: unbound, takes instance as first argument
```

```
    exprFunc := Ticket.Summary
    fmt.Println("method expression:", exprFunc(t))
}
```

Run:

```
go run method_values.go
```

Output:

```
method value: ticket 41 (table 8)
method expression: ticket 41 (table 8)
```

Method values are useful for passing callbacks (e.g. `http.HandlerFunc(srv.handleOrder)`).

The trap

The same `shut` with a value does not compile. `t.Close()` would have been rewritten to `(&t).Close()` if `t` were a variable in the caller. Once the value is in the interface, that rewrite is gone.

Save as `drawer_value.go`:

```
// drawer_value.go
package main

import "fmt"

type Drawer struct {
    Name string
    Open bool
}

func (d *Drawer) Close() {
    d.Open = false
}

type Closer interface {
    Close()
}

func shut(c Closer) {
    c.Close()
}

func main() {
```

```
    d := Drawer{Name: "till", Open: true}
    shut(d)
    fmt.Println(d.Open)
}
```

Run:

```
go run drawer_value.go
```

Output:

```
# command-line-arguments
./drawer_value.go:25:7: cannot use d (variable of struct type Drawer) as Closer value in argume
```

If you are inside a module, the first line is the module path instead of `command-line-arguments`. The error text is the point.

The fix is the previous program: pass `&d`, or give `Close` a value receiver if it does not mutate (it does, so do not). When a type has any pointer-receiver methods, treat `*T` as the type you store in interfaces.

The boring rule

- Value receiver: small struct, method only reads.
- Pointer receiver: method writes fields, or the struct is the identity you want to share.
- Do not mix receiver kinds on one type without a reason. If one method needs `*T`, most of the others can too.
- Pointer receivers can handle `nil` gracefully: check `if t == nil` before accessing fields.
- Use method values (`instance.Method`) when passing a callback to event loops or HTTP handlers.
- For interfaces, remember the method set. `*T` is the safe choice when any method has a pointer receiver.
- A method is still a function. `Ticket.Label(t)` works; `t.Label()` is the same call with nicer syntax.

Try this

1. In `ticket_label.go`, change `Label` to a pointer receiver. Confirm both `t.Label()` and `p.Label()` still run.
2. In `nil_receiver.go`, remove the `if t == nil` check and run `t1.Description()`. Observe the nil pointer dereference panic.
3. In `method_values.go`, pass `summaryFunc` into a helper function `func printReport(f func() string)` and verify it executes without needing `t`.
4. In `seats.go`, try `Table{Number: 1, Seats: 2}.bump()` — a method call on a temporary. Read the compiler error (the value is not addressable).

5. In `drawer.go`, add `func (d Drawer) NameTag() string` that returns `d.Name`. Call it on `d` and on `&d`.

Functional Options

Functional Options

A constructor with three or more optional settings is a maintenance problem waiting to happen. The functional options pattern solves it: an **option** is a function that writes one setting into a private config struct. The constructor accepts `...Option`. Callers pick what they need; unset fields stay at their zero value. Adding a new option never touches existing call sites.

Mental model

```
type config struct {
    name       string
    maxTickets int
}

type Option func(*config)

func NewDesk(opts ...Option) *Desk {
    cfg := &config{}        // all fields at zero
    for _, o := range opts {
        o(cfg)              // each option writes one knob
    }
    return &Desk{cfg: cfg}
}
```

`Option` is just a function type. `WithName("cashier")` returns a closure that sets `cfg.name`. `NewDesk()` calls every closure in the order the caller passed them. The signature of `NewDesk` never changes — new knobs are new option functions, nothing more.

Worked examples

Case 1: Basic options — zero, one, and two

Save as `desk_basic.go`. `NewDesk` accepts any number of options. Calling it with none gives sensible zero-value defaults. Calling it with options overrides exactly the fields the caller cares about.

```
// desk_basic.go
package main
```

```
import "fmt"

type config struct {
    name       string
    maxTickets int
}

type Option func(*config)

func WithName(s string) Option {
    return func(cfg *config) {
        cfg.name = s
    }
}

func WithMaxTickets(n int) Option {
    return func(cfg *config) {
        cfg.maxTickets = n
    }
}

type Desk struct {
    cfg config
}

func NewDesk(opts ...Option) *Desk {
    cfg := config{
        name:       "default",
        maxTickets: 100,
    }
    for _, o := range opts {
        o(&cfg)
    }
    return &Desk{cfg: cfg}
}

func (d *Desk) String() string {
    return fmt.Sprintf("Desk{name:%q maxTickets:%d}", d.cfg.name, d.cfg.maxTickets)
}

func main() {
    // zero options - built-in defaults apply
    d0 := NewDesk()
    fmt.Println(d0)

    // one option - only name changes
    d1 := NewDesk(WithName("reception"))
```

```
    fmt.Println(d1)

    // two options - both fields overridden
    d2 := NewDesk(WithName("cashier"), WithMaxTickets(50))
    fmt.Println(d2)
}
```

Run:

```
go run desk_basic.go
```

Output:

```
Desk{name:"default" maxTickets:100}
Desk{name:"reception" maxTickets:100}
Desk{name:"cashier" maxTickets:50}
```

Each call site is independent. Neither `d0` nor `d1` needs to know that `maxTickets` exists.

Case 2: Options that validate

Sometimes an option must reject bad input. Change `Option` to `func(*config) error`. The constructor collects every error with `errors.Join` and returns one combined error. The caller still passes a clean list of option calls; the error surfaces at `NewDesk`, not buried inside a setter.

Save as `desk_validated.go`:

```
// desk_validated.go
package main

import (
    "errors"
    "fmt"
    "log/slog"
    "os"
)

type config struct {
    name       string
    maxTickets int
    queueDepth int
}

type Option func(*config) error

func WithName(s string) Option {
    return func(cfg *config) error {
```

```
        if s == "" {
            return errors.New("desk name must not be empty")
        }
        cfg.name = s
        return nil
    }
}

func WithMaxTickets(n int) Option {
    return func(cfg *config) error {
        if n <= 0 {
            return fmt.Errorf("maxTickets must be positive, got %d", n)
        }
        cfg.maxTickets = n
        return nil
    }
}

func WithQueueDepth(n int) Option {
    return func(cfg *config) error {
        if n < 0 {
            return fmt.Errorf("queueDepth must be non-negative, got %d", n)
        }
        cfg.queueDepth = n
        return nil
    }
}

type Desk struct {
    cfg config
}

func NewDesk(opts ...Option) (*Desk, error) {
    cfg := config{
        name:       "default",
        maxTickets: 100,
        queueDepth: 10,
    }
    var errs []error
    for _, o := range opts {
        if err := o(&cfg); err != nil {
            errs = append(errs, err)
        }
    }
    if err := errors.Join(errs...); err != nil {
        return nil, err
    }
```

```
    return &Desk{cfg: cfg}, nil
}

func (d *Desk) String() string {
    return fmt.Sprintf("Desk{name:%q maxTickets:%d queueDepth:%d}",
        d.cfg.name, d.cfg.maxTickets, d.cfg.queueDepth)
}

func main() {
    logger := slog.New(slog.NewTextHandler(os.Stdout, nil))

    good, err := NewDesk(WithName("billing"), WithMaxTickets(40), WithQueueDepth(5))
    if err != nil {
        logger.Error("bad config", "err", err)
        os.Exit(1)
    }
    fmt.Println(good)

    // pass two bad options at once - both errors surface together
    _, err = NewDesk(WithName(""), WithMaxTickets(-3))
    if err != nil {
        logger.Error("bad config", "err", err)
    }
}
```

Run:

```
go run desk_validated.go
```

Output:

```
Desk{name:"billing" maxTickets:40 queueDepth:5}
time=... level=ERROR msg="bad config" err="desk name must not be empty\nmaxTickets must be posi
```

`errors.Join` collects all failures in one pass. The caller sees the full list, not just the first error.

Case 3: Composed options — a shorthand for test setup

A function can return `[]Option` and apply them as a batch. `WithDefaults` is not a single tweak — it is a curated starting point. Apply it first, then let the caller override specific fields.

Save as `desk_compose.go`:

```
// desk_compose.go
package main

import (
```

```
    "errors"
    "fmt"
)

type config struct {
    name       string
    maxTickets int
    queueDepth int
    verbose    bool
}

type Option func(*config) error

func WithName(s string) Option {
    return func(cfg *config) error {
        if s == "" {
            return errors.New("desk name must not be empty")
        }
        cfg.name = s
        return nil
    }
}

func WithMaxTickets(n int) Option {
    return func(cfg *config) error {
        if n <= 0 {
            return fmt.Errorf("maxTickets must be positive, got %d", n)
        }
        cfg.maxTickets = n
        return nil
    }
}

func WithQueueDepth(n int) Option {
    return func(cfg *config) error {
        cfg.queueDepth = n
        return nil
    }
}

func WithVerbose() Option {
    return func(cfg *config) error {
        cfg.verbose = true
        return nil
    }
}
```

```
// WithDefaults returns a standard set of options for test environments.
// Callers may append their own options to override individual fields.
func WithDefaults() []Option {
    return []Option{
        WithName("test-desk"),
        WithMaxTickets(10),
        WithQueueDepth(2),
        WithVerbose(),
    }
}

type Desk struct {
    cfg config
}

func NewDesk(opts ...Option) (*Desk, error) {
    cfg := config{}
    var errs []error
    for _, o := range opts {
        if err := o(&cfg); err != nil {
            errs = append(errs, err)
        }
    }
    if err := errors.Join(errs...); err != nil {
        return nil, err
    }
    return &Desk{cfg: cfg}, nil
}

func (d *Desk) String() string {
    return fmt.Sprintf("Desk{name:%q max:%d queue:%d verbose:%v}",
        d.cfg.name, d.cfg.maxTickets, d.cfg.queueDepth, d.cfg.verbose)
}

func main() {
    // Test helper: start from defaults, override one field.
    opts := append(WithDefaults(), WithMaxTickets(5))
    d, err := NewDesk(opts...)
    if err != nil {
        panic(err)
    }
    fmt.Println("from defaults:", d)

    // Production desk: explicit options, no defaults.
    prod, err := NewDesk(WithName("billing"), WithMaxTickets(200), WithQueueDepth(50))
    if err != nil {
        panic(err)
```

```
    }
    fmt.Println("production:   ", prod)
}
```

Run:

```
go run desk_compose.go
```

Output:

```
from defaults: Desk{name:"test-desk" max:5 queue:2 verbose:true}
production:    Desk{name:"billing" max:200 queue:50 verbose:false}
```

`WithDefaults()` returns a plain slice — not a special type. `append` combines it with extra options. The last write wins: `WithMaxTickets(5)` overwrites the `10` set by `WithDefaults`.

The trap

A long parameter list breaks callers every time you add a field.

```
// desk_trap.go - do NOT copy this pattern
package main

import (
    "fmt"
    "time"
)

type Desk struct {
    name    string
    max     int
    verbose bool
    timeout time.Duration
}

// Adding a fifth parameter here forces edits at every call site.
func NewDesk(name string, max int, verbose bool, timeout time.Duration) *Desk {
    return &Desk{name: name, max: max, verbose: verbose, timeout: timeout}
}

func main() {
    d := NewDesk("billing", 100, false, 30*time.Second)
    fmt.Printf("%+v\n", d)
}
```

Run:

```
go run desk_trap.go
```

Output:

```
&{name:billing max:100 verbose:false timeout:30s}
```

The program runs. The problem is the day you add a fifth parameter — `retryLimit int`. Every single call site fails to compile. In a large codebase that might mean dozens of files. With functional options, adding `WithRetryLimit` is a two-line addition and touches zero existing callers.

Save the corrected version as `desk_fixed.go`:

```
// desk_fixed.go
package main

import (
    "fmt"
    "time"
)

type config struct {
    name       string
    max        int
    verbose    bool
    timeout    time.Duration
    retryLimit int // added later - zero call sites changed
}

type Option func(*config)

func WithName(s string) Option           { return func(c *config) { c.name = s } }
func WithMax(n int) Option               { return func(c *config) { c.max = n } }
func WithVerbose() Option                { return func(c *config) { c.verbose = true } }
func WithTimeout(d time.Duration) Option { return func(c *config) { c.timeout = d } }
func WithRetryLimit(n int) Option        { return func(c *config) { c.retryLimit = n } }

type Desk struct{ cfg config }

func NewDesk(opts ...Option) *Desk {
    cfg := config{max: 100, timeout: 30 * time.Second}
    for _, o := range opts {
        o(&cfg)
    }
    return &Desk{cfg: cfg}
}

func (d *Desk) String() string {
```

```
    return fmt.Sprintf("Desk{name:%q max:%d verbose:%v timeout:%v retryLimit:%d}",
        d.cfg.name, d.cfg.max, d.cfg.verbose, d.cfg.timeout, d.cfg.retryLimit)
}

func main() {
    // existing call site - unchanged despite adding retryLimit
    d1 := NewDesk(WithName("billing"), WithMax(100), WithTimeout(30*time.Second))
    fmt.Println(d1)

    // new call site that uses the new option
    d2 := NewDesk(WithName("express"), WithRetryLimit(3))
    fmt.Println(d2)
}
```

Run:

```
go run desk_fixed.go
```

Output:

```
Desk{name:"billing" max:100 verbose:false timeout:30s retryLimit:0}
Desk{name:"express" max:100 verbose:false timeout:30s retryLimit:3}
```

`d1` was written before `WithRetryLimit` existed. It compiled then; it compiles now. No change required.

The boring rule

- Use `func(*config)` options when a constructor has **three or more optional knobs**.
- Use `func(*config) error` options when any knob has invalid states that must be caught at construction time.
- Use a plain struct literal when **all fields are required** — there is nothing to be optional about and the struct is self-documenting.
- Put option functions in the same package as the type. They are the public API for configuration.
- Name options `WithX`, never `SetX`. `Set` implies a method that mutates a live value; `With` implies configuration before creation.
- Return `[]Option` from helper functions (like `WithDefaults`) rather than a custom combinator type — plain slices compose with `append`.

Try this

1. In `desk_basic.go`, add `WithVerbose() Option` that sets a `verbose bool` field on `config`. Print it in `String()`. Call `NewDesk(WithVerbose())` and confirm the field appears.

2. In `desk_validated.go`, add cross-field validation: reject a `queueDepth` greater than `maxTickets`. You cannot do this inside a single option because both values must be set first. Add a `validate(cfg config) error` function called in `NewDesk` after the option loop.
3. In `desk_compose.go`, write `WithProductionDefaults() []Option` that sets `maxTickets: 500`, `queueDepth: 100`, and `verbose: false`. Call `NewDesk` once with production defaults and once with test defaults. Print both desks.
4. Starting from `desk_fixed.go`, change `WithTimeout` to return an error when the duration is shorter than one second. Change `NewDesk` to return `(*Desk, error)`. Update `main` to handle the error and try passing `WithTimeout(0)`.

Memory

Understanding Memory in Go

Understanding Memory in Go

Go passes arguments by value and decides for you whether a variable lives on the **stack** (tied to a function call) or the **heap** (alive until nothing refers to it). The boring default is to write straight-line code and let the compiler place things. You look at escape analysis when a profile says allocation is the problem — not before.

Mental model

A **stack frame** is the scratch space for one call: parameters, locals, return slots. When the function returns, that frame is gone. The **heap** is the shared pool the **garbage collector** (GC) scans. If the compiler cannot prove a variable dies with the frame, it **escapes**: the variable is allocated on the heap.

Escape is not a moral failing. Returning `&n` is legal Go. The compiler prints `moved to heap: n` and the GC frees `n` later. C programmers sometimes treat that as a bug. It is the language working.

`go build -gcflags='-m'` prints the compiler's notes: what inlined, what escaped. The first line is `# command-line-arguments` for a lone file, or `# your/module` inside a module.

Worked examples

Case 1: Pass by value copies the struct

Save as `copy_order.go`. `bump` receives its own `Order`. Adding 50 cents does not touch `main`'s `o`.

```
// copy_order.go
package main

import "fmt"

type Order struct {
    ID    int
    Table int
    Cents int
}

func bump(o Order) {
    o.Cents += 50
}
```

```
func main() {
    o := Order{ID: 9, Table: 4, Cents: 400}
    bump(o)
    fmt.Printf("id=%d cents=%d\n", o.ID, o.Cents)
}
```

Run:

```
go run copy_order.go
```

Output:

```
id=9 cents=400
```

A pointer parameter copies the pointer, not the struct. That is the next chapter. For a small `Order`, a copy is cheap and obvious.

Case 2: Locals that never escape

Save as `stackish.go`. No pointers, no interfaces, no `fmt`. The compiler inlines `add` into `main` and has nothing to heap-allocate.

```
// stackish.go
package main

func add(a, b int) int {
    s := a + b
    return s
}

func main() {
    _ = add(12, 8)
}
```

Build (not `go run` — we want the compiler notes, not a print):

```
go build -gcflags='-m' -o /tmp/stackish stackish.go
```

Output:

```
# command-line-arguments
./stackish.go:4:6: can inline add
./stackish.go:9:6: can inline main
./stackish.go:10:9: inlining call to add
```

No `moved to heap`. `s` is a number in a register or a frame. That is the stack story in practice: if the compiler can see the whole life of a value, it does not bother the GC.

Case 3: Returning an address forces the heap

Save as `escape.go`. `heapTicket` returns `*int`. `n` cannot live in `heapTicket`'s frame, because `main` still uses it afterward.

```
// escape.go
package main

import "fmt"

func localSum(a, b int) int {
    s := a + b
    return s
}

func heapTicket(id int) *int {
    n := id
    return &n
}

func main() {
    fmt.Println(localSum(12, 8))
    p := heapTicket(41)
    fmt.Println(*p)
}
```

Run:

```
go run escape.go
```

Output:

```
20
41
```

Now the notes:

```
go build -gcflags='-m' -o /tmp/escape escape.go
```

Output:

```
# command-line-arguments
./escape.go:6:6: can inline localSum
./escape.go:11:6: can inline heapTicket
./escape.go:17:22: inlining call to localSum
./escape.go:17:13: inlining call to fmt.Println
./escape.go:18:17: inlining call to heapTicket
```

```
./escape.go:19:13: inlining call to fmt.Println
./escape.go:12:2: moved to heap: n
./escape.go:17:13: ... argument does not escape
./escape.go:17:22: ~r0 escapes to heap
./escape.go:19:13: ... argument does not escape
./escape.go:19:14: *p escapes to heap
```

Read it in this order:

- `moved to heap: n` — the local in `heapTicket` cannot stay on the stack.
- `does not escape` — that value dies in the call (often an argument to `fmt`).
- `~r0 escapes to heap` — a return value the compiler named for itself, here because `fmt.Println` stores it in an `...any` list.

`fmt` makes noisy notes. When you are hunting allocations, compile a function without printing, or look at `go test -bench` and `pprof` later. `-m` is a flashlight, not a dashboard.

Case 4: A pointer parameter that does *not* escape

Save as `bump_ptr.go`. `o` is a pointer so we can mutate. The pointer itself never leaves `bump`, so the compiler says `o does not escape`. The `Order` can still sit in `main`'s frame.

```go
// bump_ptr.go
package main

import "fmt"

type Order struct {
    ID    int
    Cents int
}

func bump(o *Order) {
    o.Cents += 50
}

func main() {
    o := Order{ID: 9, Cents: 400}
    bump(&o)
    fmt.Printf("id=%d cents=%d\n", o.ID, o.Cents)
}
```

Run:

```
go run bump_ptr.go
```

Output:

```
id=9 cents=450
```

```
go build -gcflags='-m' -o /tmp/bump bump_ptr.go
```

Output:

```
# command-line-arguments
./bump_ptr.go:11:6: can inline bump
./bump_ptr.go:17:6: inlining call to bump
./bump_ptr.go:18:12: inlining call to fmt.Printf
./bump_ptr.go:11:11: o does not escape
./bump_ptr.go:18:12: ... argument does not escape
./bump_ptr.go:18:34: o.ID escapes to heap
./bump_ptr.go:18:40: o.Cents escapes to heap
```

`o does not escape` is the useful line. `o.ID escapes to heap` is `fmt.Printf` boxing numbers into `any`. Do not "fix" that by avoiding `Printf` in production logs; fix it when a profile says formatting is hot.

The trap

People coming from C refuse to return `&n`. They copy the struct into a `new` on purpose, or they take a pointer to a field of a global. In Go, return the address. The compiler already moved `n` in Case 3.

The other trap is Case 1: you passed a struct, mutated it, and wondered why the desk still shows the old price. That is not a memory leak. That is a copy. Use a pointer when you mean "change this one."

The boring rule

- Assume pass by value. Draw a pointer only when you need mutation or a shared identity.
- Returning `&local` is fine. It allocates. It does not dangle.
- Do not micro-manage stack vs heap. Read `-gcflags='-m'` when an allocation shows up in a profile.
- Ignore most `fmt` escape notes. They are the printer, not your order type.
- A pointer parameter that does not escape is still a copy of an address — cheap, and enough to mutate.

Try this

1. In `copy_order.go`, change `bump` to take `*Order` and call `bump(&o)`. Confirm cents become 450.
2. In `escape.go`, stop returning `&n`. Return `n` as an `int`. Rebuild with `-gcflags='-m'` and see whether `moved to heap: n` disappears.
3. Add `fmt.Println(s)` inside `add` in `stackish.go` (you will need `import "fmt"`). Rebuild with `-m`. Note the extra escape lines. That is the printer, not `s` becoming "bad."
4. Run `go build -gcflags='-m=2'` on `escape.go` for more detail. Skim; do not rewrite the program to silence every line.

Pointers Explained

Pointers Explained

A pointer is a value that holds the address of another value. The boring default is: take an address with `&`, follow it with `*`, check `nil` before you follow, and never do arithmetic. Go is not C.

Mental model

Piece	Meaning
`T`	a value of type `T`
`*T`	a pointer to a `T`
`&x`	the address of `x` (type `*T` if `x` is `T`)
`*p`	the `T` that `p` points at
`nil`	a pointer that points at nothing

Zero value of `*T` is `nil`. Following it panics. `new(T)` allocates a zero `T` and returns `*T` — same idea as `p := new(int)` versus `var n int; p := &n`, except `new` does not give you a named local.

There is no `p + 1`, no `p[i]` on a pointer to a single value, no `free`. The GC owns the bytes. Slices and maps are the tools for collections; pointer arithmetic is not.

Worked examples

Case 1: `&` and `*`

Save as `table_ptr.go`. `p` and `&table` are the same address. Writing through `*p` writes `table`.

```
// table_ptr.go
package main

import "fmt"

func main() {
    table := 12
    p := &table
    fmt.Println("table:", table)
    fmt.Println("same address:", p == &table)
    fmt.Println("*p:", *p)
    *p = 7
```

```
    fmt.Println("table after:", table)
}
```

Run:

```
go run table_ptr.go
```

Output:

```
table: 12
same address: true
*p: 12
table after: 7
```

We do not print `p` itself. The hex address changes every run and teaches nothing.

Case 2: `nil` is a value you must test

Save as `label_table.go`. A function that takes `*int` should decide what `nil` means *before* it stars the pointer.

```
// label_table.go
package main

import "fmt"

func label(p *int) string {
    if p == nil {
        return "no table"
    }
    return fmt.Sprintf("table %d", *p)
}

func main() {
    fmt.Println(label(nil))
    n := 4
    fmt.Println(label(&n))
}
```

Run:

```
go run label_table.go
```

Output:

```
no table
table 4
```

`nil` here is a legitimate "no table," not a crash. Document that in the function name or a comment if the rest of the desk might guess wrong.

Case 3: `new` gives a zero and a pointer

Save as `new_table.go`. `new(int)` is a pointer to `0`. Useful when you need a `*T` and do not already have a variable.

```
// new_table.go
package main

import "fmt"

func main() {
    p := new(int)
    fmt.Println("*p:", *p)
    *p = 12
    fmt.Println("*p:", *p)
}
```

Run:

```
go run new_table.go
```

Output:

```
*p: 0
*p: 12
```

For structs, composite literals are clearer: `p := &Ticket{ID: 41}` instead of `p := new(Ticket); p.ID = 41`. Keep `new` for simple zeros, or skip it.

Case 4: Pointer arithmetic does not exist

Save as `arith.go`. This is not a program you run successfully. It is the compiler telling you Go will not pretend a pointer is an index.

```
// arith.go
package main

func main() {
    table := 12
    p := &table
    p = p + 1
    _ = p
}
```

Run:

```
go run arith.go
```

Output:

```
# command-line-arguments
./arith.go:7:6: invalid operation: p + 1 (mismatched types *int and untyped int)
```

Walk a slice with `range` or an index. Walk a struct with field names. If you think you need to add to a pointer, you want a slice, an array, or a redesign.

Case 5: Struct pointers and automatic dereferencing

Save as `struct_ptr.go`. In C, you must switch between `.` and `->`. In Go, `t.Table` is automatic syntactic sugar for `(*t).Table`.

```
// struct_ptr.go
package main

import "fmt"

type Ticket struct {
    ID    int
    Table int
}

func main() {
    t := &Ticket{ID: 101, Table: 4}
    // Automatic dereference: t.Table means (*t).Table
    t.Table = 7
    (*t).ID = 102
    fmt.Printf("ticket %d at table %d\n", t.ID, t.Table)
}
```

Run:

```
go run struct_ptr.go
```

Output:

```
ticket 102 at table 7
```

Both forms compile and mutate the same backing struct. The dot notation `t.Table` is idiomatic.

Case 6: Returning pointers to local variables is safe

In languages like C or C++, returning the address of a local stack variable produces a dangling pointer and undefined memory behavior. In Go, the compiler performs **escape analysis**: if the address of a local variable escapes the function scope, Go automatically allocates that variable on the heap.

Save as `ticket_factory.go`:

```
// ticket_factory.go
package main

import "fmt"

type Ticket struct {
    ID    int
    Table int
}

func newTicket(id, table int) *Ticket {
    t := Ticket{ID: id, Table: table}
    return &t // Escapes to heap: safe in Go!
}

func main() {
    t1 := newTicket(41, 12)
    t2 := newTicket(42, 14)
    fmt.Println("t1:", t1.ID, t1.Table)
    fmt.Println("t2:", t2.ID, t2.Table)
}
```

Run:

```
go run ticket_factory.go
```

Output:

```
t1: 41 12
t2: 42 14
```

You do not need to call `malloc` or worry about stack lifetimes. Go manages the allocation location for you.

The trap

A nil pointer prints as `<nil>` and then explodes when you follow it. The print is not a safety check.

Save as `nil_deref.go`:

```
// nil_deref.go
package main

import "fmt"

func main() {
    var p *int
    fmt.Println(p)
    fmt.Println(*p)
}
```

Run:

```
go run nil_deref.go
```

Output (the `pc=` hex and file path depend on your machine):

```
<nil>
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x0 pc=0x...]

goroutine 1 [running]:
main.main()
    nil_deref.go:9
exit status 2
```

The fix is Case 2: test `p == nil` before `*p`. In methods, either refuse to run on a nil receiver or handle nil on purpose. Do not assume "I constructed this, so it cannot be nil" once the pointer has crossed a function boundary.

The boring rule

- `&x` to share or mutate `x`. `*p` to read or write the value.
- Use `t.Field` directly on struct pointers; do not write `(*t).Field` unless resolving ambiguity.
- Returning `&local` from a function is idiomatic and completely safe. Go's escape analysis moves it to the heap.
- Check `nil` at the edge where a missing pointer is allowed. Panic is for bugs, not for empty tables.
- Prefer `&Struct{...}` over `new` plus field assignments.
- Never add, subtract, or index a pointer. Use a slice.
- Do not print addresses in logs. Print the fields that matter to the desk.

Try this

1. In `table_ptr.go`, add a function `retarget(p *int, n int)` that sets `*p = n`. Call it with `&table` and 3.
2. In `ticket_factory.go`, run `go build -gcflags="-m" ticket_factory.go` to see the compiler's escape analysis report `moved to heap: t`.
3. In `label_table.go`, pass a pointer to `0`. Confirm you print `table 0`, not `no table`. Nil and zero are different.
4. Replace `new(int)` in `new_table.go` with a named `var n int` and `p := &n`. Same prints.
5. Try `p++` in `arith.go`. Same kind of error: pointers are not counters.

Mutability and Data Sharing

Mutability and Data Sharing

A slice and a map are small headers that point at data. Passing them copies the header, not the elements. The boring default is: if a function should not change the caller's data, copy first — or do not write through the header.

Mental model

A **slice header** is three words: pointer to an array, length, capacity. `func f(s []int)` copies those three words. Both headers can point at the same array. `s[i] = x` is visible to the caller. `append` is visible to the caller **when the array has spare capacity**; if `append` must grow, it may allocate a new array and then only the returned header sees the new element.

A **map** is already a pointer under the hood. `func f(m map[int]string)` copies the pointer. `m[k] = v` and `delete(m, k)` mutate the same map.

Structs and arrays are copied in full. Slices, maps, and pointers are the usual ways two functions share memory by accident.

Worked examples

Case 1: Writing through a slice header

Save as `mark_paid.go`. `markPaid` zeros the first item. `order` in `main` changes, because both headers share the array.

```
// mark_paid.go
package main

import "fmt"

func markPaid(cents []int) {
    if len(cents) == 0 {
        return
    }
    cents[0] = 0
}

func main() {
    order := []int{450, 120, 90}
```

```
    markPaid(order)
    fmt.Println(order)
}
```

Run:

```
go run mark_paid.go
```

Output:

```
[0 120 90]
```

That is useful when the function is *meant* to edit the order. It is a bug when the function was supposed to preview a discount.

Case 2: `copy` gives you a private array

Save as `mark_paid_copy.go`. Allocate a new slice, `copy` the elements, mutate the copy.

```
// mark_paid_copy.go
package main

import "fmt"

func markPaid(cents []int) {
    if len(cents) == 0 {
        return
    }
    cents[0] = 0
}

func main() {
    order := []int{450, 120, 90}
    safe := make([]int, len(order))
    copy(safe, order)
    markPaid(safe)
    fmt.Println("order:", order)
    fmt.Println("safe:", safe)
}
```

Run:

```
go run mark_paid_copy.go
```

Output:

```
order: [450 120 90]
safe: [0 120 90]
```

`append([]int(nil), order...)` is another way to copy. `copy` makes the length obvious. Either is fine. `safe := order` is **not** a copy; it copies the header only.

Case 3: Maps are shared even without a pointer in the signature

Save as `close_table.go`. `delete` inside `closeTable` removes Bo from `main`'s map.

```go
// close_table.go
package main

import "fmt"

func closeTable(open map[int]string, n int) {
    delete(open, n)
}

func main() {
    open := map[int]string{4: "Amina", 7: "Bo", 12: "Chen"}
    closeTable(open, 7)
    fmt.Println(open)
}
```

Run:

```
go run close_table.go
```

Output:

```
map[4:Amina 12:Chen]
```

(`fmt` prints maps with keys in a stable order. Ranging a map yourself is still random.)

There is no `copy` for maps in the builtin set. To snapshot, allocate a new map and assign in a loop. If the function should not mutate, do not call `delete` or write `m[k]`.

The trap

`append` on a slice that still has capacity writes into the original array. The caller's slice length does not grow, but its elements can change.

Save as `add_item.go`:

```
// add_item.go
package main

import "fmt"

func addItem(cents []int, extra int) []int {
    return append(cents, extra)
}

func main() {
    order := make([]int, 2, 4)
    order[0], order[1] = 450, 120
    next := addItem(order, 90)
    next[0] = 1
    fmt.Println("order:", order)
    fmt.Println("next:", next)
}
```

Run:

```
go run add_item.go
```

Output:

```
order: [1 120]
next: [1 120 90]
```

`order` never got a third element, but `next[0] = 1` overwrote `order[0]`. Spare capacity made the headers aliases.

The fix is to copy into a new array before `append`, so growth cannot land in the caller's backing store:

```
// add_item_copy.go
package main

import "fmt"

func addItem(cents []int, extra int) []int {
    out := make([]int, len(cents), len(cents)+1)
    copy(out, cents)
    return append(out, extra)
}

func main() {
    order := make([]int, 2, 4)
    order[0], order[1] = 450, 120
```

```
    next := addItem(order, 90)
    next[0] = 1
    fmt.Println("order:", order)
    fmt.Println("next:", next)
}
```

Run:

```
go run add_item_copy.go
```

Output:

```
order: [450 120]
next: [1 120 90]
```

If `addItem` is *supposed* to share — a builder that owns the buffer — return the new header and stop using the old one. The bug is using both.

The boring rule

- Slice and map parameters are shared storage unless you copy.
- `s2 := s` copies the header. `copy` (or `append` onto a fresh slice) copies elements.
- After `s = append(s, x)`, treat `s` as the only header you still use.
- Name functions after mutation: `markPaid`, `closeTable`. If the name is `preview`, copy inside.
- Do not clone maps and slices "just in case" on every call. Clone at the boundary where ownership splits.

Try this

1. In `mark_paid.go`, replace `cents[0] = 0` with `cents = append(cents, 0)` and print `order` in `main`. Length in `main` does not change (the header in `main` was not updated).
2. In `mark_paid_copy.go`, write `safe := order` instead of `make+copy`. Confirm `order` is mutated again.
3. In `close_table.go`, add `open[4] = "Dana"` inside `closeTable`. Confirm `main` sees Dana.
4. Change `add_item.go` so `order` is built with `[]int{450, 120}` (length 2, capacity 2). `append` must allocate. See whether `next[0] = 1` still changes `order`.

Garbage Collection

Garbage Collection

Go's garbage collector frees heap memory you no longer reference. It is not `malloc`/`free` with a different name. The boring default is: allocate normally, drop references when you are done, and reuse buffers only after a measurement says allocation is the hot path.

Mental model

You allocate by creating values the compiler cannot keep on the stack: `&Ticket{...}`, `make([]T, n)`, growing `append`, converting to `any`, and so on. You **do not** free. When no pointer can reach a value, the GC may reclaim it.

GOGC is the target heap growth, as a percent. `100` means "run a collection after the heap grows about 100% since the last one." Higher GOGC: fewer collections, more memory. Lower GOGC: more collections, a smaller heap. `GOGC=off` turns the collector off — a debug trick, not a production plan.

`runtime.ReadMemStats` snapshots counters. They move a little between runs. Use them to see *direction* (ten thousand tickets appeared; then they left), not to assert an exact byte count in a book.

Worked examples

Case 1: Live pointers keep objects; dropping them lets GC run

Save as `tickets_gc.go`. Ten thousand tickets stay alive because `held` points at them. Setting `held = nil` drops the only reference. `runtime.GC()` asks for a collection now so the numbers are readable. You rarely call that in a service.

```
// tickets_gc.go
package main

import (
    "fmt"
    "runtime"
)

type Ticket struct {
    ID    int
    Table int
```

```
    Note  string
}

func main() {
    runtime.GC()
    var before runtime.MemStats
    runtime.ReadMemStats(&before)

    held := make([]*Ticket, 0, 10_000)
    for i := range 10_000 {
        held = append(held, &Ticket{ID: i, Table: i % 20, Note: "soup"})
    }

    var after runtime.MemStats
    runtime.ReadMemStats(&after)
    fmt.Printf("tickets: %d\n", len(held))
    fmt.Printf("heap objects gained: %d\n", after.HeapObjects-before.HeapObjects)

    held = nil
    runtime.GC()
    var done runtime.MemStats
    runtime.ReadMemStats(&done)
    fmt.Printf("gc runs while held: %d\n", after.NumGC-before.NumGC)
    fmt.Printf("gc runs after drop: %d\n", done.NumGC-after.NumGC)
}
```

Run:

```
go run tickets_gc.go
```

Output (object counts can shift by a few on another machine; the 10000 tickets are the point):

```
tickets: 10000
heap objects gained: 10001
gc runs while held: 0
gc runs after drop: 1
```

While `held` was live, the collector did not need to run. After the drop, one GC pass ran and the extra objects were eligible. You did not call `free`.

Case 2: GOGC is a dial, not a rewrite

Save as `gogc.go`. `debug.SetGCPercent` returns the previous percent. The environment variable `GOGC` sets the starting value (`100` if unset).

```go
// gogc.go
package main

import (
    "fmt"
    "runtime/debug"
)

func main() {
    prev := debug.SetGCPercent(200)
    fmt.Println("previous GOGC:", prev)
    now := debug.SetGCPercent(200)
    fmt.Println("current GOGC:", now)
}
```

Run:

```
go run gogc.go
```

Output:

```
previous GOGC: 100
current GOGC: 200
```

Run with a tighter target:

```
GOGC=50 go run gogc.go
```

Output:

```
previous GOGC: 50
current GOGC: 200
```

Leave GOGC at the default until a memory limit or a latency graph says otherwise. `GOMEMLIMIT` (not shown here) is the modern partner for "do not use more than this." Do not sprinkle `SetGCPercent` inside request handlers.

Case 3: Reuse a slice when a profile says so

Save as `reuse.go`. Filling a fresh slice 50 times allocates. Reusing one backing array with `buf = buf[:0]` does not. The numbers below are malloc counts from `MemStats`, not a promise for every Go version.

```go
// reuse.go
package main
```

```
import (
    "fmt"
    "runtime"
)

func fill(buf []int, n int) []int {
    buf = buf[:0]
    for i := range n {
        buf = append(buf, i)
    }
    return buf
}

func allocEach(n, times int) uint64 {
    runtime.GC()
    var before runtime.MemStats
    runtime.ReadMemStats(&before)
    for range times {
        _ = fill(nil, n)
    }
    var after runtime.MemStats
    runtime.ReadMemStats(&after)
    return after.Mallocs - before.Mallocs
}

func reuse(n, times int) uint64 {
    runtime.GC()
    var before runtime.MemStats
    runtime.ReadMemStats(&before)
    buf := make([]int, 0, n)
    for range times {
        buf = fill(buf, n)
    }
    var after runtime.MemStats
    runtime.ReadMemStats(&after)
    _ = buf
    return after.Mallocs - before.Mallocs
}

func main() {
    fmt.Println("fresh slices mallocs:", allocEach(1000, 50))
    fmt.Println("reused slice mallocs:", reuse(1000, 50))
}
```

Run:

```
go run reuse.go
```

Output:

```
fresh slices mallocs: 450
reused slice mallocs: 1
```

Reuse won because we measured a tight loop. A desk program that builds one order list per request does not need this. `sync.Pool` is the same idea with concurrency; skip it until a benchmark names allocation as the cost.

Case 4: Setting a memory ceiling with `GOMEMLIMIT`

Save as `memlimit.go`. In containerized environments, Go services historically risked getting killed by the operating system OOM killer because `GOGC` only knows about the relative growth of the live heap, not the container's hard memory boundary. `GOMEMLIMIT` sets a soft ceiling: when total memory approaches this limit, the Go runtime triggers garbage collection proactively.

```go
// memlimit.go
package main

import (
    "fmt"
    "runtime/debug"
)

func main() {
    // Query current limit (-1 queries without changing)
    prev := debug.SetMemoryLimit(-1)
    fmt.Println("initial limit no-limit:", prev < 0 || prev > 1<<60)

    // Set soft memory limit to 128 MiB
    limit128MiB := int64(128 * 1024 * 1024)
    debug.SetMemoryLimit(limit128MiB)
    current := debug.SetMemoryLimit(-1)
    fmt.Printf("current limit: %d MiB\n", current/(1024*1024))
}
```

Run:

```
go run memlimit.go
```

Output:

```
initial limit no-limit: true
current limit: 128 MiB
```

You can set this in code with `debug.SetMemoryLimit`, or externally via the environment variable `GOMEMLIMIT=128MiB`. Set it to ~90% of your container's memory limit to provide headroom for the runtime and OS.

The trap

Trap 1: Subslice memory retention

When you slice a small header from a large buffer or payload, the subslice retains a pointer to the **entire original backing array**. The garbage collector cannot free that array as long as your subslice is reachable.

Save as `subslice_leak.go`:

```
// subslice_leak.go
package main

import (
    "fmt"
    "slices"
)

func main() {
    big := make([]byte, 10*1024*1024) // 10 MiB payload
    leaky := big[:4]
    clean := slices.Clone(big[:4])

    fmt.Printf("leaky len=%d cap=%d bytes pinned\n", len(leaky), cap(leaky))
    fmt.Printf("clean len=%d cap=%d bytes pinned\n", len(clean), cap(clean))
}
```

Run:

```
go run subslice_leak.go
```

Output:

```
leaky len=4 cap=10485760 bytes pinned
clean len=4 cap=8 bytes pinned
```

`leaky` pinned all 10 megabytes of memory for a 4-byte slice. `slices.Clone` (or `copy`) copied the 4 bytes into an independent backing array, allowing the 10 MiB payload to be freed.

Trap 2: Treating GC like manual malloc

Treating GC like a poorly hidden `malloc` leads to two bad desks: either you never allocate (unreadable code, buffer pools for a 20-line tool) or you `runtime.GC()` after every ticket "to keep memory tidy." The second makes the process stall for no reason. The first makes every change a puzzle.

The collector is part of the language. Write clear code. Keep references only while you need the values. Measure. Then reuse.

The boring rule

- Drop the last pointer when you are done (`held = nil`, end of function, clear a map). That is the free.
- Do not call `runtime.GC()` in normal request paths.
- Leave `GOGC` alone until you have a memory or pause graph.
- Set `GOMEMLIMIT` in container environments to prevent unexpected OOM kills.
- Use `slices.Clone` or `copy` when keeping a tiny subslice from a large buffer, so the large buffer can be reclaimed.
- Reuse slices (`s = s[:0]`) and buffers after a benchmark, not after a blog post.
- `MemStats` is a snapshot. Log it in a debug endpoint if you must; do not unit-test exact `Alloc` values.

Try this

1. In `tickets_gc.go`, comment out `held = nil` and keep `runtime.GC()`. How does `gc runs after drop` change? The tickets are still reachable.
2. In `subslice_leak.go`, replace `slices.Clone` with manual `copy` into a `make([]byte, 4)` slice. Verify `cap` is still tiny.
3. Run `GOMEMLIMIT=64MiB go run memlimit.go` and inspect the initial limit reported before modification.
4. Run `GOGC=off go run tickets_gc.go`. Read about `GOGC=off` in `go doc runtime`. Do not ship that.
5. In `reuse.go`, change `fill(nil, n)` to reuse a slice the same way `reuse` does. The two printed numbers should get close.

Interfaces

Designing with Interfaces

Designing with Interfaces

An interface is a set of methods. A type **satisfies** it by having those methods — no `implements` keyword. The boring default is a small interface at the point of use: one or two methods, a function that **accepts** the interface and **returns** a concrete struct.

Mental model

```
type Payer interface {
    Pay(cents int) string
}
```

Anything with `Pay(int) string` is a `Payer`. `Card` does not mention `Payer`. That is **implicit satisfaction**. Adding a method to `Card` does not break anyone; adding a method to `Payer` does, because every existing implementation must grow.

`io.Reader` is the standard-library version of this idea: one method, `Read([]byte) (int, error)`. `strings.NewReader` and `bytes.Buffer` both satisfy it. Your function takes `io.Reader` and does not care which.

A nil interface is a later chapter. Here: pass real values, keep interfaces small, return the type you actually built.

Worked examples

Case 1: Two types, one method, no registration

Save as `take_pay.go`. `take` prints whatever `Pay` returns. `Card` and `Tab` never name `Payer`.

```
// take_pay.go
package main

import "fmt"

type Payer interface {
    Pay(cents int) string
}

type Card struct{ Last4 string }
```

```
func (c Card) Pay(cents int) string {
    return fmt.Sprintf("card %s: %d cents", c.Last4, cents)
}

type Tab struct{ Table int }

func (t Tab) Pay(cents int) string {
    return fmt.Sprintf("table %d tab: %d cents", t.Table, cents)
}

func take(p Payer, cents int) {
    fmt.Println(p.Pay(cents))
}

func main() {
    take(Card{Last4: "4242"}, 450)
    take(Tab{Table: 12}, 180)
}
```

Run:

```
go run take_pay.go
```

Output:

```
card 4242: 450 cents
table 12 tab: 180 cents
```

The interface earned its keep because **two** implementations exist. One implementation is a function that takes `Card`.

Case 2: `io.Reader` with a string and a buffer

Save as `slurp.go`. `slurp` reads any `io.Reader`. `strings.NewReader` holds a string. `bytes.Buffer` is a growable buffer that also implements `Read`.

```
// slurp.go
package main

import (
    "bytes"
    "fmt"
    "io"
    "strings"
)
```

```
func slurp(r io.Reader) (string, error) {
    b, err := io.ReadAll(r)
    if err != nil {
        return "", err
    }
    return string(b), nil
}

func main() {
    note, err := slurp(strings.NewReader("hold the onions"))
    if err != nil {
        fmt.Println("err:", err)
        return
    }
    fmt.Println(note)

    var buf bytes.Buffer
    buf.WriteString("two coffees")
    drink, err := slurp(&buf)
    if err != nil {
        fmt.Println("err:", err)
        return
    }
    fmt.Println(drink)
}
```

Run:

```
go run slurp.go
```

Output:

```
hold the onions
two coffees
```

`slurp` does not import a file type. Tests can pass a `strings.NewReader`. Production can pass an `os.File`. That is the whole payoff of a one-method interface.

`bytes.Buffer`'s `Read` has a pointer receiver, so you pass `&buf`. `strings.NewReader` already returns a pointer.

Case 3: Accept an interface, return a struct

Save as `new_ticket.go`. The input can be any reader. The output is a `Ticket` you can field-select, JSON-encode, and store. Callers do not get a `NoteSource` interface they have to type-assert.

```
// new_ticket.go
package main

import (
    "fmt"
    "io"
    "strings"
)

type Ticket struct {
    ID   int
    Note string
}

func NewTicket(id int, r io.Reader) (Ticket, error) {
    b, err := io.ReadAll(r)
    if err != nil {
        return Ticket{}, err
    }
    return Ticket{ID: id, Note: string(b)}, nil
}

func main() {
    t, err := NewTicket(41, strings.NewReader("soup"))
    if err != nil {
        fmt.Println("err:", err)
        return
    }
    fmt.Printf("ticket %d: %s\n", t.ID, t.Note)
}
```

Run:

```
go run new_ticket.go
```

Output:

```
ticket 41: soup
```

`NewTicket` would be weaker as `func NewTicket(...) (io.Reader, error)` or `func NewTicket(...) (Payer, error)`. You built a ticket. Return a ticket.

The trap

Returning an interface from a constructor that only ever builds one type hides the fields and forces every caller into the interface, including tests that wanted `Last4`.

Save as `new_payer.go`:

```
// new_payer.go
package main

import "fmt"

type Payer interface {
    Pay(cents int) string
}

type Card struct{ Last4 string }

func (c Card) Pay(cents int) string {
    return fmt.Sprintf("card %s: %d cents", c.Last4, cents)
}

func newPayer() Payer {
    return Card{Last4: "4242"}
}

func main() {
    p := newPayer()
    fmt.Println(p.Pay(450))
}
```

Run:

```
go run new_payer.go
```

Output:

```
card 4242: 450 cents
```

It works. `p.Last4` does not compile. The fix is `func newCard() Card` (or `*Card` if you need a pointer). Let *callers* store it in a `Payer` if they need to.

The boring rule

- Define interfaces with the methods you call, not the methods you might one day call.
- Two real implementations (or a test double you cannot avoid) justify an interface. One does not.
- Accept interfaces, return structs.
- Steal small shapes from the standard library (`io.Reader`, `io.Writer`, `fmt.Stringer`) instead of inventing `TicketReader`.
- Pointer vs value receivers still decide the method set. Pass `&buf` when `Read` is on `*bytes.Buffer`.

Try this

1. Add a `Cash` type to `take_pay.go` with `Pay` that prints `cash: N cents`. Call `take(Cash{}, 90)`.
2. In `slurp.go`, pass `os.Stdin` instead of the string reader and run `echo extra napkins | go run slurp.go`. You will need `import "os"`.
3. Change `NewTicket` to return `*Ticket`. Keep the `io.Reader` parameter. Confirm `t.Note` still works.
4. In `new_payer.go`, change `newPayer` to return `Card`. Print `p.Last4` in `main`.

Interface Best Practices

Interface Best Practices

Interfaces belong to the **consumer** — the function that calls the methods — not to the type that happens to have them. The boring default is a concrete struct until a second implementation appears. `any` (`interface{}`) is not a junk drawer for the desk.

Mental model

A package that stores tickets should export `Ticket`, not `Ticketer`. The kitchen package that *prints* labels can declare:

```
type Labeler interface {
    Label() string
}
```

next to `announce`, because `announce` is what needs `Label`. `Ticket` in another file grows a `Label` method and satisfies `Labeler` without importing the kitchen.

Premature interfaces (one implementation, a constructor that returns the interface, a file named `interfaces.go`) make the code harder to read and harder to change. Empty interface (`any`) throws the type checker away. Use a real type. When many types need the same function, part 09 (generics) is the tool — not `func f(v any)`.

Worked examples

Case 1: Concrete first

Save as `send_kitchen.go`. One type, one function, no interface. This is the whole design until a second label shape exists.

```
// send_kitchen.go
package main

import "fmt"

type Ticket struct {
    ID    int
    Table int
}
```

```
func (t Ticket) Label() string {
    return fmt.Sprintf("ticket %d → table %d", t.ID, t.Table)
}

func send(t Ticket) {
    fmt.Println("kitchen:", t.Label())
}

func main() {
    send(Ticket{ID: 41, Table: 12})
}
```

Run:

```
go run send_kitchen.go
```

Output:

```
kitchen: ticket 41 → table 12
```

If you feel the urge to extract `KitchenService` now, wait. A second type is a better reason than a slogan.

Case 2: Interface at the consumer, when there are two types

Save as `announce.go`. `Labeler` sits next to `announce`, which is the consumer. `Ticket` and `Note` do not import it; they just have `Label()`.

```
// announce.go
package main

import "fmt"

type Labeler interface {
    Label() string
}

type Ticket struct {
    ID    int
    Table int
}

func (t Ticket) Label() string {
    return fmt.Sprintf("ticket %d → table %d", t.ID, t.Table)
}
```

```
type Note struct {
    Text string
}

func (n Note) Label() string { return n.Text }

func announce(items []Labeler) {
    for _, it := range items {
        fmt.Println("up:", it.Label())
    }
}

func main() {
    announce([]Labeler{
        Ticket{ID: 41, Table: 12},
        Note{Text: "86 soup"},
    })
}
```

Run:

```
go run announce.go
```

Output:

```
up: ticket 41 → table 12
up: 86 soup
```

The slice type is `[]Labeler` because `announce` needs to range mixed values. That is a real consumer need. A `[]Ticket` would have been enough for Case 1.

Case 3: `any` erases the desk

Save as `print_any.go`. It compiles. The function cannot add cents, cannot call `Label`, cannot tell a table from a soup. You pushed the type check to every caller, or into a type assertion (next chapter).

```
// print_any.go
package main

import "fmt"

func printAny(v any) {
    fmt.Println(v)
}
```

```
func main() {
    printAny(12)
    printAny("soup")
    printAny([]int{1, 2})
}
```

Run:

```
go run print_any.go
```

Output:

```
12
soup
[1 2]
```

`fmt.Println` itself takes `...any` because printing truly is "whatever." Your order total is not "whatever." Write `func add(cents int)` or, later, a generic `func Sum[T int | int64](xs []T) T`. Do not store tickets in `[]any`.

Case 4: Compile-time interface verification

Save as `interface_check.go`. If a struct is intended to implement an interface, how do you prevent someone from renaming a method and breaking callers silently? You declare a blank identifier assignment: `var _ Interface = (*ConcreteType)(nil)`.

```
// interface_check.go
package main

import "fmt"

type Printer interface {
    PrintLabel() string
}

type Ticket struct {
    ID int
}

func (t *Ticket) PrintLabel() string {
    return fmt.Sprintf("ticket #%d", t.ID)
}

// Compile-time check: ensures *Ticket satisfies Printer at build time.
var _ Printer = (*Ticket)(nil)
```

```
func main() {
    t := &Ticket{ID: 42}
    fmt.Println(t.PrintLabel())
}
```

Run:

```
go run interface_check.go
```

Output:

```
ticket #42
```

This line costs zero memory and runs zero instructions at runtime. If you later change `PrintLabel()` to `PrintLabel(prefix string)`, the compiler refuses to build immediately on the line where `_ Printer = (*Ticket)(nil)` sits.

Case 5: Consumer-side fakes for testing

Save as `notify_fake.go`. The caller defines what it requires (`Notifier`). In production, this might send network notifications. In tests, you implement a lightweight 5-line slice recorder right where the test lives.

```
// notify_fake.go
package main

import "fmt"

type Notifier interface {
    Notify(msg string)
}

type DeskService struct {
    notifier Notifier
}

func (s *DeskService) PlaceOrder(item string) {
    s.notifier.Notify("order placed: " + item)
}

// In unit tests, a fake recorder implementation:
type FakeNotifier struct {
    Messages []string
}

func (f *FakeNotifier) Notify(msg string) {
```

```
    f.Messages = append(f.Messages, msg)
}

func main() {
    fake := &FakeNotifier{}
    desk := &DeskService{notifier: fake}

    desk.PlaceOrder("toast")
    fmt.Printf("recorded %d notification: %s\n", len(fake.Messages), fake.Messages[0])
}
```

Run:

```
go run notify_fake.go
```

Output:

```
recorded 1 notification: order placed: toast
```

No heavyweight mocking framework, code generators, or dynamic proxies required.

The trap

An interface with one implementation, defined next to that implementation, is a costume. It looks like flexibility. It is a detour every time you need a field.

Save as `too_soon.go`:

```
// too_soon.go
package main

import "fmt"

type Sender interface {
    Send()
}

type Ticket struct {
    ID    int
    Table int
}

func (t Ticket) Send() {
    fmt.Printf("kitchen: ticket %d → table %d\n", t.ID, t.Table)
}
```

```
func NewSender(id, table int) Sender {
    return Ticket{ID: id, Table: table}
}

func main() {
    s := NewSender(41, 12)
    s.Send()
}
```

Run:

```
go run too_soon.go
```

Output:

```
kitchen: ticket 41 → table 12
```

`s.ID` does not compile. Tests cannot set `Table` without a type assertion. The boring version is Case 1: `func NewTicket(id, table int) Ticket` and `send(t Ticket)`.

When a test needs a fake, define a small interface in the *test's package* or next to the function under test — still the consumer — not on `Ticket`.

The boring rule

- Concrete structs until a second implementation is real.
- Declare interfaces where they are used, with the methods that are used.
- One-method interfaces named after the method (`Labeler`, `Payer`, `Reader`) beat `Manager` and `Service`.
- Use compile-time checks (`var _ Interface = (*Concrete)(nil)`) when exporting a type meant to satisfy a contract.
- Keep test fakes tiny and local to the test. Do not import heavy mocking libraries.
- Do not put `any` in your core types. `fmt` and `encoding/json` have reasons; your till does not.
- Do not invent `Ticketer` in the tickets package so that "someone might mock it." Mock (or stub) at the consumer.

Try this

1. In `send_kitchen.go`, add a `Note` type and change `send` to take `Labeler` only after both types exist. That is the right moment, not the first commit.
2. In `interface_check.go`, change `PrintLabel` to take an argument `func (t *Ticket) PrintLabel(p string) string`. Observe the exact compiler error from the `var _ Printer` check.
3. In `notify_fake.go`, add a second call `desk.PlaceOrder("tea")` and assert `len(fake.Messages) == 2`.

4. Move `type Labeler` in `announce.go` to sit directly above `announce` if you placed it elsewhere. The name should live with the function that needs it.
5. In `too_soon.go`, return `Ticket` from `NewSender` (rename it `NewTicket`). Print `t.Table` in `main`.

Type Assertions and Reflection Boundaries

Type Assertions and Reflection Boundaries

A type assertion asks an interface for a concrete type. The boring default is the **comma-ok** form, or a **type switch**, never a panic. A nil pointer stored in a non-nil interface is a different bug: the interface is not `nil`, and method calls still run.

Mental model

An interface value is a pair: **dynamic type** and **dynamic value**. A bare `var c Closer` has no type and no value — it is `nil`. A `var d *Drawer` that is `nil`, assigned to `c`, gives `c` type `*Drawer` and value `nil`. Then `c == nil` is false.

```
x.(T)       // panics if x does not hold T
x.(T) ok    // v, ok := x.(T)
switch x := v.(type) { case int: ... }
```

`any` is `interface{}`. Assertions on `any` are how you recover a type you threw away. Prefer not to throw it away.

Reflection (`reflect`) can look at types at run time. It exists. It is slow, ugly, and the right tool for things like `encoding/json`. It is not the tool for routing a ticket. A later chapter treats it as a topic. This chapter stops at the boundary: if you need `reflect` to call your own methods, the interface is too vague.

Worked examples

Case 1: Comma-ok instead of a panic

Save as `describe.go`. `v.(int)` without `ok` panics on the wrong type. With `ok`, you get a boolean and a zero `int`.

```
// describe.go
package main

import "fmt"

func describe(v any) {
    n, ok := v.(int)
    if !ok {
```

```
        fmt.Println("not a table number")
        return
    }
    fmt.Println("table", n)
}

func main() {
    describe(12)
    describe("booth")
}
```

Run:

```
go run describe.go
```

Output:

```
table 12
not a table number
```

Use this when you have one expected type. If you have two or three, use a type switch.

Case 2: A type switch at the desk

Save as `route_any.go`. Each `case` is a type. `x` has that type inside the case. `default` is required if you do not control every caller.

```
// route_any.go
package main

import "fmt"

func route(v any) {
    switch x := v.(type) {
    case int:
        fmt.Println("table", x)
    case string:
        fmt.Println("note", x)
    default:
        fmt.Println("unknown")
    }
}

func main() {
    route(12)
    route("hold the onions")
```

```
    route(true)
}
```

Run:

```
go run route_any.go
```

Output:

```
table 12
note hold the onions
unknown
```

A type switch on `any` is a smell if *you* built `v`. Make `v` an `int` or a `string` or a small interface instead. It is reasonable when you are at the edge of a decoder that already lost the type.

Case 3: Assertion without `ok` panics

Save as `bad_assert.go`. This is the form to avoid.

```
// bad_assert.go
package main

import "fmt"

func main() {
    var v any = "soup"
    n := v.(int)
    fmt.Println(n)
}
```

Run:

```
go run bad_assert.go
```

Output (path and offset vary):

```
panic: interface conversion: interface {} is string, not int

goroutine 1 [running]:
main.main()
    bad_assert.go:8
exit status 2
```

The panic text is exact enough: `interface {}` is `any`. The fix is Case 1.

The trap

A nil `*Drawer` inside a `Closer` is not a nil `Closer`. The nil check in `shut` does not save you. The method runs. `d.Name` follows a nil pointer.

Save as `typed_nil.go`:

```
// typed_nil.go
package main

import "fmt"

type Drawer struct {
    Name string
}

func (d *Drawer) Close() {
    fmt.Println("closed", d.Name)
}

type Closer interface {
    Close()
}

func shut(c Closer) {
    if c == nil {
        fmt.Println("nothing to close")
        return
    }
    c.Close()
}

func main() {
    var c Closer
    fmt.Println("bare interface nil:", c == nil)

    var d *Drawer
    c = d
    fmt.Println("typed nil in interface:", c == nil)

    shut(c)
}
```

Run:

```
go run typed_nil.go
```

Output (hex and path vary; the two printed lines and the panic message do not):

```
bare interface nil: true
typed nil in interface: false
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x0 pc=0x...]

goroutine 1 [running]:
main.(*Drawer).Close(...)
    typed_nil.go:11
main.shut(...)
    typed_nil.go:23
main.main()
    typed_nil.go:34
exit status 2
```

The interface held type `*Drawer` and value `nil`. `c == nil` was false. `Close` ran.

The fix is to put a true nil interface in, not a typed nil. Check the pointer *before* it becomes an interface:

```
// typed_nil_fix.go
package main

import "fmt"

type Drawer struct {
    Name string
}

func (d *Drawer) Close() {
    fmt.Println("closed", d.Name)
}

type Closer interface {
    Close()
}

func shut(c Closer) {
    if c == nil {
        fmt.Println("nothing to close")
        return
    }
    c.Close()
}

func main() {
    var d *Drawer
    if d == nil {
        shut(nil)
```

```
        return
    }
    shut(d)
}
```

Run:

```
go run typed_nil_fix.go
```

Output:

```
nothing to close
```

`shut(nil)` is a nil interface. `shut(d)` when `d` is a nil `*Drawer` is not. Returning `error` has the same trap: `var err *MyError; return err` is a non-nil `error`. Return the untyped `nil` instead: `return nil`.

Do not "fix" this with `reflect`. Check the concrete pointer, or do not store it.

The boring rule

- `v, ok := x.(T)`. Do not assert in a way that can panic unless a panic *is* the bug report.
- Type switch for two or three types at a boundary. For your own code, keep the type.
- Never store a nil pointer in an interface and then test the interface for `nil`.
- `return nil` for a nil `error` or `Closer`. Do not return a nil pointer of a concrete type.
- `reflect` is for codecs and tools. If you are reflecting a ticket, you wanted a field or a method.

Try this

1. In `describe.go`, assert `v.(string)` with comma-ok as a second branch so `"booth"` prints as a note.
2. In `route_any.go`, add `case bool:` and print `flag`. The `true` call should leave `default`.
3. In `typed_nil.go`, add `fmt.Printf("%T\n", c)` after `c = d`. You should see `*main.Drawer` while `c == nil` is still false.
4. Write `func wrap() error { var err *Drawer; return err }` in a scratch file (`Drawer` would need to implement `error` — or use a tiny `type E struct{}; func (*E) Error() string { return "x" }`). Print `wrap() == nil`. Then change the body to `return nil`.

Generics

Why Generics Matter

Why Generics Matter

Generics exist so you can write **one** helper for many types without copy-paste and without `any` plus a type assertion. The boring default is still a concrete function. Reach for a type parameter when you have already written the same logic twice.

Mental model

A **type parameter** is a name for a type the caller fills in. `min[T]` is not "min of anything." It is "min of one type `T`, chosen at the call site." The compiler substitutes `int`, `float64`, or `Cents` and type-checks the body against a **constraint** — the set of types `T` is allowed to be.

Before Go 1.18, the desk either duplicated `minInt` / `minFloat64` or hid the type behind `interface{}` and hoped. Duplication is honest but rot. `any` is one function that can panic at 5pm. Type parameters keep the honesty and drop the rot.

If every call uses the same type, you do not need a type parameter.

Worked examples

Case 1: The pre-generic pain

The desk tracks table counts (`int`) and prices in cents (`int64`). Two "smaller of these two" helpers. Same body. Two names. Two tests. Tomorrow someone adds `minFloat` for a tax rate.

Save as `dup_min.go`:

```
// dup_min.go
package main

import "fmt"

func minInt(a, b int) int {
    if a < b {
        return a
    }
    return b
}

func minInt64(a, b int64) int64 {
```

```
    if a < b {
        return a
    }
    return b
}

func main() {
    tables := minInt(4, 2)
    price := minInt64(1250, 990)
    fmt.Printf("tables %d, price %d cents\n", tables, price)
}
```

Run:

```
go run dup_min.go
```

Output:

```
tables 2, price 990 cents
```

The `<` is identical. The types are not interchangeable, so the compiler will not let you share the function. That is the pain: not "I wish I had a framework," but "I am about to paste a third copy."

Case 2: One generic function

`cmp.Ordered` is the standard-library constraint for types that support `<`. One function, two call sites, still statically typed.

Save as `generic_min.go`:

```
// generic_min.go
package main

import (
    "cmp"
    "fmt"
)

func min[T cmp.Ordered](a, b T) T {
    if a < b {
        return a
    }
    return b
}

func main() {
```

```
    tables := min(4, 2)
    price := min(int64(1250), int64(990))
    fmt.Printf("tables %d, price %d cents\n", tables, price)
}
```

Run:

```
go run generic_min.go
```

Output:

```
tables 2, price 990 cents
```

`min(4, 2)` infers `T` as `int`. `min(int64(1250), int64(990))` infers `int64`. Mixing `min(4, int64(990))` does not compile: there is no single `T`. That is a feature.

The standard library already has `min` and `max` built-ins for ordered values, and `slices.Min` / `slices.Max` for slices. You would not ship `min` as a helper at work. You *would* ship a generic when the duplicated logic is yours — clamping a desk value, picking the cheaper of two quotes, merging two sorted ticket lists.

Case 3: Slice helpers copy-pasted

Same story with slices. A "first matching table" helper for `[]int` and a "first matching name" helper for `[]string` will drift. One type parameter, one loop.

Save as `first.go`:

```
// first.go
package main

import "fmt"

func first[T comparable](list []T, want T) (T, bool) {
    var zero T
    for _, v := range list {
        if v == want {
            return v, true
        }
    }
    return zero, false
}

func main() {
    tables := []int{3, 7, 12}
    names := []string{"Amina", "Bo", "Chen"}
```

```
    t, ok := first(tables, 7)
    fmt.Printf("table %d found=%t\n", t, ok)

    n, ok := first(names, "Dee")
    fmt.Printf("name %q found=%t\n", n, ok)
}
```

Run:

```
go run first.go
```

Output:

```
table 7 found=true
name "" found=false
```

A miss returns the **zero value** of `T` (`""` for `string`) and `false`. `comparable` is the constraint for types that support `==`. Maps, slices, and functions are not comparable, so `first([][]int{...}, ...)` does not compile. The constraint is the documentation.

At work, prefer `slices.Contains` and `slices.Index` over writing `first` yourself. The next chapter does that. This chapter is the *reason* those functions can exist once.

Case 4: When not to use generics

This function only ever totals ticket prices in cents. There is one type. A type parameter would be a lie you tell the next reader: "this might be anything."

Save as `total.go`:

```
// total.go
package main

import "fmt"

func totalCents(prices []int) int {
    sum := 0
    for _, p := range prices {
        sum += p
    }
    return sum
}

func main() {
    fmt.Println(totalCents([]int{450, 800, 250}))
}
```

Run:

```
go run total.go
```

Output:

```
1500
```

Do not write `func total[T ~int](prices []T) T` until a second integer-like type actually shows up. One concrete type is enough.

The trap

A generic that exists to look generic. This program wraps `fmt.Println` in a type parameter. Every call still prints one value. The type parameter does no work.

Save as `too_generic.go`:

```
// too_generic.go
package main

import "fmt"

func announce[T any](v T) {
    fmt.Println(v)
}

func main() {
    announce("desk is open")
    announce(12)
}
```

Run:

```
go run too_generic.go
```

Output:

```
desk is open
12
```

`fmt.Println` already accepts any values. The boring version is `fmt.Println(v)`. A type parameter has to **remove duplication** or **preserve a concrete type** the caller needs back. If it does neither, delete it.

The boring rule

- Duplicate once. Genericize when the second copy appears, not before.
- Constrain `T`. `any` is last resort, not the default.
- If every call site uses one type, write that type.
- Prefer a standard-library generic (`slices.*`, `cmp.*`, `maps.*`) over a house helper.
- Generics do not replace interfaces. Interfaces are for *behavior*. Type parameters are for *containers and algorithms* that must keep the element type.

Try this

1. In `dup_min.go`, add `minFloat64`. Feel the paste. Then delete all three and call the built-in `min` instead.
2. In `generic_min.go`, try `min(4, int64(990))`. Read the compiler error. Fix it by converting one argument.
3. In `total.go`, do **not** add a type parameter. Add a `discount int` argument instead and subtract it from the sum, clamped at zero with the built-in `max`.

Generic Functions

Generic Functions

A generic function is a function whose signature names types the caller supplies. The boring default is not to write one. The boring default is to **call** `slices.Contains`, `slices.Clone`, `slices.Index`, and friends. Write your own when the standard library does not already do the job.

Mental model

The syntax is:

```
func Name[T Constraint](args) results
```

- `T` is a **type parameter**.
- `Constraint` is an interface (often a small one from `cmp` or a union of types). `any` means no extra methods. `comparable` means `==` is allowed. `~int` means `int` **or a defined type whose underlying type is** `int`.
- At the call site the compiler **infers** `T` from the arguments when it can. You write `Name[int](...)` only when inference fails.

The body is type-checked once against the constraint, then instantiated per concrete type. There is no boxing and no runtime type switch unless you add one.

Worked examples

Case 1: The standard library is the helper

Ticket IDs on a shift. Need "is 7 on this list?" and "a copy I can sort without touching the original." Do not invent `ContainsInt`.

Save as `shift_ids.go`:

```
// shift_ids.go
package main

import (
    "fmt"
    "slices"
)
```

```
func main() {
    ids := []int{3, 7, 12}
    fmt.Println("has 7:", slices.Contains(ids, 7))
    fmt.Println("has 9:", slices.Contains(ids, 9))

    copy := slices.Clone(ids)
    copy[0] = 99
    fmt.Println("original", ids)
    fmt.Println("clone   ", copy)
}
```

Run:

```
go run shift_ids.go
```

Output:

```
has 7: true
has 9: false
original [3 7 12]
clone    [99 7 12]
```

`slices.Clone` copies the slice header and the elements. Nested slices and maps inside elements are still shared; for integers that does not matter. `slices.Contains` requires `comparable` elements, which `int` is.

Case 2: `comparable` when you must write it

A tiny "index of this name on the roster." `slices.Index` already exists; this listing is so you can see a constraint in a function you own.

Save as `roster_index.go`:

```
// roster_index.go
package main

import "fmt"

func index[T comparable](list []T, want T) int {
    for i, v := range list {
        if v == want {
            return i
        }
    }
    return -1
}
```

```
func main() {
    roster := []string{"Amina", "Bo", "Chen"}
    fmt.Println("Bo at", index(roster, "Bo"))
    fmt.Println("Dee at", index(roster, "Dee"))
    fmt.Println("table 12 at", index([]int{3, 7, 12}, 12))
}
```

Run:

```
go run roster_index.go
```

Output:

```
Bo at 1
Dee at -1
table 12 at 2
```

Inference picks `T` from `list` and `want` together. After you have typed this once, delete it and use `slices.Index`. Same contract, same `-1`, maintained by the Go team.

Case 3: `~T` for defined types

The desk stores money as `Cents`, not raw `int`, so you cannot pass a `Cents` to a function that demands `int`. The approximation `~int` means "`int` or anything declared as `type X int`."

Save as `cents.go`:

```
// cents.go
package main

import "fmt"

type Cents int

func double[T ~int](v T) T {
    return v * 2
}

func main() {
    tip := Cents(150)
    fmt.Println(double(tip))
    fmt.Println(double(8))
}
```

Run:

```
go run cents.go
```

Output:

```
300
16
```

If the constraint were `int` instead of `~int`, `double(tip)` would not compile. `~` is how defined types keep their name while still using integer algorithms. Do not sprinkle `~` on every parameter. Use it when a defined type is a real part of the desk (money, IDs, table numbers) and the algorithm is numeric.

Case 4: Inference, and when to write the type

Most calls need no brackets. Composite literals, conversions, and channel sends can also infer in Go 1.27. This program shows a call that infers, a call that needs an explicit type (no arguments to infer from), and a slice of function values that infers in the literal.

Save as `infer.go`:

```
// infer.go
package main

import "fmt"

func zero[T any]() T {
    var v T
    return v
}

func label[T any](v T) string {
    return fmt.Sprintf("ticket:%v", v)
}

func main() {
    fmt.Printf("%q\n", zero[string]())
    fmt.Println(zero[int]())

    fmt.Println(label(41))

    printers := []func(int) string{label}
    fmt.Println(printers[0](41))
}
```

Run:

```
go run infer.go
```

Output:

```
""
0
ticket:41
ticket:41
```

`zero()` cannot infer: there are no arguments. You must write `zero[string]()`. `label(41)` infers `T` as `int`. A composite literal of type `[]func(int) string` infers `T` as `int` (Go 1.27). `append(s, label)` still needs `label[int]` — inference there is not a composite literal. If inference is ugly, write the type argument. Clarity beats a puzzle.

The trap

Hand-rolling what `slices` already provides, then getting the edge cases wrong. This "clone" shares the backing array.

Save as `fake_clone.go`:

```
// fake_clone.go
package main

import "fmt"

func fakeClone[T any](s []T) []T {
    return s[:]
}

func main() {
    ids := []int{3, 7, 12}
    copy := fakeClone(ids)
    copy[0] = 1
    fmt.Println("original", ids)
    fmt.Println("copy    ", copy)
}
```

Run:

```
go run fake_clone.go
```

Output:

```
original [1 7 12]
copy     [1 7 12]
```

`s[:]` reuses the backing array. Writing `copy[0]` **also** writes `ids[0]`. `slices.Clone` allocates a new array. Use it. Write a generic slice helper only when you have a desk-specific rule the standard library cannot express.

The boring rule

- Import `slices` (and `maps`, `cmp`) before you write a type parameter.
- Name constraints after what the body *does* (`comparable` for `==`, `cmp.Ordered` for `<`).
- Use `~T` for defined types with an underlying type you actually operate on.
- Let inference work. Write `[T]` when the call has no value to infer from.
- `any` means "I do not touch `T` except to store or return it." If you compare or order, say so in the constraint.

Try this

1. In `shift_ids.go`, replace the `Contains` prints with `slices.Index`. Print the index of `7` and of `9`.
2. Change `double` in `cents.go` to `T ~int | ~int64` and call it with `int64(150)`. Confirm `Cents` still works.
3. Delete `index` from `roster_index.go` and call `slices.Index` instead. The output should match.

Generic Types

Generic Types

A generic type is a type that takes type parameters: `Set[T]`, `List[E]`. The boring default for "a pile of unique IDs" is still `map[T]struct{}`. Wrap it in a named type when you have methods you want to keep next to the data, not because `Set` looks like computer science.

Mental model

`type Set[T comparable] map[T]struct{}` says: for any comparable `T`, a `Set[T]` is a map from `T` to empty struct. Instantiating `Set[string]` is a real type, distinct from `Set[int]`. Methods on `Set[T]` may use `T`. They may also declare **their own** type parameters (Go 1.27). That is for rare conversions. Most methods only need `T`.

Zero value: a nil map. `Has` on a nil `Set[T]` is fine (no keys). `Add` on a nil `Set[T]` panics. Provide `NewSet` or document "call Add only after make."

Worked examples

Case 1: The map is already a set

Open tables this shift. Membership is the whole API. A named generic type would be extra surface.

Save as `open_tables.go`:

```
// open_tables.go
package main

import "fmt"

func main() {
    open := map[int]struct{}{
        3:  {},
        7:  {},
        12: {},
    }
    _, seated := open[7]
    _, wait := open[4]
    fmt.Printf("table 7 open: %t\n", seated)
    fmt.Printf("table 4 open: %t\n", wait)
```

```
    delete(open, 7)
    _, seated = open[7]
    fmt.Printf("table 7 after close: %t\n", seated)
}
```

Run:

```
go run open_tables.go
```

Output:

```
table 7 open: true
table 4 open: false
table 7 after close: false
```

Stay here if the set never grows methods. The empty struct uses no extra memory per key.

Case 2: `Set[T comparable]` with methods

The desk now has two sets: names on the roster, and table numbers. Same Add / Has / Len. That is the second copy. Name the type.

Save as `set.go`:

```
// set.go
package main

import "fmt"

type Set[T comparable] map[T]struct{}

func NewSet[T comparable]() Set[T] {
    return make(Set[T])
}

func (s Set[T]) Add(v T) {
    s[v] = struct{}{}
}

func (s Set[T]) Has(v T) bool {
    _, ok := s[v]
    return ok
}

func (s Set[T]) Len() int {
    return len(s)
}
```

```
func main() {
    roster := NewSet[string]()
    roster.Add("Amina")
    roster.Add("Bo")
    roster.Add("Amina")

    tables := NewSet[int]()
    tables.Add(3)
    tables.Add(12)

    fmt.Printf("roster len=%d has Bo=%t\n", roster.Len(), roster.Has("Bo"))
    fmt.Printf("tables len=%d has 7=%t\n", tables.Len(), tables.Has(7))
}
```

Run:

```
go run set.go
```

Output:

```
roster len=2 has Bo=true
tables len=2 has 7=false
```

`Add` has a **value** receiver on a map type: the map header is copied, the backing hash table is shared, so inserts are visible to the caller. `NewSet` makes the map. Calling `var s Set[string]; s.Add("Amina")` panics — nil map. That is why `NewSet` exists.

Case 3: A small collection type, not a framework

Tickets on a rail: push, peek, length. Generic because tickets and shift names both queue. Still a struct and three methods.

Save as `rail.go`:

```
// rail.go
package main

import "fmt"

type Rail[T any] struct {
    items []T
}

func (r *Rail[T]) Push(v T) {
    r.items = append(r.items, v)
}
```

```
func (r *Rail[T]) Peek() (T, bool) {
    var zero T
    if len(r.items) == 0 {
        return zero, false
    }
    return r.items[0], true
}

func (r *Rail[T]) Len() int {
    return len(r.items)
}

func main() {
    var tickets Rail[int]
    tickets.Push(41)
    tickets.Push(42)
    id, ok := tickets.Peek()
    fmt.Printf("peek %d ok=%t len=%d\n", id, ok, tickets.Len())

    var names Rail[string]
    fmt.Printf("empty peek ok=%t\n", func() bool {
        _, ok := names.Peek()
        return ok
    }())
}
```

Run:

```
go run rail.go
```

Output:

```
peek 41 ok=true len=2
empty peek ok=false
```

Pointer receiver: `Push` must replace the slice header. `T any` because a rail does not compare elements. If you need `Has`, constrain `T` to `comparable` instead of adding a type assertion in the method.

Case 4: Methods that introduce a new type parameter (Go 1.27)

Before 1.27, a method could only use the receiver's `T`. Mapping a rail of IDs to labels had to be a package-level function. Now a method may declare `U`. Use it when the conversion belongs to the type. Do not build a query engine.

Save as `rail_labels.go`:

```
// rail_labels.go
package main

import "fmt"

type Rail[T any] struct {
    items []T
}

func (r *Rail[T]) Push(v T) {
    r.items = append(r.items, v)
}

func (r Rail[T]) Labels[U any](f func(T) U) []U {
    out := make([]U, len(r.items))
    for i, v := range r.items {
        out[i] = f(v)
    }
    return out
}

func main() {
    var tickets Rail[int]
    tickets.Push(41)
    tickets.Push(42)
    fmt.Println(tickets.Labels(func(id int) string {
        return fmt.Sprintf("T-%d", id)
    }))
}
```

Run:

```
go run rail_labels.go
```

Output:

```
[T-41 T-42]
```

`Labels` is a method so it reads as `tickets.Labels(...)`. A package function `labels(r Rail[int], f ...)` is equally boring and easier to find in docs. Prefer the function if you only have one conversion.

The trap

A generic type whose API is just the built-in underneath. This `Box[T]` is a field. It does not earn the name.

Save as `box.go`:

```
// box.go
package main

import "fmt"

type Box[T any] struct {
    V T
}

func (b Box[T]) Get() T { return b.V }

func main() {
    n := Box[int]{V: 12}
    fmt.Println(n.Get())
}
```

Run:

```
go run box.go
```

Output:

```
12
```

Use `int`. Wrap a type when you have **invariants or methods** (a set that ignores duplicates, a rail that peeks). A single field named `V` is not an invariant.

The boring rule

- `map[T]struct{}` first. `Set[T]` when Add/Has/Len show up in more than one file.
- Constrain the type parameter to what methods need (`comparable` for map keys).
- Nil maps: `Has` is safe, `Add` is not. Construct with `make` or `NewSet`.
- Methods that only use `T` are ordinary methods. Methods that introduce `U` are a Go 1.27 feature — keep them rare.
- Do not genericize a struct that always holds one concrete type at your desk.

Try this

1. Add `Remove(v T)` to `Set` in `set.go`. Delete `"Bo"` and print `Has("Bo")` again.
2. In `rail.go`, add `Pop() (T, bool)` that removes the front item. Pop twice from the ticket rail; the second should be `42`.
3. Replace `Labels` in `rail_labels.go` with a package-level function. Confirm the printed slice is unchanged.

Idiomatic Generics

Idiomatic Generics

Idiomatic generics are **short constraints, obvious instantiations, and no type puzzles**. If a reviewer needs a whiteboard to parse the signature, use an interface or a concrete type instead.

Mental model

Pick the tool from the job:

Job	Tool
Several types share methods; the caller only needs those methods	Interface
An algorithm or container must return the **same concrete type** it received	Type parameter
One type at every call site	That type, no generics, no interface

Constraints should read like English: `cmp.Ordered`, `comparable`, or a tiny interface you named (`interface{ ID() int }`). Union constraints (`int | int64 | float64`) are fine when the body uses operators those types share. Stacking three anonymous interfaces with `~` and method sets is how signatures rot.

Worked examples

Case 1: Interface for behavior, generic for the type you need back

Anything with a `Price() int` can go on a receipt. That is an interface. Finding the cheaper of two values of the **same** type is a generic (or the built-in `min`). Mixing them is the usual mistake: a generic `Pricer[T]` that only calls `Price()`.

Save as `receipt.go`:

```
// receipt.go
package main

import "fmt"

type Pricer interface {
    Price() int
}
```

```
type Ticket struct {
    ID    int
    Cents int
}

func (t Ticket) Price() int { return t.Cents }

type Drink struct {
    Name  string
    Cents int
}

func (d Drink) Price() int { return d.Cents }

func total(items []Pricer) int {
    sum := 0
    for _, it := range items {
        sum += it.Price()
    }
    return sum
}

func cheaper[T Pricer](a, b T) T {
    if a.Price() <= b.Price() {
        return a
    }
    return b
}

func main() {
    t := Ticket{ID: 41, Cents: 1250}
    d := Drink{Name: "tea", Cents: 300}
    fmt.Println("total", total([]Pricer{t, d}))

    a := Ticket{ID: 1, Cents: 900}
    b := Ticket{ID: 2, Cents: 400}
    fmt.Printf("cheaper ticket %d\n", cheaper(a, b).ID)
}
```

Run:

```
go run receipt.go
```

Output:

```
total 1550
cheaper ticket 2
```

`total` takes `[]Pricer` because the slice holds mixed types and you only need `Price()`. `cheaper` is generic because the result must still be a `Ticket` (so `.ID` type-checks). If `cheaper` took `Pricer`, the return would be an interface and `.ID` would not compile. That is the whole distinction.

Case 2: A constraint that stays readable

Name the constraint. Put the method set in one place. The function signatures stay short.

Save as `identified.go`:

```
// identified.go
package main

import "fmt"

type Identified interface {
    ID() int
}

type Ticket struct{ N int }

func (t Ticket) ID() int { return t.N }

type Shift struct{ N int }

func (s Shift) ID() int { return s.N }

func lookup[T Identified](list []T, id int) (T, bool) {
    var zero T
    for _, v := range list {
        if v.ID() == id {
            return v, true
        }
    }
    return zero, false
}

func main() {
    tickets := []Ticket{{41}, {42}, {43}}
    t, ok := lookup(tickets, 42)
    fmt.Printf("ticket %d ok=%t\n", t.N, ok)

    shifts := []Shift{{1}, {2}}
    s, ok := lookup(shifts, 9)
    fmt.Printf("shift %d ok=%t\n", s.N, ok)
}
```

Run:

```
go run identified.go
```

Output:

```
ticket 42 ok=true
shift 0 ok=false
```

`Identified` is three lines. Readers do not have to parse `T interface{ ID() int }` in every signature. Reuse the name in tests and docs.

Case 3: Let `slices` express the algorithm

Sorting tickets by cents does not need a generic desk type. It needs `slices.SortFunc`.

Save as `sort_tickets.go`:

```
// sort_tickets.go
package main

import (
    "fmt"
    "slices"
)

type Ticket struct {
    ID    int
    Cents int
}

func main() {
    tickets := []Ticket{
        {ID: 41, Cents: 1250},
        {ID: 42, Cents: 400},
        {ID: 43, Cents: 800},
    }
    slices.SortFunc(tickets, func(a, b Ticket) int {
        return a.Cents - b.Cents
    })
    for _, t := range tickets {
        fmt.Printf("%d:%d\n", t.ID, t.Cents)
    }
}
```

Run:

```
go run sort_tickets.go
```

Output:

```
42:400
43:800
41:1250
```

No `OrderedTicket` interface. No `Sortable[T]`. A function value and a standard-library generic. That is the idiomatic shape.

The trap

Type gymnastics: converting through `any` to "make it generic," or a constraint that lists every type you might ever meet.

Save as `gym.go`:

```
// gym.go
package main

import "fmt"

func convert[A, B any](a A) B {
    return any(a).(B)
}

func main() {
    n := convert[int, int](12)
    fmt.Println(n)
    defer func() {
        fmt.Println("panic:", recover())
    }()
    fmt.Println(convert[int, string](12))
}
```

Run:

```
go run gym.go
```

Output:

```
12
panic: interface conversion: interface {} is int, not string
```

The signature claims any `A` becomes any `B`. The body is a type assertion. You get a generic that panics — the worst of both worlds. If you know the types, write a function that takes them. If you do not, you cannot convert them.

A related smell is a constraint like `interface{ ~int | ~int8 | ~int16 | ~int32 | ~int64; String() string; Price() int }` invented so one function can do three jobs. Split the function.

The boring rule

- Interface when you need methods and mixed concrete types.
- Type parameter when you need the concrete type back, or a container of `T`.
- Name constraints. Keep them one idea.
- Prefer `slices`, `maps`, and `cmp` over a house `Filter` / `Map` / `Reduce` chain.
- Go 1.27 generic methods are allowed. A package-level function is still easier to search and test. Use a method when it is clearly about that type.
- If the body contains `any(v).(T)`, you are not writing generics. You are writing a type assertion with extra syntax.

Try this

1. In `receipt.go`, try `cheaper(t, d)` where `t` is a `Ticket` and `d` is a `Drink`. Read the error. That mixed pair belongs in `total`, not in `cheaper`.
2. Add `func ids[T Identified](list []T) []int` to `identified.go` and print the ticket IDs.
3. Delete `convert` from your brain. Write `func centsOf(t Ticket) int { return t.Cents }` instead whenever you know the type.

Generic Methods

Generic Methods

Go 1.18 gave you generic **functions** and generic **types**. Helpers that belonged with a type still had to live as package-level functions (`MapList`, `FilterTickets`, …), so call sites nested inside out and the package namespace filled up. **Go 1.27** lets a **concrete** method declare its own type parameters. Keep the helper on the type. Chain left to right. Do not put type parameters on interface methods — that is still impossible, on purpose.

Mental model

- A **generic method** looks like a method with a type-parameter list after the name: `func (List[E]) Map[R any](f func(E) R) List[R]`.
- The receiver's type parameters (`E`) and the method's type parameters (`R`) are both in scope in the signature and body.
- Instantiation works like generic functions: usually inferred at the call site, or written explicitly (`list.Map[string](...)`).
- A method **expression** turns a generic method into a function: `List[int].Map[string]`.
- **Interfaces cannot declare type parameters on methods.** A concrete generic method does **not** make the type implement an interface that has a same-named non-generic method. Methods that participate in interfaces stay non-generic.
- Prefer a package-level generic function when the operation is not really about one receiver. Prefer a generic method when the operation is clearly owned by that type and you want chaining.

Stdlib taste: `math/rand/v2` kept the old typed helpers and added `(*Rand).N[Int intType](n Int) Int` so one method covers every integer width.

Worked examples

Case 1: Map on a desk ticket list

A generic `List[E]` already parameterizes the element type. Mapping to another element type needs a **second** type parameter on the method — that is what Go 1.18 could not put on the receiver.

Save as `ticket_list.go`:

```
// ticket_list.go
package main
```

```
import "fmt"

type List[E any] struct {
    elem E
    next *List[E]
}

func NewList[E any](elems ...E) *List[E] {
    var head *List[E]
    for i := len(elems) - 1; i >= 0; i-- {
        head = &List[E]{elem: elems[i], next: head}
    }
    return head
}

func (l *List[E]) Map[R any](f func(E) R) *List[R] {
    if l == nil {
        return nil
    }
    return &List[R]{elem: f(l.elem), next: l.next.Map(f)}
}

func (l *List[E]) String() string {
    if l == nil {
        return "[]"
    }
    s := fmt.Sprintf("[%v", l.elem)
    for p := l.next; p != nil; p = p.next {
        s += fmt.Sprintf(" %v", p.elem)
    }
    return s + "]"
}

func main() {
    ids := NewList(41, 42, 43)
    labels := ids.Map(func(id int) string {
        return fmt.Sprintf("T-%d", id)
    })
    fmt.Println(ids)
    fmt.Println(labels)
}
```

Run:

```
go run ticket_list.go
```

Output:

```
[41 42 43]
[T-41 T-42 T-43]
```

`Map` stays next to `List`. The package does not need `MapList`. The call reads left to right.

Case 2: Chain transforms without nesting

Package-level `MapList` forces inside-out nesting. A method chain stays readable.

Save as `ticket_chain.go`:

```go
// ticket_chain.go
package main

import "fmt"

type List[E any] struct {
    elem E
    next *List[E]
}

func NewList[E any](elems ...E) *List[E] {
    var head *List[E]
    for i := len(elems) - 1; i >= 0; i-- {
        head = &List[E]{elem: elems[i], next: head}
    }
    return head
}

func (l *List[E]) Map[R any](f func(E) R) *List[R] {
    if l == nil {
        return nil
    }
    return &List[R]{elem: f(l.elem), next: l.next.Map(f)}
}

func (l *List[E]) String() string {
    if l == nil {
        return "[]"
    }
    s := fmt.Sprintf("[%v", l.elem)
    for p := l.next; p != nil; p = p.next {
        s += fmt.Sprintf(" %v", p.elem)
    }
    return s + "]"
}
```

```
func main() {
    // Cents on the desk → dollars as float64 → printed labels.
    out := NewList(1250, 400, 800).
        Map(func(cents int) float64 { return float64(cents) / 100 }).
        Map(func(dollars float64) string { return fmt.Sprintf("$%.2f", dollars) })
    fmt.Println(out)
}
```

Run:

```
go run ticket_chain.go
```

Output:

```
[$12.50 $4.00 $8.00]
```

If you still want the function shape, take a method expression: `f := (*List[int]).Map[float64]`, then call `f(list, transform)`. Instantiation is required before the expression is a plain function value.

Case 3: One random helper for every integer width

Before 1.27, a seeded `Rand` needed `IntN`, `Int32N`, `Int64N`, …. Go 1.27 adds a generic method on the value: `(*Rand).N[Int intType](n Int) Int`. Same method, every integer width.

Save as `desk_rand.go`:

```
// desk_rand.go
package main

import (
    "fmt"
    "math/rand/v2"
)

func main() {
    r := rand.New(rand.NewPCG(1, 2))
    // Argument type picks Int; result type matches.
    a := r.N(10)
    b := r.N(int32(10))
    c := r.N(int64(10))
    fmt.Printf("%T\n", a)
    fmt.Printf("%T\n", b)
    fmt.Printf("%T\n", c)
    // Values are deterministic for this seed; assert they stay in range.
    if a < 0 || a >= 10 || b < 0 || b >= 10 || c < 0 || c >= 10 {
        panic("out of range")
```

```
    }
    fmt.Println("in range")
}
```

Run:

```
go run desk_rand.go
```

Output:

```
int
int32
int64
in range
```

You still have the old typed methods. Prefer `N` when the call site already knows the integer type you want back.

Case 4: Explicit instantiation when inference cannot see the result

Sometimes the type parameter appears only in the result. Spell it out.

Save as `ticket_fold.go`:

```
// ticket_fold.go
package main

import "fmt"

type Bag[E any] []E

func (b Bag[E]) FirstOr[R any](f func(E) R, fallback R) R {
    if len(b) == 0 {
        return fallback
    }
    return f(b[0])
}

func main() {
    open := Bag[int]{41, 42}
    empty := Bag[int]{}

    fmt.Println(open.FirstOr(func(id int) string {
        return fmt.Sprintf("ticket-%d", id)
    }, "none"))

    // Fallback is string; f's result is string - R is inferred.
```

```
    fmt.Println(empty.FirstOr(func(id int) string {
        return fmt.Sprintf("ticket-%d", id)
    }, "none"))

    // Prefer an explicit type argument when the call gets noisy:
    fmt.Println(open.FirstOr[string](func(id int) string {
        return fmt.Sprintf("#%d", id)
    }, "none"))
}
```

Run:

```
go run ticket_fold.go
```

Output:

```
ticket-41
none
#41
```

The trap

Treat a generic concrete method as if it implemented an interface. It does not.

Save as `trap_iface.go`:

```
// trap_iface.go
package main

import "fmt"

type Closer interface {
    Close() error
}

type Gate struct{}

// Generic concrete method - legal on Gate, useless for Closer.
func (Gate) Close[T any]() error {
    var zero T
    _ = zero
    return nil
}

func main() {
    var c Closer
```

```
    // Gate does not implement Closer: Close is generic, Closer.Close is not.
    // Uncommenting the next line fails to compile:
    // c = Gate{}
    _ = c
    fmt.Println(Gate{}.Close[int]())
}
```

Run:

```
go run trap_iface.go
```

Output:

```
<nil>
```

If you need `Closer`, write `func (Gate) Close() error`. Keep generic helpers as separately named methods (`CloseWith[T any](...)`) that are not part of the interface set.

Second trap: inventing a generic method for a one-off that is clearer as a package-level function. Methods are for organization around a type, not for hiding every free function.

The boring rule

- Use a generic **method** when the operation belongs on that type and chaining or method-set locality helps readers.
- Use a generic **function** when the operation is shared across types or is easier to find at package scope.
- Instantiation is required (inferred or explicit) before you call the method or take a method expression.
- Never design an interface that needs type parameters on methods — the language will not grow that feature for free.
- Do not expect a generic method to satisfy a non-generic interface method of the same name.
- Mirror stdlib taste: keep old clear helpers if they are widely used; add a generic method when it collapses a family (`Rand.N`).

Try this

1. In `ticket_list.go`, add `func (l *List[E]) Filter(keep func(E) bool) *List[E]` (no new type parameter) and keep only even ticket IDs.
2. Rewrite Case 2 using only package-level `MapList` helpers. Compare the nested call to the method chain.
3. In `desk_rand.go`, assign `f := (*rand.Rand).N[int]` and call `f(r, 10)`. Confirm it matches `r.N(10)`.
4. Try to write `type M interface { Map[R any](func(int) R) }` and read the compiler error. That limitation is the feature boundary, not a temporary gap.

Errors

Errors as Values

Errors as Values

In Go an error is a **value**: a value that implements a one-method interface. Functions return it. Callers check it. Nothing unwinds the stack in secret. The boring default is `if err != nil { return fmt.Errorf("...", err) }` at every call that can fail.

Mental model

```
type error interface {
    Error() string
}
```

`nil` means success. A non-nil `error` means failure. The string from `Error()` is for humans and logs. Programs that need to **branch** on a failure use a sentinel (`var ErrClosed = errors.New(...)`) or a dedicated type (`TableError`), not string matching.

Check every error. Discarding one with `_` is how a closed desk looks open until Friday.

Worked examples

Case 1: Return `error`, check it

Opening a table fails when the number is not positive. `main` prints the error and exits 1. The success path is the path with no `err`.

Save as `open_table.go`:

```
// open_table.go
package main

import (
    "fmt"
    "os"
)

func openTable(n int) error {
    if n <= 0 {
        return fmt.Errorf("table %d: number must be positive", n)
    }
```

```
    fmt.Printf("opened table %d\n", n)
    return nil
}

func main() {
    if err := openTable(3); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    if err := openTable(0); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}
```

Run:

```
go run open_table.go
```

Output (stdout, then stderr, then exit 1):

```
opened table 3
table 0: number must be positive
```

`fmt.Errorf` builds an error from a format string. There is no exception type to catch. The `if err != nil` line is the entire control-flow story.

Case 2: A sentinel for a known condition

The desk closes at the end of the shift. Callers that care about *that* failure compare with `==` (or `errors.Is`, next chapter). The sentinel is a package-level `var`.

Save as `sentinel.go`:

```
// sentinel.go
package main

import (
    "errors"
    "fmt"
)

var ErrClosed = errors.New("desk closed")

func ring(open bool, table int) error {
    if !open {
        return ErrClosed
```

```
    }
    if table <= 0 {
        return fmt.Errorf("table %d: number must be positive", table)
    }
    fmt.Printf("rang table %d\n", table)
    return nil
}

func main() {
    err := ring(false, 7)
    if err == ErrClosed {
        fmt.Println("go home:", err)
        return
    }
    if err != nil {
        fmt.Println("other:", err)
    }
}
```

Run:

```
go run sentinel.go
```

Output:

```
go home: desk closed
```

Use a sentinel when the condition is **one value** the whole package shares: closed, not found, already paid. Do not invent a sentinel per table number.

Case 3: A dedicated type when the error carries data

A bad table number should include the number so the caller can log it, skip it, or show it on a screen. That is a struct, not a sentinel.

Save as `table_error.go`:

```
// table_error.go
package main

import "fmt"

type TableError struct {
    Table int
    Msg   string
}
```

```
func (e TableError) Error() string {
    return fmt.Sprintf("table %d: %s", e.Table, e.Msg)
}

func openTable(n int) error {
    if n <= 0 {
        return TableError{Table: n, Msg: "number must be positive"}
    }
    fmt.Printf("opened table %d\n", n)
    return nil
}

func main() {
    err := openTable(-2)
    te, ok := err.(TableError)
    if ok {
        fmt.Printf("bad table=%d (%s)\n", te.Table, te.Msg)
        return
    }
    fmt.Println(err)
}
```

Run:

```
go run table_error.go
```

Output:

```
bad table=-2 (number must be positive)
```

The type assertion `err.(TableError)` is acceptable in `main` of a tiny program. At work, prefer `errors.As` (next chapter) so wrapping still works. The point here: **data lives on the type**, not in a parsed string.

Case 4: Check every error, including the "cannot fail" ones

Writing a note to stdout can fail (pipe closed). The boring program checks it.

Save as `note.go`:

```
// note.go
package main

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

```
func main() {
    _, err := fmt.Println("desk is open")
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}
```

Run:

```
go run note.go
```

Output:

```
desk is open
```

In a CLI, checking `fmt.Println` is optional taste. In a server, checking `Write`, `Close`, and `Encode` is not optional. The rule is the same: if the signature returns `error`, you have a decision to make. The decision is almost never `_`.

The trap

Assigning the error to `_` because the happy path compiled.

Save as `ignored.go`:

```
// ignored.go
package main

import "fmt"

func openTable(n int) error {
    if n <= 0 {
        return fmt.Errorf("table %d: number must be positive", n)
    }
    fmt.Printf("opened table %d\n", n)
    return nil
}

func main() {
    _ = openTable(0)
    fmt.Println("still going")
}
```

Run:

```
go run ignored.go
```

Output:

```
still going
```

Table 0 never opened. The program claims progress. That is the bug. Delete the `_`. Handle the error or return it.

The boring rule

- Return `error` as the last result. Return `nil` on success.
- Check `err` at the call site. Do not panic for the caller's bad input.
- `errors.New("...")` or `fmt.Errorf("...")` for simple failures.
- A package-level sentinel for one shared condition.
- A dedicated type when the caller needs fields.
- Never match `err.Error()` strings in production code. Strings are for people.

Try this

1. In `open_table.go`, open tables `3`, `0`, and `11` in a loop. Print each error and keep going instead of `os.Exit`.
2. Add a second sentinel `ErrPaid` in `sentinel.go`. Return it when `table == 7`. Branch on both sentinels in `main`.
3. Change `TableError` to a pointer receiver on `Error()` and return `&TableError{...}`. Keep the assertion in sync (`*TableError`).

Wrapping and Inspecting Errors

Wrapping and Inspecting Errors

Wrapping adds context **without erasing** the original error. Inspecting walks that chain. The boring default is `fmt.Errorf("verb: %w", err)` on the way out, and `errors.Is` / `errors.As` on the way in. Do not parse `Error()` strings.

Mental model

- `%w` (wrap) stores the inner error. `%v` / `%s` only copy the string; `errors.Is` will not see the inner value.
- `errors.Unwrap(err)` returns the next inner error, or `nil`.
- `errors.Is(err, target)` walks the chain (and joined errors) and reports whether any equals `target` (or implements an `Is` method).
- `errors.As(err, &dst)` walks the chain and stores the first matching type in `dst`.
- `errors.Join(e1, e2, ...)` combines independent failures into one value. `Is` / `As` see each of them.

Wrap when you add a layer (open ticket → charge card → print receipt). Join when several things failed at once (two tables both refused).

Worked examples

Case 1: `%w` keeps the sentinel reachable

The inner failure is `ErrClosed`. The outer message names the ticket. `errors.Is` still finds `ErrClosed`.

Save as `wrap.go`:

```
// wrap.go
package main

import (
    "errors"
    "fmt"
)

var ErrClosed = errors.New("desk closed")
```

```
func charge(open bool) error {
    if !open {
        return ErrClosed
    }
    return nil
}

func pay(ticket int, open bool) error {
    if err := charge(open); err != nil {
        return fmt.Errorf("pay ticket %d: %w", ticket, err)
    }
    return nil
}

func main() {
    err := pay(41, false)
    fmt.Println(err)
    fmt.Println("closed:", errors.Is(err, ErrClosed))
    fmt.Println("unwrap:", errors.Unwrap(err))
}
```

Run:

```
go run wrap.go
```

Output:

```
pay ticket 41: desk closed
closed: true
unwrap: desk closed
```

The printed string is for logs. The `Is` check is for control flow. Keep both.

Case 2: `%v` looks the same and breaks `Is`

Same program, `%v` instead of `%w`. The log line is identical. The sentinel is gone.

Save as `nowrap.go`:

```
// nowrap.go
package main

import (
    "errors"
    "fmt"
)
```

```
var ErrClosed = errors.New("desk closed")

func pay(ticket int) error {
    return fmt.Errorf("pay ticket %d: %v", ticket, ErrClosed)
}

func main() {
    err := pay(41)
    fmt.Println(err)
    fmt.Println("closed:", errors.Is(err, ErrClosed))
    fmt.Println("unwrap:", errors.Unwrap(err))
}
```

Run:

```
go run nowrap.go
```

Output:

```
pay ticket 41: desk closed
closed: false
unwrap: <nil>
```

This is the bug that shows up a month later: a handler that is supposed to treat “desk closed” as non-fatal never fires. Use `%w` when the inner error is a real error value.

Case 3: `errors.As` for a dedicated type

Wrapping must not hide fields. `As` finds `TableError` through the outer `fmt.Errorf`.

Save as `as.go`:

```
// as.go
package main

import (
    "errors"
    "fmt"
)

type TableError struct {
    Table int
    Msg   string
}

func (e TableError) Error() string {
    return fmt.Sprintf("table %d: %s", e.Table, e.Msg)
```

```
}

func openTable(n int) error {
    if n <= 0 {
        return TableError{Table: n, Msg: "number must be positive"}
    }
    return nil
}

func seat(n int) error {
    if err := openTable(n); err != nil {
        return fmt.Errorf("seat: %w", err)
    }
    return nil
}

func main() {
    err := seat(-2)
    var te TableError
    if errors.As(err, &te) {
        fmt.Printf("cannot seat table %d (%s)\n", te.Table, te.Msg)
        return
    }
    fmt.Println(err)
}
```

Run:

```
go run as.go
```

Output:

```
cannot seat table -2 (number must be positive)
```

`As` needs a pointer to the destination. Pointer vs value must match what was returned (`TableError` here, not `*TableError`). If you return `&TableError{...}`, pass `**TableError` or use `var te *TableError; errors.As(err, &te)`.

Case 4: `errors.Join` for independent failures

Closing two tables: both can fail. Returning the first error hides the second. Join them.

Save as `join.go`:

```
// join.go
package main
```

```
import (
    "errors"
    "fmt"
)

var (
    ErrBusy  = errors.New("busy")
    ErrDirty = errors.New("dirty")
)

func closeTable(n int) error {
    switch n {
    case 3:
        return fmt.Errorf("table %d: %w", n, ErrBusy)
    case 7:
        return fmt.Errorf("table %d: %w", n, ErrDirty)
    default:
        fmt.Printf("closed table %d\n", n)
        return nil
    }
}

func closeAll(tables []int) error {
    var errs []error
    for _, n := range tables {
        if err := closeTable(n); err != nil {
            errs = append(errs, err)
        }
    }
    return errors.Join(errs...)
}

func main() {
    err := closeAll([]int{3, 4, 7})
    fmt.Println(err)
    fmt.Println("busy:", errors.Is(err, ErrBusy))
    fmt.Println("dirty:", errors.Is(err, ErrDirty))
}
```

Run:

```
go run join.go
```

Output:

```
closed table 4
table 3: busy
```

```
table 7: dirty
busy: true
dirty: true
```

`errors.Join` skips nils. Joining a single error returns that error. Joining nothing returns nil. `Is` sees through the join.

The trap

String matching on `err.Error()`. A wrap changes the string. Your `strings.Contains(err.Error(), "closed")` misses `pay ticket 41: desk closed` if someone rewords the inner message — or matches the wrong error that happens to contain the word.

Save as `by_string.go`:

```
// by_string.go
package main

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

var ErrClosed = errors.New("desk closed")

func main() {
    err := fmt.Errorf("pay ticket %d: %w", 41, ErrClosed)
    fmt.Println("contains closed:", strings.Contains(err.Error(), "closed"))

    other := errors.New("enclosure door closed")
    fmt.Println("false friend:", strings.Contains(other.Error(), "closed"))
    fmt.Println("Is ErrClosed:", errors.Is(other, ErrClosed))
}
```

Run:

```
go run by_string.go
```

Output:

```
contains closed: true
false friend: true
Is ErrClosed: false
```

`errors.Is` is the comparison. Strings are for logs.

The boring rule

- Wrap with `%w` when the caller might inspect the inner error.
- `fmt.Errorf("op: %w", err)` — operation first, inner error last.
- `errors.Is` for sentinels. `errors.As` for types with fields.
- `errors.Join` when failures are independent. Do not Join a single causal chain; wrap that.
- Do not compare `err.Error()` in `if` statements.
- Pointer vs value: return one, inspect the same one.

Try this

1. In `wrap.go`, change `%w` to `%v`. Confirm `Is` becomes false. Change it back.
2. Return `*TableError` from `openTable` in `as.go`. Fix the `As` destination so the program still prints `cannot seat table -2`.
3. In `join.go`, close only table `4`. Print `err == nil` (it should be true).

panic, recover, and Failure Modes

panic, recover, and Failure Modes

`panic` is for **broken invariants**: the program is wrong, not the user. `recover` is for a **process or goroutine boundary** that must not take the rest of the desk down with it. The boring default for a missing table, a bad request, or a closed card reader is `error`.

Mental model

- `panic(v)` stops the current goroutine, runs deferred functions, then crashes the process if nothing recovers.
- `recover()` is only useful **inside a deferred function**. It returns the panic value, or `nil` if there is no panic.
- Recovering in the middle of business logic hides bugs. Recovering in a worker so one bad ticket does not kill the shift is the intended use.

Failure modes at a desk:

Situation	Tool
Caller passed table `0`	`error`
JSON from the network is junk	`error`
A length is negative because your own code subtracted wrong	`panic` (or a test that should have caught it)
A worker goroutine should die without the process dying	`recover` at that goroutine's top `defer`

Worked examples

Case 1: A panic for a true invariant

A shift roster with zero names is not an input error in this program: `main` builds the roster. If it is empty, the author made a mistake. Panic is a loud crash.

Save as `invariant.go`:

```
// invariant.go
package main

import "fmt"
```

```
func mustLead(roster []string) string {
    if len(roster) == 0 {
        panic("empty shift roster")
    }
    return roster[0]
}

func main() {
    fmt.Println("lead:", mustLead([]string{"Amina", "Bo"}))
    fmt.Println("lead:", mustLead(nil))
}
```

Run:

```
go run invariant.go
```

Output (first lines; a stack trace follows, then the process exits 2):

```
lead: Amina
panic: empty shift roster
```

The first call is fine. The second is a bug. Do not `recover` here just to keep `main` pretty. Fix the roster.

Case 2: Recover at a worker boundary

Each ticket runs in its own worker. A panic in one worker must not kill the others. `recover` lives in a `defer` at the top of the worker, logs the value, and returns.

Save as `worker.go`:

```
// worker.go
package main

import (
    "fmt"
    "sync"
)

func handle(ticket int) {
    if ticket <= 0 {
        panic(fmt.Sprintf("ticket %d: id must be positive", ticket))
    }
    fmt.Printf("handled ticket %d\n", ticket)
}

func worker(ticket int, wg *sync.WaitGroup) {
```

```
    defer wg.Done()
    defer func() {
        if v := recover(); v != nil {
            fmt.Printf("worker recovered: %v\n", v)
        }
    }()
    handle(ticket)
}

func main() {
    var wg sync.WaitGroup
    for _, id := range []int{41, 0, 42} {
        wg.Add(1)
        go worker(id, &wg)
    }
    wg.Wait()
    fmt.Println("desk still open")
}
```

Run:

```
go run worker.go
```

Possible output (handled lines and the recovered line may interleave):

```
handled ticket 41
worker recovered: ticket 0: id must be positive
handled ticket 42
desk still open
```

The process exits 0. Ticket `0` is still a bug in whoever queued it — recovery is not forgiveness. It is isolation. After you recover, increment a metric or log at error level; do not pretend the ticket succeeded.

Case 3: Recover only in a defer

Calling `recover()` in ordinary line order always returns `nil`. This program shows the mistake, then the defer.

Save as `recover_place.go`:

```
// recover_place.go
package main

import "fmt"

func wrong() {
```

```
    defer func() {
        fmt.Println("wrong recovered:", recover())
    }()
    v := recover()
    fmt.Println("inline recover:", v)
    panic("rail jammed")
}

func main() {
    wrong()
    fmt.Println("after wrong")
}
```

Run:

```
go run recover_place.go
```

Output:

```
inline recover: <nil>
wrong recovered: rail jammed
after wrong
```

The inline `recover` did nothing (no panic was in flight). The deferred `recover` caught `rail jammed` and `main` continued. That is why every real recover is a `defer`.

Case 4: Recover at an HTTP middleware boundary

Save as `server_recover.go`. Web handlers run concurrently. If one request handler panics due to unexpected input or a nil pointer, the server should log the panic, return an HTTP 500 error to the client, and continue serving other clients normally.

```
// server_recover.go
package main

import (
    "fmt"
    "net/http"
    "net/http/httptest"
)

func recoveryMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        defer func() {
            if err := recover(); err != nil {
                http.Error(w, "internal desk error", http.StatusInternalServerError)
```

```
                fmt.Printf("recovered request panic: %v\n", err)
            }
        }()
        next.ServeHTTP(w, r)
    })
}

func buggyHandler(w http.ResponseWriter, r *http.Request) {
    panic("unexpected nil pointer in order processor")
}

func main() {
    handler := recoveryMiddleware(http.HandlerFunc(buggyHandler))

    req := httptest.NewRequest("GET", "/order", nil)
    rec := httptest.NewRecorder()

    handler.ServeHTTP(rec, req)
    fmt.Printf("HTTP status: %d\n", rec.Code)
    fmt.Printf("HTTP body: %s", rec.Body.String())
}
```

Run:

```
go run server_recover.go
```

Output:

```
recovered request panic: unexpected nil pointer in order processor
HTTP status: 500
HTTP body: internal desk error
```

The middleware cleanly caught the handler panic, prevented a process crash, and responded with standard HTTP status code 500.

The trap

Trap 1: Using panic as a substitute for error

Using panic as a substitute for `error`, then recovering in the same function so it "looks like exceptions."

Save as `fake_exceptions.go`:

```
// fake_exceptions.go
package main
```

```
import "fmt"

func openTable(n int) (err error) {
    defer func() {
        if v := recover(); v != nil {
            err = fmt.Errorf("%v", v)
        }
    }()
    if n <= 0 {
        panic(fmt.Sprintf("table %d: number must be positive", n))
    }
    fmt.Printf("opened table %d\n", n)
    return nil
}

func main() {
    fmt.Println(openTable(0))
    fmt.Println(openTable(3))
}
```

Run:

```
go run fake_exceptions.go
```

Output:

```
table 0: number must be positive
opened table 3
<nil>
```

It works. It is also slower, harder to read, and hostile to `errors.Is`. Write `return fmt.Errorf(...)`. Save panic for "this cannot happen if the program is correct."

Trap 2: Panics do not cross goroutine boundaries

A `defer recover()` in `main` or in an HTTP handler **cannot catch a panic that happens inside a spawned goroutine**. Each goroutine maintains its own independent call stack. If a background goroutine panics without its own deferred recover, the entire process crashes immediately.

Save as `goroutine_boundary.go`:

```
// goroutine_boundary.go
package main

import (
    "fmt"
```

```
    "time"
)

func main() {
    // Deferred recover in main
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("recovered in main:", r)
        }
    }()

    // Spawned background worker with its own recovery boundary
    go func() {
        defer func() {
            if r := recover(); r != nil {
                fmt.Println("recovered inside worker:", r)
            }
        }()
        panic("hardware sensor failed")
    }()

    time.Sleep(50 * time.Millisecond)
    fmt.Println("desk server still running")
}
```

Run:

```
go run goroutine_boundary.go
```

Output:

```
recovered inside worker: hardware sensor failed
desk server still running
```

If you remove the `defer func() { recover() }` from inside the goroutine, `recovered in main` is never called. The process exits with code 2. Every background goroutine you launch must own its own error or recovery boundary.

The boring rule

- User input, network, disk, "not found": `error`.
- Impossible state in *your* code: `panic`, or a test that fails before production.
- `recover` only at a goroutine or process boundary (worker, HTTP server middleware, plugin).
- Panics do not bubble across goroutines. Every spawned goroutine that could panic must have its own recovery handler.
- After recover, log the value. Do not continue as if the work succeeded.

- `recover()` belongs in a `defer` func. Anywhere else it is a no-op.
- Do not build an exception system out of panic.

Try this

1. In `invariant.go`, stop calling `mustLead(nil)`. The program should print one line and exit 0.
2. In `worker.go`, panic on ticket `42` as well. Confirm you still see `desk still open` and two recovered lines.
3. In `goroutine_boundary.go`, comment out the worker's inner `defer` block and run again. Notice that `recovered in main` does not catch it and the program crashes.
4. Replace `panic` in `fake_exceptions.go` with `return fmt.Errorf(...)` and delete the `defer`. Behavior of `main` stays the same; the function gets honest.

Testing

Testing Fundamentals

Testing Fundamentals

Tests in Go are ordinary functions in files named `*_test.go`. You run them with `go test`. The boring default is a **table-driven** test: a slice of cases, `t.Run` per case, `t.Fatalf` on the first mismatch. No framework.

Mental model

- Production code lives in `desk.go` (`package desk`).
- Tests live in `desk_test.go` (`package desk` to see unexported names, or `package desk_test` to see only the public API — next chapter).
- `func TestXxx(t *testing.T)` is a test. The name after `Test` is capital.
- `t.Fatalf` fails and **stops that test function** (or that `t.Run` subtest). `t.Errorf` fails and continues.
- `t.Helper()` marks a function so failure line numbers point at the caller, not the helper.

Put both files in an **empty directory**, then:

```
go mod init desk
go test
```

Each listing below is a complete file. Copy the pair that the case names.

Worked examples

Case 1: A package and a table-driven test

`Total` adds ticket prices in cents. The test names each case, runs it as a subtest, and fatals with `got` / `want`.

Save as `desk.go`:

```
// desk.go
package desk

func Total(prices []int) int {
    sum := 0
    for _, p := range prices {
        sum += p
    }
```

```
    return sum
}
```

Save as `desk_test.go`:

```
// desk_test.go
package desk

import "testing"

func TestTotal(t *testing.T) {
    tests := []struct {
        name   string
        prices []int
        want   int
    }{
        {name: "empty", prices: nil, want: 0},
        {name: "one ticket", prices: []int{450}, want: 450},
        {name: "three tickets", prices: []int{450, 800, 250}, want: 1500},
    }
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            got := Total(tt.prices)
            if got != tt.want {
                t.Fatalf("Total() = %d, want %d", got, tt.want)
            }
        })
    }
}
```

Run (from the directory that contains both files):

```
go mod init desk
go test
```

Output (the duration varies):

```
PASS
ok      desk    0.002s
```

`t.Run` gives you `--- FAIL: TestTotal/three_tickets` when something breaks. Without it, you only know that `TestTotal` failed.

If `go.mod` already exists, skip `go mod init`. `go test` compiles the package plus the tests; it does not run `main`. There is no `package main` here.

Case 2: `t.Helper` so line numbers tell the truth

An assertion helper without `t.Helper()` reports its own line. With `t.Helper()`, `go test` points at the call in the test.

Save as `shift.go`:

```
// shift.go
package desk

func Lead(roster []string) (string, bool) {
    if len(roster) == 0 {
        return "", false
    }
    return roster[0], true
}
```

Save as `shift_test.go` (same module, same directory as `desk.go` from Case 1, or a fresh `desk` module that contains only these two files):

```
// shift_test.go
package desk

import "testing"

func assertLead(t *testing.T, roster []string, want string, wantOK bool) {
    t.Helper()
    got, ok := Lead(roster)
    if ok != wantOK || got != want {
        t.Fatalf("Lead(%q) = %q, %t; want %q, %t", roster, got, ok, want, wantOK)
    }
}

func TestLead(t *testing.T) {
    assertLead(t, []string{"Amina", "Bo"}, "Amina", true)
    assertLead(t, nil, "", false)
}
```

Run:

```
go test
```

Output (the duration varies):

```
PASS
ok      desk    0.002s
```

Delete `t.Helper()` and fail a case on purpose: the file:line in the failure will be inside `assertLead`, not `TestLead`. Put `t.Helper()` back.

Case 3: `t.Fatalf` vs `t.Errorf`

`Fatalf` is for "this case is done." `Errorf` is for "collect more mismatches." In a table with `t.Run`, **Fatalf is the default**: one assertion, stop, next subtest still runs.

Save as `clamp.go`:

```
// clamp.go
package desk

func Clamp(n, lo, hi int) int {
    if n < lo {
        return lo
    }
    if n > hi {
        return hi
    }
    return n
}
```

Save as `clamp_test.go`:

```
// clamp_test.go
package desk

import "testing"

func TestClamp(t *testing.T) {
    tests := []struct {
        n, lo, hi, want int
    }{
        {n: 5, lo: 1, hi: 10, want: 5},
        {n: 0, lo: 1, hi: 10, want: 1},
        {n: 99, lo: 1, hi: 10, want: 10},
    }
    for _, tt := range tests {
        t.Run("", func(t *testing.T) {
            got := Clamp(tt.n, tt.lo, tt.hi)
            if got != tt.want {
                t.Fatalf("Clamp(%d,%d,%d) = %d, want %d", tt.n, tt.lo, tt.hi, got, tt.want)
            }
        })
    }
}
```

Run:

```
go test
```

Output (the duration varies):

```
PASS
ok      desk    0.002s
```

Empty subtest names are legal and ugly. Prefer `name` fields as in Case 1. `t.Run("", ...)` is here only to keep the table loop's fatals from skipping later rows.

The trap

A test that hard-codes one happy path and uses `panic` when it fails. You lose `go test`'s reporting, parallel subtests, and the habit of tables.

Save as `trap_test.go` (with `desk.go` from Case 1):

```
// trap_test.go
package desk

import "testing"

func TestTotalHappyOnly(t *testing.T) {
    if Total([]int{450, 800, 250}) != 1500 {
        panic("total is wrong")
    }
}
```

Run:

```
go test
```

It passes today. Tomorrow `Total` returns `1501` and you get a panic stack instead of `TestTotalHappyOnly: Total() = 1501, want 1500`. Write the table. Call `t.Fatalf`. Cover empty input.

The boring rule

- One module directory. `go mod init desk`. `go test`.
- `TestXxx(t *testing.T)` in `*_test.go`.
- Tables: `name`, inputs, `want`. Loop with `t.Run(tt.name, ...)`.
- `t.Fatalf` for a broken case. `t.Helper()` on helpers.
- Assert `got` and `want` in the message. Do not panic in tests.
- Tests that compile but never fail are souvenirs, not tests. Include an empty or error case.

Try this

1. Add a case `negative` to `TestTotal` with `prices: []int{-100, 250}` and `want: 150`. Run `go test`.
2. In `TestLead`, add a case for a single-name roster. Use `assertLead`.
3. Break `Clamp` so `n > hi` returns `hi - 1`. Run `go test` and read the `FAIL` line. Restore it.

Test Organization and Coverage

Test Organization and Coverage

Keep tests next to the code they cover. Use `package desk` when you must see unexported functions. Use `package desk_test` when you want to test the **public** API like a caller. Put fixtures in `testdata/`. Measure with `go test -cover`. The boring default is external tests for the API plus a few internal tests for the awkward bits.

Mental model

Same directory:

File	Package clause	Sees unexported names?
`desk.go`	`package desk`	— (production)
`desk_test.go`	`package desk`	yes (white-box / internal test)
`api_test.go`	`package desk_test`	no (black-box / external test)

Go compiles `package desk_test` as a separate package that **imports** `desk`. That is the external test package. Naming the file `desk_internal_test.go` does not change that — the **package clause** does. Prefer `package desk_test` and a filename that says what it tests.

`testdata/` is ignored as a Go package. It is a folder of files your tests read.

Coverage is the fraction of statements executed. 100% is not a goal. Uncovered error branches you care about are.

Worked examples

Case 1: Internal test (`package desk`)

`normalize` is unexported. Only an internal test can call it directly.

In an empty directory:

```
go mod init desk
```

Save as `desk.go`:

```
// desk.go
package desk
```

```
import "strings"

func normalize(name string) string {
    return strings.TrimSpace(strings.ToLower(name))
}

func RosterHas(roster []string, name string) bool {
    want := normalize(name)
    for _, n := range roster {
        if normalize(n) == want {
            return true
        }
    }
    return false
}
```

Save as `desk_test.go`:

```
// desk_test.go
package desk

import "testing"

func TestNormalize(t *testing.T) {
    tests := []struct {
        in, want string
    }{
        {in: "Amina", want: "amina"},
        {in: "  Bo  ", want: "bo"},
        {in: "", want: ""},
    }
    for _, tt := range tests {
        t.Run(tt.in, func(t *testing.T) {
            got := normalize(tt.in)
            if got != tt.want {
                t.Fatalf("normalize(%q) = %q, want %q", tt.in, got, tt.want)
            }
        })
    }
}
```

Run:

```
go test
```

Output (the duration varies):

```
PASS
```

```
ok      desk    0.002s
```

Case 2: External test (`package desk_test`)

This file cannot mention `normalize`. It imports the module and uses `RosterHas`. That is the API a waiter would call.

Save as `api_test.go`:

```
// api_test.go
package desk_test

import (
    "testing"

    "desk"
)

func TestRosterHas(t *testing.T) {
    roster := []string{"Amina", "Bo"}
    tests := []struct {
        name string
        want bool
    }{
        {name: "amina", want: true},
        {name: "  BO ", want: true},
        {name: "Chen", want: false},
    }
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            got := desk.RosterHas(roster, tt.name)
            if got != tt.want {
                t.Fatalf("RosterHas(%q) = %t, want %t", tt.name, got, tt.want)
            }
        })
    }
}
```

Run:

```
go test
```

Output (the duration varies):

```
PASS
ok      desk    0.002s
```

The import path `"desk"` matches `go mod init desk`. In a real repo it would be the module path (`github.com/you/desk`). Both test packages run in one `go test`.

Case 3: `testdata/` fixtures and `-cover`

A menu file the test must not invent inline — it is a fixture, so it lives in `testdata/`.

Save as `testdata/menu.txt` (plain text, not Go):

```
soup 400
tea 300
pie 800
```

Save as `menu.go`:

```
// menu.go
package desk

import (
    "bufio"
    "fmt"
    "os"
    "strings"
)

func SumMenuFile(path string) (int, error) {
    f, err := os.Open(path)
    if err != nil {
        return 0, err
    }
    defer f.Close()

    sum := 0
    sc := bufio.NewScanner(f)
    for sc.Scan() {
        line := strings.TrimSpace(sc.Text())
        if line == "" {
            continue
        }
        var name string
        var cents int
        if _, err := fmt.Sscanf(line, "%s %d", &name, &cents); err != nil {
            return 0, fmt.Errorf("%s: %w", line, err)
        }
        sum += cents
    }
    if err := sc.Err(); err != nil {
```

```
            return 0, err
        }
        return sum, nil
    }
```

Save as `menu_test.go`:

```
    // menu_test.go
    package desk

    import "testing"

    func TestSumMenuFile(t *testing.T) {
        got, err := SumMenuFile("testdata/menu.txt")
        if err != nil {
            t.Fatalf("SumMenuFile: %v", err)
        }
        if got != 1500 {
            t.Fatalf("SumMenuFile() = %d, want 1500", got)
        }
    }
```

Run:

```
    go test -cover
```

Output (the duration varies; coverage is for this case's files alone):

```
PASS
coverage: 81.8% of statements
ok      desk    0.002s  coverage: 81.8% of statements
```

`go test` sets the working directory to the package directory, so `"testdata/menu.txt"` resolves. Do not put fixtures in `/tmp` and hope CI agrees.

If you only have the files from this case in the directory, coverage of `menu.go` is high; the error returns for a missing file are still untested. That is fine. Add a case with a bad path when you care.

The trap

Testing unexported details **and** nothing else. When `normalize` is rewritten as a one-liner inside `RosterHas`, a pile of internal tests go red and no test still describes waiter-facing behavior.

The fix is Case 2: at least one external test per exported function. Keep internal tests for parsers and bit-twiddling that are painful through the public API.

A second trap: chasing 100% coverage by testing `fmt.Println`. Coverage is a flashlight, not a score.

The boring rule

- Default: `package desk_test` against exported names.
- `package desk` only for unexported helpers that are actually subtle.
- Fixtures in `testdata/`. Never generate golden files in the package root without a good reason.
- `go test -cover` to see holes. Write a test for a hole you care about, not for a number.
- One module path. External tests import that path.

Try this

1. From `package desk_test`, try to call `normalize`. Read the compiler error. That is the boundary working.
2. Add `TestSumMenuFileMissing` that calls `SumMenuFile("testdata/nope.txt")` and fatals if `err == nil`.
3. Run `go test -coverprofile=cover.out` then `go tool cover -func=cover.out`. Read which functions are under 100%. Do not "fix" them unless a branch matters.

Advanced Testing

Advanced Testing

Tables cover the cases you thought of. **Fuzzing** throws bytes you did not. **Benchmarks** time a function. `httptest` gives a `ResponseRecorder` so a handler can be tested without a TCP port. The boring default is still a table. Add these when the function parses input, sits on a hot path, or is an HTTP handler.

Mental model

- `func FuzzXxx(f *testing.F)` — seed with `f.Add`, then `f.Fuzz`. `go test` **already runs the seeds** as ordinary tests. `go test -fuzz=FuzzXxx` keeps generating values until you stop it (or `-fuzztime` elapses).
- `func BenchmarkXxx(b *testing.B)` — loop with `for b.Loop()` (Go 1.24+; this book is on 1.27). Do not use `b.N` in new code.
- `httptest.NewRequest` + `httptest.NewRecorder` — in-memory handler test. A full server is the next chapter.

Fuzz targets must be deterministic and must not fail on "bad" input if the function is supposed to return an error for junk. Assert **invariants**: if `err == nil`, the value is in range.

Worked examples

Case 1: A parser, a table test, and a fuzz target

`ParseTable` accepts `"12"` and rejects junk. Seeds run under `go test`. Fuzzing is optional extra.

Empty directory:

```
go mod init desk
```

Save as `table.go`:

```
// table.go
package desk

import (
    "fmt"
    "strconv"
)
```

```
func ParseTable(s string) (int, error) {
    n, err := strconv.Atoi(s)
    if err != nil {
        return 0, err
    }
    if n <= 0 {
        return 0, fmt.Errorf("table %d: number must be positive", n)
    }
    return n, nil
}
```

Save as `table_test.go`:

```
// table_test.go
package desk

import "testing"

func TestParseTable(t *testing.T) {
    tests := []struct {
        in      string
        want    int
        wantErr bool
    }{
        {in: "12", want: 12},
        {in: "0", wantErr: true},
        {in: "soup", wantErr: true},
    }
    for _, tt := range tests {
        t.Run(tt.in, func(t *testing.T) {
            got, err := ParseTable(tt.in)
            if tt.wantErr {
                if err == nil {
                    t.Fatalf("ParseTable(%q) = %d, nil; want error", tt.in, got)
                }
                return
            }
            if err != nil || got != tt.want {
                t.Fatalf("ParseTable(%q) = %d, %v; want %d, nil", tt.in, got, err, tt.want)
            }
        })
    }
}

func FuzzParseTable(f *testing.F) {
    f.Add("12")
    f.Add("1")
```

```
    f.Add("0")
    f.Fuzz(func(t *testing.T, s string) {
        n, err := ParseTable(s)
        if err != nil {
            return
        }
        if n <= 0 {
            t.Fatalf("ParseTable(%q) = %d, nil; invariant n > 0", s, n)
        }
    })
}
```

Run (seeds only — no long fuzz):

```
go test
```

Output (the duration varies):

```
PASS
ok      desk    0.003s
```

Optional, bounded fuzz:

```
go test -fuzz=FuzzParseTable -fuzztime=2s
```

Output looks like this (counts vary):

```
fuzz: elapsed: 0s, gathering baseline coverage: 0/3 completed
fuzz: elapsed: 2s, execs: 20000 (10000/sec), new interesting: 4 (total: 7)
PASS
ok      desk    2.050s
```

If fuzzing finds a crash, it writes a file under `testdata/fuzz/`. Keep that file; `go test` will replay it. Do not assert `err != nil` for every random string — `ParseTable("8")` is valid and would fail the fuzz.

Case 2: A benchmark with `b.Loop`

`Total` from a slice of cents. Setup sits **outside** the loop so you do not measure slice allocation.

Save as `total.go`:

```
// total.go
package desk

func Total(prices []int) int {
```

```
    sum := 0
    for _, p := range prices {
        sum += p
    }
    return sum
}
```

Save as `total_test.go`:

```
// total_test.go
package desk

import "testing"

func BenchmarkTotal(b *testing.B) {
    prices := []int{450, 800, 250, 1250, 300}
    for b.Loop() {
        if Total(prices) != 3050 {
            b.Fatal("unexpected total")
        }
    }
}
```

Run:

```
go test -bench=BenchmarkTotal -benchmem
```

Output (ns/op and bytes/op vary by machine):

```
goos: linux
goarch: amd64
pkg: desk
BenchmarkTotal-16        80000000           14.20 ns/op        0 B/op          0 allocs/op
PASS
ok      desk    1.150s
```

`b.Loop` excludes setup, keeps the compiler from deleting the call, and is the 1.27 default. Checking the result inside the loop is allowed and prevents the compiler from treating `Total` as dead. For a tighter bench, assign to a package-level `var sink int` instead of `b.Fatal`.

Case 3: `httptest` recorder for a handler

No listen socket. A fake request in, a recorded response out.

Save as `menu_http.go`:

```
// menu_http.go
package desk

import (
    "net/http"
)

func Menu(w http.ResponseWriter, r *http.Request) {
    if r.URL.Path != "/menu" {
        http.NotFound(w, r)
        return
    }
    w.Header().Set("Content-Type", "text/plain")
    _, _ = w.Write([]byte("soup 400\n"))
}
```

Save as `menu_http_test.go`:

```
// menu_http_test.go
package desk

import (
    "io"
    "net/http"
    "net/http/httptest"
    "testing"
)

func TestMenu(t *testing.T) {
    t.Run("ok", func(t *testing.T) {
        rec := httptest.NewRecorder()
        req := httptest.NewRequest(http.MethodGet, "/menu", nil)
        Menu(rec, req)
        res := rec.Result()
        defer res.Body.Close()
        if res.StatusCode != http.StatusOK {
            t.Fatalf("status %d, want %d", res.StatusCode, http.StatusOK)
        }
        body, err := io.ReadAll(res.Body)
        if err != nil {
            t.Fatalf("read: %v", err)
        }
        if string(body) != "soup 400\n" {
            t.Fatalf("body %q", body)
        }
    })
    t.Run("missing", func(t *testing.T) {
```

```
        rec := httptest.NewRecorder()
        req := httptest.NewRequest(http.MethodGet, "/nope", nil)
        Menu(rec, req)
        if rec.Code != http.StatusNotFound {
            t.Fatalf("status %d, want %d", rec.Code, http.StatusNotFound)
        }
    })
}
```

Run:

```
go test
```

Output (the duration varies):

```
PASS
ok      desk    0.003s
```

The `_ = w.Write` in the handler is the one place this book still discards an error: `ResponseWriter.Write` on a recorder does not fail. On a real `ResponseWriter`, check it or wrap the handler. The next chapter uses a real test server.

The trap

A fuzz function that fatals on every error. Random strings are supposed to fail `Atoi`. The fuzz then "finds" thousands of bugs that are the happy error path.

Save as `badfuzz_test.go` only if you want to see it go red immediately — do not keep it:

```
// badfuzz_test.go
package desk

import "testing"

func FuzzParseTableWrong(f *testing.F) {
    f.Add("nope")
    f.Fuzz(func(t *testing.T, s string) {
        if _, err := ParseTable(s); err != nil {
            t.Fatalf("unexpected error for %q: %v", s, err)
        }
    })
}
```

`go test` runs the seed `"nope"`, `ParseTable` returns an error, the seed fails. That is not a parser bug. Assert invariants on the **success** path.

The boring rule

- Write a table first. Fuzz the parser after the table is green.
- Fuzz invariants (`err == nil  n > 0`), not "every string parses."
- `go test` runs seeds. Use `-fuzz` when you want generation. Cap it with `-fuzztime`.
- Benchmarks: `for b.Loop() { ... }`. Setup outside. `go test -bench=. -benchmem`.
- Handlers: `NewRequest` + `NewRecorder` before you spin a server.
- Check errors in tests. `b.Fatal` / `t.Fatal` exist for a reason.

Try this

1. Add a seed `f.Add("-3")` to `FuzzParseTable`. `go test` should still pass (`-3` is an error path).
2. Run `go test -bench=BenchmarkTotal -count=5` and look at the spread in ns/op. Do not treat one run as truth.
3. In `TestMenu`, add a subtest `POST /menu` and decide whether your handler should 405. Implement that decision.

Integration and System Testing

Integration and System Testing

An integration test runs **several real pieces** at once: an HTTP server plus a client, or a clock plus opening hours. It still lives in `_test.go`. The boring defaults: `httptest.NewServer` for HTTP, a **fake clock** for time, and a **build tag** so slow tests do not run on every `go test`.

Mental model

- Unit test: one function, fakes at the edge if needed.
- Integration test: two or more collaborating packages or a real HTTP stack. Still no production database unless you meant to pay for it.
- `httptest.NewServer(h)` listens on `127.0.0.1:0`, serves `h`, returns a URL. `defer ts.Close()`.
- Time in tests: do not `time.Sleep` and hope. Pass a `Clock` interface. Production uses `time.Now`. Tests pass a struct whose `Now()` returns Tuesday 09:00.
- `//go:build integration` on a file. `go test` skips it. `go test -tags=integration` includes it.

Worked examples

Case 1: A real test server

The handler is the same idea as the last chapter. The test speaks **HTTP over a loopback port**, which catches mistakes `ResponseRecorder` will not (redirects, client headers).

Empty directory:

```
go mod init desk
```

Save as `server.go`:

```
// server.go
package desk

import (
    "fmt"
    "net/http"
)
```

```
func NewMux() http.Handler {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /menu", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprint(w, "soup 400\n")
    })
    mux.HandleFunc("GET /health", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprint(w, "ok")
    })
    return mux
}
```

Save as `server_test.go`:

```
// server_test.go
package desk

import (
    "io"
    "net/http"
    "net/http/httptest"
    "testing"
)

func TestMenuServer(t *testing.T) {
    ts := httptest.NewServer(NewMux())
    defer ts.Close()

    res, err := http.Get(ts.URL + "/menu")
    if err != nil {
        t.Fatalf("GET /menu: %v", err)
    }
    defer res.Body.Close()
    body, err := io.ReadAll(res.Body)
    if err != nil {
        t.Fatalf("read: %v", err)
    }
    if res.StatusCode != http.StatusOK {
        t.Fatalf("status %d", res.StatusCode)
    }
    if string(body) != "soup 400\n" {
        t.Fatalf("body %q", body)
    }
}
```

Run:

```
go test
```

Output (the duration varies):

```
PASS
ok      desk    0.004s
```

`GET /menu` in `HandleFunc` is Go's method-aware mux (1.22+). A GET to `/health` is a second route; add a test when you care. Always `Close` the server and the body.

Case 2: A fake clock

The desk is open from 08:00 to 16:00 local. Tests must not depend on the wall clock.

Save as `hours.go`:

```
// hours.go
package desk

import "time"

type Clock interface {
    Now() time.Time
}

type realClock struct{}

func (realClock) Now() time.Time { return time.Now() }

func IsOpen(c Clock, day time.Time) bool {
    now := c.Now()
    open := time.Date(day.Year(), day.Month(), day.Day(), 8, 0, 0, 0, day.Location())
    close := time.Date(day.Year(), day.Month(), day.Day(), 16, 0, 0, 0, day.Location())
    return !now.Before(open) && now.Before(close)
}

func OpenNow() bool {
    n := time.Now()
    return IsOpen(realClock{}, n)
}
```

Save as `hours_test.go`:

```
// hours_test.go
package desk

import (
    "testing"
    "time"
```

```
)

type fakeClock struct{ t time.Time }

func (f fakeClock) Now() time.Time { return f.t }

func TestIsOpen(t *testing.T) {
    loc := time.UTC
    day := time.Date(2026, 3, 10, 0, 0, 0, 0, loc)
    tests := []struct {
        name string
        at   time.Time
        want bool
    }{
        {name: "morning", at: time.Date(2026, 3, 10, 9, 0, 0, 0, loc), want: true},
        {name: "just before close", at: time.Date(2026, 3, 10, 15, 59, 0, 0, loc), want: tru
        {name: "at close", at: time.Date(2026, 3, 10, 16, 0, 0, 0, loc), want: false},
        {name: "dawn", at: time.Date(2026, 3, 10, 7, 59, 0, 0, loc), want: false},
    }
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            got := IsOpen(fakeClock{t: tt.at}, day)
            if got != tt.want {
                t.Fatalf("IsOpen(%s) = %t, want %t", tt.at.Format(time.Kitchen), got, tt.wan
            }
        })
    }
}
```

Run:

```
go test
```

Output (the duration varies):

```
PASS
ok      desk    0.003s
```

`OpenNow` is untested on purpose: it touches the real clock. Production `main` calls `OpenNow`. Tests call `IsOpen` with `fakeClock`. Do not sleep until 16:00.

Case 3: Build tags so integration tests stay off the default path

A test that hits the server **and** checks `/health` is slower and more coupled. Tag it. Default `go test` stays fast.

Save as `health_integration_test.go`:

```
// health_integration_test.go
//go:build integration

package desk

import (
    "io"
    "net/http"
    "net/http/httptest"
    "testing"
)

func TestHealthIntegration(t *testing.T) {
    ts := httptest.NewServer(NewMux())
    defer ts.Close()

    res, err := http.Get(ts.URL + "/health")
    if err != nil {
        t.Fatalf("GET /health: %v", err)
    }
    defer res.Body.Close()
    body, err := io.ReadAll(res.Body)
    if err != nil {
        t.Fatalf("read: %v", err)
    }
    if string(body) != "ok" {
        t.Fatalf("body %q", body)
    }
}
```

Run without the tag (Cases 1–2 still pass; this file is ignored):

```
go test
```

Output (the duration varies):

```
PASS
ok      desk    0.004s
```

Run with the tag:

```
go test -tags=integration
```

Output (the duration varies):

```
PASS
ok      desk    0.004s
```

Confirm the tag is doing work:

```
go test -tags=integration -v -run TestHealthIntegration
```

Output (the duration varies):

```
=== RUN   TestHealthIntegration
--- PASS: TestHealthIntegration (0.00s)
PASS
ok      desk    0.004s
```

Without `-tags=integration`, that `-run` matches nothing and `go test` still exits 0 (no tests to run is not a failure unless you pass `-count` tricks or use CI that checks coverage of that name). Your CI job for integration is the one that passes `-tags=integration`.

The trap

Sleeping to "wait for the server" or for a clock. Flaky on a loaded machine, slow on a fast one.

Save as `sleepy_test.go` — do not keep this as the real test:

```
// sleepy_test.go
package desk

import (
    "testing"
    "time"
)

func TestOpenNowFlaky(t *testing.T) {
    time.Sleep(50 * time.Millisecond)
    _ = OpenNow()
}
```

It "passes" and teaches nothing. Use `httptest.NewServer` (it is listening before it returns) and `fakeClock`. If you must wait for a condition, poll with `t.Context()` and a timeout — still not `Sleep` as the assertion.

The boring rule

- `httptest.NewServer` for client/server tests. `defer ts.Close()`.
- Inject time through a tiny `Clock`. Fake it in tests.
- Tag slow or environment-heavy tests: `//go:build integration`, then `go test -tags=integration`.
- Default `go test` stays unit-fast.
- No `time.Sleep` as a synchronization strategy.

- Integration tests still use tables, `t.Fatalf`, and `t.Helper`. The extra machinery is the server or the clock, not a new framework.

Try this

1. In `TestMenuServer`, GET `/missing` and assert `404`.
2. Add a `TestIsOpen` case at exactly `08:00`. Decide if that instant is open (the code uses `!Before`, so yes).
3. Move `TestMenuServer` behind `//go:build integration` and confirm plain `go test` no longer runs it. Restore or keep the tag on purpose.

Testing Async Code with synctest

Testing Async Code with synctest

The last chapter used a **Clock** interface when *you* own time. Often the code under test already calls `time.Sleep`, `time.After`, or `context.WithTimeout`. Do not rewrite production APIs just to make a test finish. Prefer `testing/synctest`: run the test in a **bubble** with a fake clock so five seconds of deadline logic finishes in a blink. Stable. Fast. No `GOEXPERIMENT` on Go 1.25+ (this book is on 1.27).

Mental model

- `synctest.Test(t, f)` runs `f` inside a new **bubble**. Goroutines started in `f` stay in that bubble.
- Inside a bubble, `time` uses a **fake clock**. It always starts at midnight UTC **2000-01-01**.
- Time advances only when every goroutine in the bubble is **durably blocked** (waiting on something only another bubble goroutine can unblock): `time.Sleep`, channel send/receive on channels created in the bubble, `WaitGroup.Wait` tied to the bubble, and a few related cases.
- `synctest.Wait()` blocks until every *other* goroutine in the bubble is durably blocked. Use it after you expect background work (or a timer callback) to have run.
- `t.Context()` inside the bubble is tied to the bubble. Prefer it over `context.Background()` in these tests.
- Real network sockets are **not** durably blocking. Prefer `net.Pipe`, `httptest`, or fakes — or use `httptest.NewTestServer` (Go 1.27) when you need HTTP inside a bubble.
- Go **1.27** adds `synctest.Sleep(d)`: `time.Sleep(d)` plus `Wait` in one call. Use it when you only need "advance and quiesce."

A `Clock` interface is still the right boring default when *your* package decides what "now" means. Reach for synctest when the standard library's timers are already on the path.

Worked examples

Case 1: A desk hold that times out

`HoldTicket` blocks until the context is done. The test proves it still holds just before the deadline and returns `context.DeadlineExceeded` after.

Empty directory:

```
go mod init desk
```

Save as `hold.go`:

```go
// hold.go
package desk

import (
    "context"
)

// HoldTicket blocks until ctx is done, then returns ctx.Err().
func HoldTicket(ctx context.Context) error {
    <-ctx.Done()
    return ctx.Err()
}
```

Save as `hold_test.go`:

```go
// hold_test.go
package desk

import (
    "context"
    "testing"
    "testing/synctest"
    "time"
)

func TestHoldTicketTimesOut(t *testing.T) {
    synctest.Test(t, func(t *testing.T) {
        const timeout = 5 * time.Second
        ctx, cancel := context.WithTimeout(t.Context(), timeout)
        defer cancel()

        errCh := make(chan error, 1)
        go func() {
            errCh <- HoldTicket(ctx)
        }()

        // Just before the deadline: still holding.
        time.Sleep(timeout - time.Nanosecond)
        synctest.Wait()
        select {
        case err := <-errCh:
            t.Fatalf("ticket released early: %v", err)
        default:
        }

        // Cross the deadline (fake clock - runs instantly).
        time.Sleep(time.Nanosecond)
```

```
        synctest.Wait()
        if err := <-errCh; err != context.DeadlineExceeded {
            t.Fatalf("got %v, want DeadlineExceeded", err)
        }
    })
}
```

Run:

```
go test
```

Output (the duration varies):

```
PASS
ok      desk    0.004s
```

Five seconds of timeout logic, milliseconds of wall clock. The `Wait` after each `Sleep` lets the context package's timer goroutines finish before you assert.

Case 2: Wait before you assert

Without `Wait`, a test can race the background goroutine. After cancel, wait for quiescence, then check the flag.

Save as `hold_wait_test.go` (same module as Case 1):

```
// hold_wait_test.go
package desk

import (
    "testing"
    "testing/synctest"
)

func TestHoldTicketReleasesOnCancel(t *testing.T) {
    synctest.Test(t, func(t *testing.T) {
        released := false
        ctx, cancel := context.WithCancel(t.Context())

        go func() {
            _ = HoldTicket(ctx)
            released = true
        }()

        synctest.Wait()
        if released {
            t.Fatal("released before cancel")
```

```
        }

        cancel()
        synctest.Wait()
        if !released {
            t.Fatal("not released after cancel")
        }
    })
}
```

Run:

```
go test
```

Output (the duration varies):

```
PASS
ok      desk    0.003s
```

`Wait` is not a sleep. It means "every other goroutine in this bubble is stuck waiting on something inside the bubble."

Case 3: `synctest.Sleep` (Go 1.27)

When the pattern is always "sleep, then wait for the bubble to settle," use the helper.

Save as `hold_sleep_test.go`:

```
// hold_sleep_test.go
package desk

import (
    "testing"
    "testing/synctest"
    "time"
)

func TestSynctestSleepAdvancesClock(t *testing.T) {
    synctest.Test(t, func(t *testing.T) {
        start := time.Now() // midnight UTC 2000-01-01 in the bubble
        done := false
        go func() {
            time.Sleep(2 * time.Second)
            done = true
        }()

        synctest.Sleep(2 * time.Second) // Sleep + Wait
```

```
        if !done {
            t.Fatal("background sleep not finished")
        }
        if got := time.Since(start); got != 2*time.Second {
            t.Fatalf("since start = %v, want 2s", got)
        }
    })
}
```

Run:

```
go test -run TestSynctestSleepAdvancesClock
```

Output (the duration varies):

```
PASS
ok      desk    0.002s
```

Prefer the explicit `time.Sleep` + `synctest.Wait` pair when you need an assert *between* advancing the clock and waiting. Prefer `synctest.Sleep` when you only need both.

The trap

A wall-clock sleep **outside** a bubble is slow and flaky. The "test" below can pass on a quiet laptop and fail under load — and it wastes two seconds every run.

Save as `trap_test.go` (do not keep this pattern):

```
// trap_test.go
package desk

import (
    "context"
    "testing"
    "time"
)

func TestHoldTicketWallClockTrap(t *testing.T) {
    t.Skip("trap demo - slow and flaky; use synctest instead")

    ctx, cancel := context.WithTimeout(context.Background(), 2*time.Second)
    defer cancel()

    errCh := make(chan error, 1)
    go func() {
        errCh <- HoldTicket(ctx)
```

```
    }()

    time.Sleep(3 * time.Second) // real wall clock
    if err := <-errCh; err != context.DeadlineExceeded {
        t.Fatalf("got %v, want DeadlineExceeded", err)
    }
}
```

Run (shows the skip):

```
go test -run TestHoldTicketWallClockTrap -v
```

Output:

```
=== RUN   TestHoldTicketWallClockTrap
    trap_test.go:11: trap demo - slow and flaky; use synctest instead
--- SKIP: TestHoldTicketWallClockTrap (0.00s)
PASS
ok      desk    0.002s
```

Second trap: blocking on a **real** network socket inside a bubble. That wait is not durably blocked, so `Wait` and the fake clock can deadlock or hang. Use `net.Pipe`, in-memory fakes, or `httptest.NewTestServer` (1.27) when HTTP must live in the bubble.

The boring rule

- Prefer sync APIs when you control the design. Prefer synctest when `time` / `context` timers are already in the path.
- Always wrap async timer tests in `synctest.Test`. Use `t.Context()` inside the bubble.
- After you expect background work or a timer to fire, call `synctest.Wait` (or `synctest.Sleep`) before asserting.
- Keep a `Clock` interface when *your* package owns "now." Do not invent a clock just to avoid learning synctest.
- Do not use `GOEXPERIMENT=synctest`. That was for the 1.24 experiment. On 1.25+ the package is ordinary stdlib.
- Do not dial the real network inside a bubble. Fake the wire.

Try this

1. Start from Case 1. Before the deadline, assert `ctx.Err() == nil` on the parent context (not only that `errCh` is empty).
2. Replace `WithTimeout` with `WithCancel` and `context.AfterFunc` that sets a `bool`. Cancel, `Wait`, assert the flag — same shape as the package docs.

3. Change Case 3 to `synctest.Sleep(1 * time.Second)` while the background goroutine sleeps two seconds. Predict whether `done` is true, run the test, then fix the sleep so it passes.
4. Optional: move Case 1's two-step sleep into one `synctest.Sleep(timeout)` and assert only *after* the deadline. What coverage did you lose?

Concurrency

Concurrency Basics

Concurrency Basics

A **goroutine** is a function the Go runtime can run alongside others. Starting one is a single keyword. Stopping it, and knowing it finished, is your job. The boring default is: **do not guess that work is done — wait for it**.

Mental model

`go f()` schedules `f` to run concurrently with the caller. It does not create an operating-system thread per call. The runtime multiplexes goroutines onto a small pool of threads.

Three facts matter on day one:

- When `main` returns, the process exits. Other goroutines are not asked politely. They vanish.
- A **WaitGroup** counts unfinished tasks. `Wait` blocks until the count is zero.
- A **channel** is a typed pipe. Send (`ch <- v`) and receive (`v := <-ch`) transfer a value and, on an unbuffered channel, meet in time.

`time.Sleep` is not a synchronization tool. It is a pause. If you sleep "long enough," you will be wrong on a slow machine, a busy desk, or the first production deploy.

Worked examples

Case 1: Main returns, the shift vanishes

Save as `lost_shift.go`. The goroutine would print a clock-in. `main` does not wait.

```
// lost_shift.go
package main

import "fmt"

func main() {
    go fmt.Println("Amina clocked in")
}
```

Run:

```
go run lost_shift.go
```

Typical output (empty — the process already exited):

Sometimes you will see the line. That luck is the bug. The program is not "usually fine." It is a race against `main`.

Case 2: WaitGroup counts the work

Save as `wait_shift.go`. `WaitGroup.Go` (Go 1.25+) starts the goroutine and accounts for it. `Wait` returns only after the function returns.

```
// wait_shift.go
package main

import (
    "fmt"
    "sync"
)

func main() {
    var wg sync.WaitGroup
    wg.Go(func() {
        fmt.Println("Amina clocked in")
    })
    wg.Wait()
}
```

Run:

```
go run wait_shift.go
```

Output:

```
Amina clocked in
```

Older code uses `wg.Add(1)`, `go func() { defer wg.Done(); ... }()`, then `wg.Wait()`. Same contract. Prefer `Go` in new Go 1.27 code. The function passed to `Go` must not panic.

Case 3: Channel send and receive

Save as `handoff.go`. The printer goroutine sends one ticket id. `main` receives it. The receive is the wait.

```
// handoff.go
package main

import "fmt"
```

```
func printer(id int, out chan<- int) {
    out <- id
}

func main() {
    ch := make(chan int)
    go printer(7, ch)
    got := <-ch
    fmt.Println("printed ticket", got)
}
```

Run:

```
go run handoff.go
```

Output:

```
printed ticket 7
```

`chan<- int` is a send-only view. The printer cannot receive. `main` uses a plain `chan int` and receives. The send in `printer` cannot finish until `main` is ready to receive: that meeting is the **rendezvous**.

Case 4: Several workers, one WaitGroup

Save as `three_windows.go`. Three windows clock in. The WaitGroup holds `main` until all three functions return. Print order is not guaranteed, so we collect names and sort them.

```
// three_windows.go
package main

import (
    "fmt"
    "slices"
    "sync"
)

func main() {
    names := []string{"Amina", "Bo", "Chen"}
    got := make([]string, len(names))
    var wg sync.WaitGroup
    for i, name := range names {
        wg.Go(func() {
            got[i] = name + " ready"
        })
    }
```

```
    wg.Wait()
    slices.Sort(got)
    for _, line := range got {
        fmt.Println(line)
    }
}
```

Run:

```
go run three_windows.go
```

Output:

```
Amina ready
Bo ready
Chen ready
```

Each goroutine writes a **different index**. That is safe. Two goroutines appending to the same slice would not be. The next chapters cover mutexes and channels for shared results.

Case 5: Receive the number of sends you planned

Save as `two_tickets.go`. Two sends, two receives. `main` does not exit early. Order of the two lines can swap, so we sort again.

```
// two_tickets.go
package main

import (
    "fmt"
    "slices"
)

func main() {
    ch := make(chan string)
    go func() { ch <- "ticket 7" }()
    go func() { ch <- "ticket 8" }()

    got := []string{<-ch, <-ch}
    slices.Sort(got)
    fmt.Println(got[0])
    fmt.Println(got[1])
}
```

Run:

```
go run two_tickets.go
```

Output:

```
ticket 7
ticket 8
```

If you receive only once, `main` can return while a send is still blocked. If you receive three times, `main` waits forever. Count the handoffs.

The trap

Save as `sleep_sync.go`. This treats a pause as a proof that the printer finished.

```
// sleep_sync.go
package main

import (
    "fmt"
    "time"
)

func main() {
    go fmt.Println("ticket 7 printed")
    time.Sleep(50 * time.Millisecond)
}
```

Run:

```
go run sleep_sync.go
```

Output on a quiet machine:

```
ticket 7 printed
```

It looks correct. Shorten the sleep, load the CPU, or put real work in the goroutine, and the line disappears like Case 1. Sleep is a delay. A WaitGroup or a channel receive is a **condition**. Use the condition.

The boring rule

- `go f()` starts work. It does not wait for work.
- When `main` returns, everything dies. Wait on purpose.
- Use `WaitGroup` when you need "all of these functions returned."
- Use a channel when you need a **value** (or a signal) from another goroutine.
- Never use `time.Sleep` to "give the goroutine time."
- Plan the number of sends and receives. A mismatch is a hang or a lost result.

Try this

1. In `wait_shift.go`, start a second `wg.Go` that prints `Bo clocked in`. Keep one `Wait`. Sort or accept either print order.
2. In `handoff.go`, change `make(chan int)` to `make(chan int, 1)`. Confirm it still prints. Then try sending two ids from `printer` without a second receive — the extra send blocks, and the program hangs until you add a receive or stop it.
3. In `two_tickets.go`, comment out one receive. Run it. Restore the receive before you move on.
4. Rewrite `three_windows.go` with `Add` / `Done` instead of `Go`. Same output.

Channel Patterns

Channel Patterns

Channels are how goroutines hand off tickets without sharing a slice. The boring patterns are few: **rendezvous or queue, close when you will send no more, `select` when you wait on more than one thing**. Everything else is a combination of those.

Mental model

`make(chan T)` is **unbuffered**. A send waits for a receive, and a receive waits for a send. That meeting is the whole synchronization.

`make(chan T, n)` is **buffered**. Send succeeds immediately until `n` values sit in the queue. Then send waits, same as unbuffered.

`close(ch)` means "no more sends." Receivers still drain what is there. `for v := range ch` ends after the close and the drain. Sending on a closed channel panics. Closing a nil channel panics. Closing twice panics.

`select` waits on several channel operations and runs one that is ready. If several are ready, it picks one at random. `time.After` gives you a channel that receives once when a duration elapses.

A **done** channel is usually `chan struct{}`. Close it to tell every waiter "stop." Closing broadcasts. Sending a single `struct{}{}` does not.

Worked examples

Case 1: Unbuffered rendezvous

Save as `rendezvous.go`. The window cannot finish the send until the printer is listening. That is the point: the handoff is a meeting, not a mailbox.

```
// rendezvous.go
package main

import "fmt"

func main() {
    ch := make(chan string)
    go func() {
        ch <- "ticket 7"
        fmt.Println("window: handed off")
```

```
    }()
    fmt.Println("printer:", <-ch)
}
```

Run:

```
go run rendezvous.go
```

Output (the two lines can appear in either order after the receive unblocks the send):

```
printer: ticket 7
window: handed off
```

or:

```
window: handed off
printer: ticket 7
```

The receive always happens. The print order of the two `Println` calls is the only wobble. Both lines always appear because `main` does not return until it has received.

Case 2: Buffered queue

Save as `buffer.go`. Capacity 2 lets the window drop two tickets without a printer standing there.

```
// buffer.go
package main

import "fmt"

func main() {
    ch := make(chan string, 2)
    ch <- "ticket 7"
    ch <- "ticket 8"
    fmt.Println(<-ch)
    fmt.Println(<-ch)
}
```

Run:

```
go run buffer.go
```

Output:

```
ticket 7
ticket 8
```

No goroutine. The buffer is a small, bounded queue. A third send with no receive would block `main` forever. Bound the buffer to a number you can defend ("two tickets in flight"), not "big enough that we never think about it."

Case 3: Close and range

Save as `close_range.go`. The kitchen sends every ticket, then closes. The printer ranges until the channel is empty and closed.

```
// close_range.go
package main

import "fmt"

func kitchen(out chan<- int) {
    for id := range 3 {
        out <- id + 7
    }
    close(out)
}

func main() {
    ch := make(chan int)
    go kitchen(ch)
    for id := range ch {
        fmt.Println("print", id)
    }
    fmt.Println("window closed")
}
```

Run:

```
go run close_range.go
```

Output:

```
print 7
print 8
print 9
window closed
```

The sender closes. The receiver ranges. Flip that and you hang or panic. `range` on a channel that is never closed is a loop that never ends.

Case 4: Select and a timeout

Save as `select_timeout.go`. The printer waits for a ticket or for `time.After`. Only the timeout is ready.

```
// select_timeout.go
package main

import (
    "fmt"
    "time"
)

func main() {
    tickets := make(chan int)
    select {
    case id := <-tickets:
        fmt.Println("print", id)
    case <-time.After(20 * time.Millisecond):
        fmt.Println("no ticket")
    }
}
```

Run:

```
go run select_timeout.go
```

Output:

```
no ticket
```

`time.After` is a one-shot timer channel. Nobody sends on `tickets`, so the timeout wins. If a send were ready too, `select` would pick at random — do not treat `select` as priority order unless you use a `default` or nested `select`.

Case 5: Fan-in

Save as `fan_in.go`. Two windows send into one printer channel. A WaitGroup closes the output when both inputs are done. Results are sorted so the listing is stable.

```
// fan_in.go
package main

import (
    "fmt"
    "slices"
    "sync"
```

```
)

func window(name string, ids []int, out chan<- string) {
    for _, id := range ids {
        out <- fmt.Sprintf("%s ticket %d", name, id)
    }
}

func main() {
    out := make(chan string)
    var wg sync.WaitGroup
    wg.Go(func() { window("A", []int{7, 8}, out) })
    wg.Go(func() { window("B", []int{9}, out) })
    go func() {
        wg.Wait()
        close(out)
    }()

    var got []string
    for line := range out {
        got = append(got, line)
    }
    slices.Sort(got)
    for _, line := range got {
        fmt.Println(line)
    }
}
```

Run:

```
go run fan_in.go
```

Output:

```
A ticket 7
A ticket 8
B ticket 9
```

Fan-in is "many senders, one channel." Close the channel from the **single** place that knows all senders finished — here, the goroutine that `Wait`s. A sender must not close a channel other senders still use.

Case 6: Done channel

Save as `done.go`. Closing `done` broadcasts stop. The worker waits on `select`. Only `done` is ready, so the result is stable. `Wait` proves the goroutine returned.

```
// done.go
package main

import (
    "fmt"
    "sync"
)

func worker(done <-chan struct{}, tickets <-chan int) {
    select {
    case <-done:
        fmt.Println("stopped")
    case id := <-tickets:
        fmt.Println("print", id)
    }
}

func main() {
    done := make(chan struct{})
    tickets := make(chan int)
    close(done)
    var wg sync.WaitGroup
    wg.Go(func() { worker(done, tickets) })
    wg.Wait()
    fmt.Println("desk closed")
}
```

Run:

```
go run done.go
```

Output:

```
stopped
desk closed
```

Close, do not send, to broadcast. If `tickets` also had a value ready, `select` would pick at random — keep tests to one ready case.

The trap

Save as `send_closed.go`. Closing then sending panics. The recover is only so you can see the message; do not recover this at the desk.

```
// send_closed.go
package main

import "fmt"

func main() {
    ch := make(chan int)
    close(ch)
    defer func() {
        fmt.Println("panic:", recover())
    }()
    ch <- 7
}
```

Run:

```
go run send_closed.go
```

Output:

```
panic: send on closed channel
```

The matching hang: `for range ch` when nobody ever closes `ch`. That program never prints "window closed." Close from the sender, or receive a fixed count, or wait on `done`. Do not hope.

The boring rule

- Unbuffered means "meet me." Buffered means "queue at most n."
- The sender closes. One closer. Never send after close.
- `range` over a channel until close when the stream is finite.
- `select` for "whichever happens." `time.After` for a deadline on that wait.
- Close a `done` channel to broadcast stop. Do not send on it as a broadcast.
- Fan-in: many senders, one channel, one close after the last sender finishes.

Try this

1. In `buffer.go`, add a third send and do not add a receive. Run it. Interrupt when it hangs. Then raise the buffer to 3 and run again.
2. In `close_range.go`, delete `close(out)`. Run it. Interrupt when it hangs. Put the close back.
3. In `select_timeout.go`, start a goroutine that sends `7` on `tickets` before the `select`. Run it several times. You may see `print 7` or `no ticket` depending on scheduling — then buffer the channel, send in `main` before `select`, and confirm you always print 7.
4. Change `done.go` so `tickets` is buffered and already holds `7` before `select`. Run it a few times. Notice `select` can pick either case. That is why this listing closes `done` with no competing ready send.

Synchronization

Synchronization

Two goroutines that both touch the same variable are sharing memory. The boring question is not "how do I make this faster?" It is **who owns this value right now?** A mutex says "we share, but only one at a time." A channel says "I am giving this to you." Prefer the second when you can. Use the first when a shared structure is the whole point.

Mental model

A **mutex** (`sync.Mutex`) is a lock. `Lock` then `Unlock`. The code between them is the **critical section**. Maps, slices, and ordinary integers are not safe for concurrent write. The mutex does not mark the data. You have to remember to take it.

A **channel** moves ownership. After `ch <- ticket`, the sender should not touch `ticket` again. After `ticket := <-ch`, the receiver is the owner. No lock, because there is no share.

`sync.Once` runs a function at most once, even if many goroutines call `Do`. Use it for process-wide setup (load the menu file), not for "call this sometimes."

`atomic` operations (`atomic.Int64`) are for simple counters and flags. They are not a general replacement for a mutex around a map.

A **data race** is two goroutines accessing the same memory, at least one writing, with no happens-before. The race detector (`go test -race` / `go run -race`) finds many of them. A race is a bug even if the printed number looks right today.

Worked examples

Case 1: Mutex around shared state

Save as `mutex_tables.go`. Several windows seat names into one map. The map is the shared state. The mutex is the rule.

```
// mutex_tables.go
package main

import (
    "fmt"
    "sync"
)
```

```
func main() {
    var mu sync.Mutex
    tables := map[int]string{}
    var wg sync.WaitGroup
    seats := []struct {
        n    int
        name string
    }{
        {12, "Amina"},
        {3, "Bo"},
        {7, "Chen"},
    }
    for _, s := range seats {
        wg.Go(func() {
            mu.Lock()
            tables[s.n] = s.name
            mu.Unlock()
        })
    }
    wg.Wait()
    mu.Lock()
    fmt.Println("table 3:", tables[3])
    fmt.Println("table 7:", tables[7])
    fmt.Println("table 12:", tables[12])
    mu.Unlock()
}
```

Run:

```
go run mutex_tables.go
```

Output:

```
table 3: Bo
table 7: Chen
table 12: Amina
```

The mutex makes each write **whole**. It does not decide business rules. If “table 12 already seated” should error, check under the same lock.

Case 2: Channel as ownership handoff

Save as `own_ticket.go`. The kitchen fills a ticket and sends it. The printer is the only goroutine that edits the note after the send.

```
// own_ticket.go
package main

import "fmt"

type Ticket struct {
    ID   int
    Note string
}

func kitchen(out chan<- Ticket) {
    t := Ticket{ID: 7, Note: "no onions"}
    out <- t
}

func main() {
    ch := make(chan Ticket)
    go kitchen(ch)
    t := <-ch
    t.Note = t.Note + ", extra napkins"
    fmt.Printf("ticket %d: %s\n", t.ID, t.Note)
}
```

Run:

```
go run own_ticket.go
```

Output:

```
ticket 7: no onions, extra napkins
```

No mutex. The value crossed the channel. If `kitchen` kept using `t` after the send, that would be a share again — send a copy and stop touching it.

Case 3: sync.Once

Save as `once_menu.go`. Three windows ask for the menu. The file is "loaded" once.

```
// once_menu.go
package main

import (
    "fmt"
    "sync"
)
```

```
func main() {
    var once sync.Once
    load := func() {
        fmt.Println("loaded menu")
    }
    var wg sync.WaitGroup
    for range 3 {
        wg.Go(func() {
            once.Do(load)
            fmt.Println("menu ready")
        })
    }
    wg.Wait()
}
```

Run:

```
go run once_menu.go
```

Output (the three `menu ready` lines may appear in any order; `loaded menu` prints once, before the `Do` calls return):

```
loaded menu
menu ready
menu ready
menu ready
```

`Do` waits if another goroutine is already inside `load`. Callers never see a half-loaded menu.

Case 4: Atomic counter

Save as `atomic_count.go`. A thousand increments, no mutex. `atomic.Int64` is the whole state.

```
// atomic_count.go
package main

import (
    "fmt"
    "sync"
    "sync/atomic"
)

func main() {
    var n atomic.Int64
    var wg sync.WaitGroup
    for range 1000 {
```

```
        wg.Go(func() {
            n.Add(1)
        })
    }
    wg.Wait()
    fmt.Println(n.Load())
}
```

Run:

```
go run atomic_count.go
```

Output:

```
1000
```

If the "counter" grows a second field (last ticket id, last name), you are back to a mutex. Atomics do not compose into a critical section.

The trap

Save as `race.go`. Two goroutines increment a plain `int`. That is a data race.

```
// race.go
package main

import (
    "fmt"
    "sync"
)

func main() {
    var n int
    var wg sync.WaitGroup
    for range 1000 {
        wg.Go(func() {
            n++
        })
    }
    wg.Wait()
    fmt.Println(n)
}
```

Run without the detector — the number may look fine or not:

```
go run race.go
```

Possible output (any value up to 1000; 1000 is luck, not proof):

```
1000
```

The race detector is the check that matters:

```
go run -race race.go
```

You should see `WARNING: DATA RACE` (file names and goroutine ids vary) and a non-zero exit. Treat that as a failed build.

The same check in a module (`go.mod` with `module desk`, plus the test file):

```
// race_test.go
package desk

import (
    "sync"
    "testing"
)

func TestTicketCountRace(t *testing.T) {
    var n int
    var wg sync.WaitGroup
    for range 100 {
        wg.Go(func() {
            n++
        })
    }
    wg.Wait()
    if n != 100 {
        t.Fatalf("got %d", n)
    }
}
```

Run:

```
go test -race
```

The count assertion may pass or fail. The detector still prints `WARNING: DATA RACE` and fails the command when it sees the race. Fix: a mutex around `n++`, or `atomic.Int64` as in Case 4. For a one-file check, prefer `go run -race race.go`.

The boring rule

- Mutex: shared structure, short critical section, `Unlock` with `defer` if the section has more than one return.
- Channel: hand off a value. Do not use both a mutex and a channel on the same data without a drawing.
- `Once` for one-time setup. Not for request logic.
- `atomic` for a counter or a flag. Not for a map.
- Run `go test -race` in CI. A silent race is still a race.
- Do not share a slice header or a map with append/assign from two goroutines.

Try this

1. In `mutex_tables.go`, delete both `mu.Lock()` / `mu.Unlock()` pairs inside the goroutines. Run `go run -race mutex_tables.go`. Put the locks back.
2. In `own_ticket.go`, after `out <- t`, add `t.Note = "changed in kitchen"` in `kitchen`. You will not see it on the printer — the copy already left. Then change the channel type to `chan *Ticket` and send `&t`. Now the late write is a share. Run `-race` if you write from both sides.
3. In `atomic_count.go`, replace `atomic.Int64` with a plain `int` and `n++`. Compare `go run` and `go run -race`.
4. Add `defer mu.Unlock()` immediately after `mu.Lock()` in Case 1's loop body, and delete the manual `Unlock`. Confirm the program still prints.

Context and Cancellation

Context and Cancellation

A **context** is how one function tells another "this work is still wanted." The boring default is: **the first argument is `ctx context.Context`**, you check it, and you pass it down. When the desk closes, workers stop because the context says so — not because someone pulled the power cord.

Mental model

`context.Background()` is the empty root. Use it in `main`, in tests, and at the edge of a program. `context.TODO()` is a placeholder when you have not wired a real context yet. Do not leave `TODO` in production paths.

`WithCancel` returns a child context and a `cancel` function. Calling `cancel()` makes `ctx.Done()` ready. `ctx.Err()` becomes `context.Canceled`.

`WithTimeout` / `WithDeadline` cancel automatically when time runs out. `ctx.Err()` is then `context.DeadlineExceeded`. Always `defer cancel()` so the timer is released if you finish early.

`WithCancelCause` (and `context.Cause`) attach a real error ("kitchen closed") instead of the generic canceled value.

Cancel is **advisory**. A worker that never looks at `ctx` will not stop. The next chapter is about not starting goroutines you cannot stop. This chapter is the stop signal itself.

Do not store a context inside a struct that outlives one request. Pass it as the first argument.

Worked examples

Case 1: Background, first argument

Save as `ctx_print.go`. The printer takes `ctx` first even though this call never cancels. The signature is the habit.

```
// ctx_print.go
package main

import (
    "context"
    "fmt"
)
```

```
func printTicket(ctx context.Context, id int) error {
    if err := ctx.Err(); err != nil {
        return err
    }
    fmt.Println("printed", id)
    return nil
}

func main() {
    ctx := context.Background()
    if err := printTicket(ctx, 7); err != nil {
        fmt.Println("err:", err)
    }
}
```

Run:

```
go run ctx_print.go
```

Output:

```
printed 7
```

`Background` is never canceled. `ctx.Err()` is `nil`. The check is still the right first line for anything that might later run under a timeout.

Case 2: WithCancel stops a worker

Save as `cancel_worker.go`. The worker loops on tickets until `ctx.Done()`. `main` cancels, then waits.

```
// cancel_worker.go
package main

import (
    "context"
    "fmt"
    "sync"
)

func worker(ctx context.Context, tickets <-chan int) {
    for {
        select {
        case <-ctx.Done():
            fmt.Println("stopped:", ctx.Err())
            return
```

```
        case id, ok := <-tickets:
            if !ok {
                fmt.Println("queue closed")
                return
            }
            fmt.Println("printed", id)
        }
    }
}

func main() {
    ctx, cancel := context.WithCancel(context.Background())
    tickets := make(chan int)
    var wg sync.WaitGroup
    wg.Go(func() { worker(ctx, tickets) })
    cancel()
    wg.Wait()
}
```

Run:

```
go run cancel_worker.go
```

Output:

```
stopped: context canceled
```

`tickets` is never ready. `Done` is. The worker returns. `Wait` unblocks. If you omit `cancel()`, this program hangs. If you omit `Wait`, you are back to killing the worker by exiting `main`.

Case 3: WithTimeout

Save as `timeout.go`. The kitchen is slow. The desk only waits 20 milliseconds.

```
// timeout.go
package main

import (
    "context"
    "fmt"
    "time"
)

func kitchen(ctx context.Context) error {
    select {
    case <-time.After(time.Second):
```

```
        fmt.Println("order ready")
        return nil
    case <-ctx.Done():
        return ctx.Err()
    }
}

func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 20*time.Millisecond)
    defer cancel()
    err := kitchen(ctx)
    fmt.Println("desk:", err)
}
```

Run:

```
go run timeout.go
```

Output:

```
desk: context deadline exceeded
```

`defer cancel()` stops the timeout timer as soon as `main` returns. Even though the deadline already fired, calling `cancel` is still required — if the kitchen had finished in 1ms, you would leak the timer without it.

Case 4: WithCancelCause

Save as `cause.go`. The cancel reason is a desk error, not a generic string you parse later.

```
// cause.go
package main

import (
    "context"
    "fmt"
)

func main() {
    ctx, cancel := context.WithCancelCause(context.Background())
    cancel(fmt.Errorf("kitchen closed"))
    fmt.Println("err:", ctx.Err())
    fmt.Println("cause:", context.Cause(ctx))
}
```

Run:

```
go run cause.go
```

Output:

```
err: context canceled
cause: kitchen closed
```

`Err` stays `context.Canceled`. `Cause` is the error you passed. `errors.Is(ctx.Err(), context.Canceled)` is still true. Use `Cause` when a human or a log needs the why.

Case 5: Timeout plus a worker that checks

Save as `prep.go`. Same worker shape as Case 2, deadline as Case 3.

```
// prep.go
package main

import (
    "context"
    "fmt"
    "sync"
    "time"
)

func prep(ctx context.Context) {
    select {
    case <-time.After(time.Second):
        fmt.Println("prep done")
    case <-ctx.Done():
        fmt.Println("prep stopped:", context.Cause(ctx))
    }
}

func main() {
    ctx, cancel := context.WithTimeoutCause(
        context.Background(),
        20*time.Millisecond,
        fmt.Errorf("shift ended"),
    )
    defer cancel()
    var wg sync.WaitGroup
    wg.Go(func() { prep(ctx) })
    wg.Wait()
}
```

Run:

```
go run prep.go
```

Output:

```
prep stopped: shift ended
```

`WithTimeoutCause` is Go 1.21+. `Cause` is `shift ended`. `Err` would still be `context.DeadlineExceeded`. Pass the same `ctx` into every function that should stop together.

The trap

Save as `ignore_ctx.go`. The worker takes a context and never looks at it. Cancel does nothing. `main` waits on a channel that nobody receives from — hang — so we use a timeout on `main` to show the worker is still alive.

```
// ignore_ctx.go
package main

import (
    "context"
    "fmt"
    "time"
)

func worker(ctx context.Context, out chan<- string) {
    time.Sleep(50 * time.Millisecond)
    out <- "still working"
}

func main() {
    ctx, cancel := context.WithCancel(context.Background())
    out := make(chan string, 1)
    go worker(ctx, out)
    cancel()
    select {
    case msg := <-out:
        fmt.Println(msg)
    case <-time.After(200 * time.Millisecond):
        fmt.Println("main gave up")
    }
}
```

Run:

```
go run ignore_ctx.go
```

Output:

```
still working
```

Cancel ran. The worker slept anyway and sent. That is a leak of work and, if `out` were unbuffered and `main` had moved on, a leaked goroutine blocked on send. Checking `ctx` is not decoration. The fix is Case 2: `select` on `ctx.Done()` and the real work.

The boring rule

- First argument: `ctx context.Context`.
- Create with `Background` at the edge; derive cancel/timeout children below.
- `defer cancel()` every time you get a `cancel` function.
- Workers `select` on `ctx.Done()` (or return `ctx.Err()` at the top of a loop).
- Prefer `WithCancelCause` / `WithTimeoutCause` when the reason matters in logs.
- Do not put optional parameters in `context.Value`. A function argument is cheaper and typed.
- Do not store `ctx` on a long-lived struct.

Try this

1. In `cancel_worker.go`, send `7` on a buffered `tickets` channel before `cancel()`. Run it a few times. You may see `printed 7` then `stopped`, or only `stopped`. Then send after the worker is running and before cancel, with `Wait` still at the end.
2. In `timeout.go`, raise the timeout to `2 * time.Second`. Confirm `order ready` and `desk: <nil>`.
3. In `cause.go`, print `context.Cause(ctx)` before `cancel`. It should match `ctx.Err()` (`nil` until cancel).
4. Fix `ignore_ctx.go`: in `worker`, `select` on `<-ctx.Done()` and `time.After(50 * time.Millisecond)`. After `cancel()`, you should print a stop path, not `still working`.

Concurrency Design Guidelines

Concurrency Design Guidelines

Goroutines are cheap. Leaked goroutines are not. The boring default is: **write the sequential version first**, start a goroutine only when you can state how it stops, and bound how much work can run at once.

Mental model

A goroutine is a leak if:

- nothing will ever receive the value it is sending
- nothing will ever send the value it is receiving
- it is in a loop that never looks at a `done` channel or `ctx.Done()`
- `main` or a request handler returned and the goroutine is still blocked

"I started a goroutine per ticket" is unbounded. A slow kitchen plus a busy window becomes a process that grows until it dies.

Sequential code has one stack, one error path, and an obvious end. Concurrent code needs a stop signal, a wait, and a cap. If you cannot name those three, do not type `go`.

Worked examples

Case 1: Sequential is enough

Save as `seq_print.go`. Three tickets, one printer, no goroutines. This is the version to ship until a measurement says you are waiting on something real.

```
// seq_print.go
package main

import "fmt"

func printTicket(id int) {
    fmt.Println("printed", id)
}

func main() {
    for _, id := range []int{7, 8, 9} {
        printTicket(id)
```

```
    }
}
```

Run:

```
go run seq_print.go
```

Output:

```
printed 7
printed 8
printed 9
```

Same order every run. Errors would return to `main` on the same line. Start here.

Case 2: The leak

Save as `leak.go`. A worker sends a ticket. Nobody receives. The goroutine blocks forever. `main` returns, so this process exits — in a server, `main` would not return, and the goroutine would stay.

We keep the process alive long enough to show the hang using a second channel `main` never hears from the worker on.

```
// leak.go
package main

import (
    "fmt"
    "time"
)

func leak(out chan<- int) {
    out <- 7
    fmt.Println("sent")
}

func main() {
    ch := make(chan int)
    go leak(ch)
    time.Sleep(30 * time.Millisecond)
    fmt.Println("main leaving; worker still blocked on send")
}
```

Run:

```
go run leak.go
```

Output:

```
main leaving; worker still blocked on send
```

`sent` never prints. The goroutine is stuck on `out <- 7`. In a server that calls `leak` per request, that is a goroutine leak. Sleep here is only a window to observe the bug, not a fix.

Case 3: Fix the leak

Save as `no_leak.go`. The worker `selects` on send and on `ctx.Done()`. Cancel stops the send. `Wait` proves the goroutine returned.

```go
// no_leak.go
package main

import (
    "context"
    "fmt"
    "sync"
)

func worker(ctx context.Context, out chan<- int) {
    select {
    case out <- 7:
        fmt.Println("sent")
    case <-ctx.Done():
        fmt.Println("abandoned:", ctx.Err())
    }
}

func main() {
    ctx, cancel := context.WithCancel(context.Background())
    ch := make(chan int)
    var wg sync.WaitGroup
    wg.Go(func() { worker(ctx, ch) })
    cancel()
    wg.Wait()
}
```

Run:

```
go run no_leak.go
```

Output:

```
abandoned: context canceled
```

Nobody received. That is fine: the worker gave up because the context said the ticket is no longer wanted. The goroutine returned. `Wait` ended. No leak.

If `main` still wanted the value, the fix would be to receive: `fmt.Println(<-ch)` and skip cancel. Stopping and delivering are different designs. Pick one.

Case 4: Bound the work

Save as `bound.go`. At most two printers run at once. Extra tickets wait on a semaphore channel.

```
// bound.go
package main

import (
    "fmt"
    "sync"
)

func main() {
    tickets := []int{7, 8, 9, 10}
    sem := make(chan struct{}, 2)
    var wg sync.WaitGroup
    for _, id := range tickets {
        sem <- struct{}{}
        wg.Go(func() {
            defer func() { <-sem }()
            fmt.Println("printed", id)
        })
    }
    wg.Wait()
    fmt.Println("done")
}
```

Run:

```
go run bound.go
```

Output (print order among the four tickets may vary; `done` is last):

```
printed 8
printed 9
printed 10
printed 7
done
```

`sem` has capacity 2. The third `sem <- struct{}{}` waits until a printer receives from `sem` in `defer`. You can name the cap (`const maxPrinters = 2`). You cannot name "one goroutine per incoming request forever."

To make the listing stable for tests, collect ids and sort:

```
// bound_sorted.go
package main

import (
    "fmt"
    "slices"
    "sync"
)

func main() {
    tickets := []int{7, 8, 9, 10}
    sem := make(chan struct{}, 2)
    got := make([]int, len(tickets))
    var wg sync.WaitGroup
    for i, id := range tickets {
        sem <- struct{}{}
        wg.Go(func() {
            defer func() { <-sem }()
            got[i] = id
        })
    }
    wg.Wait()
    slices.Sort(got)
    fmt.Println(got)
    fmt.Println("done")
}
```

Run:

```
go run bound_sorted.go
```

Output:

```
[7 8 9 10]
done
```

Same bound, stable print.

The trap

Save as `per_ticket.go`. A goroutine per ticket, no cap, no stop, no wait except `WaitGroup`. The WaitGroup prevents a *process* leak on this tiny list. The design still has no bound and no cancel. Replace `tickets` with a million ids from a live queue and you have a million goroutines.

```go
// per_ticket.go
package main

import (
    "fmt"
    "sync"
)

func main() {
    tickets := []int{7, 8, 9}
    var wg sync.WaitGroup
    for _, id := range tickets {
        wg.Go(func() {
            fmt.Println("printed", id)
        })
    }
    wg.Wait()
}
```

Run:

```
go run per_ticket.go
```

Possible output (order varies):

```
printed 7
printed 8
printed 9
```

It "works" for three. That is the trap. The sequential program (Case 1) was clearer. The bounded program (Case 4) is what you graduate to when printing is actually slow. The leaked program (Case 2) is what you get when you add `go` and forget the receive.

Fix on a live desk:

- cap with a semaphore or a fixed worker pool
- pass `ctx` and `select` on `Done`
- `Wait` (or return errors on a channel) before the handler returns

The boring rule

- Sequential first. Concurrent when a clock or a profiler says so.
- Do not start a goroutine you cannot stop.
- Every `go` has a matching wait (`WaitGroup` or a receive) on the path that cares.
- Bound in-flight work with a number you can explain.
- Close or cancel from the owner. Do not leak senders or receivers.
- Never use `time.Sleep` as the stop policy.

Try this

1. In `leak.go`, add `fmt.Println(<-ch)` in `main` and delete the sleep. Confirm `sent` and `7` print.
2. In `no_leak.go`, receive from `ch` in `main` instead of calling `cancel()`. Confirm `sent` prints and `Wait` still returns.
3. In `bound.go`, set the semaphore capacity to 1. The program still finishes; it is just a queue of one printer.
4. Add a `context.WithCancel` to `bound_sorted.go`. Cancel after starting. Without a `select` on `ctx.Done()`, workers still run. Add the `select` (or a check at the start of the goroutine) so cancel actually stops new prints.

Standard Library

Time and Scheduling

Time and Scheduling

`time.Time` is an instant. `time.Duration` is a length. The boring default is: **store instants, compare them, and let a Timer or Ticker fire**. Do not use `time.Sleep` as the program's clock.

Mental model

`Time` holds wall time and a location. `t.Add(d)` and `t.Sub(u)` are the arithmetic. `t.Before(u)`, `t.After(u)`, `t.Equal(u)` are the comparisons. `Equal` is timezone-aware; `==` on `Time` values is not what you want.

`Duration` is nanoseconds in an `int64`. Literals look like `20 * time.Millisecond`. There is no `time.Hour` of wall-clock "wait until 8am" — that is a `Time` plus `time.Until`.

A **Timer** fires once. A **Ticker** fires repeatedly. Both have a `C` channel and a `Stop` method. Stop a ticker when you are done. A timer you no longer need should be stopped too, and if `Stop` returns false you may still need to drain `C`.

`time.Now()` is the real clock. In examples and in testable desk logic, pass `now time.Time` into the function so the clock is an argument, not a hidden `Now()` call.

Worked examples

Case 1: Time and Duration

Save as `shift_length.go`. Two instants, one duration.

```
// shift_length.go
package main

import (
    "fmt"
    "time"
)

func main() {
    start := time.Date(2026, 9, 7, 8, 0, 0, 0, time.UTC)
    end := time.Date(2026, 9, 7, 16, 30, 0, 0, time.UTC)
    d := end.Sub(start)
```

```
    fmt.Println(start.Format(time.RFC3339))
    fmt.Println(d)
    fmt.Println(d.Hours())
}
```

Run:

```
go run shift_length.go
```

Output:

```
2026-09-07T08:00:00Z
8h30m0s
8.5
```

`Format` is for people and logs. `Sub` is for math. Do not parse your own formatted strings back to get a duration.

Case 2: Location

Save as `desk_zone.go`. The same instant in UTC and on the desk clock.

```
// desk_zone.go
package main

import (
    "fmt"
    "time"
)

func main() {
    utc := time.Date(2026, 9, 7, 12, 0, 0, 0, time.UTC)
    desk := time.FixedZone("desk", -4*60*60)
    local := utc.In(desk)
    fmt.Println(utc.Format(time.RFC3339))
    fmt.Println(local.Format(time.RFC3339))
    fmt.Println(local.Hour())
}
```

Run:

```
go run desk_zone.go
```

Output:

```
2026-09-07T12:00:00Z
2026-09-07T08:00:00-04:00
8
```

`FixedZone` always works. `time.LoadLocation("America/New_York")` needs timezone data on the machine (or `import _ "time/tzdata"`). For a desk in one building, a fixed offset or a loaded location both beat storing wall-clock strings with no zone.

Case 3: Open window from a clock argument

Save as `open.go`. Whether the desk is open is a function of `now`, not of `Sleep`.

```
// open.go
package main

import (
    "fmt"
    "time"
)

func open(now time.Time) bool {
    h := now.UTC().Hour()
    return h >= 8 && h < 22
}

func main() {
    morning := time.Date(2026, 9, 7, 8, 0, 0, 0, time.UTC)
    night := time.Date(2026, 9, 7, 23, 0, 0, 0, time.UTC)
    fmt.Println("08:00", open(morning))
    fmt.Println("23:00", open(night))
}
```

Run:

```
go run open.go
```

Output:

```
08:00 true
23:00 false
```

Tests pass a `Time`. Production passes `time.Now()`. The rule lives in one function.

Case 4: Timer

Save as `timer.go`. One shot. Stop in `defer` so a later edit that returns early does not leak the timer.

```
// timer.go
package main

import (
    "fmt"
    "time"
)

func main() {
    t := time.NewTimer(20 * time.Millisecond)
    defer t.Stop()
    <-t.C
    fmt.Println("shift started")
}
```

Run:

```
go run timer.go
```

Output:

```
shift started
```

`time.After` is a timer you cannot stop. Use `NewTimer` when the function might return before the duration (a context canceled, a ticket arrived). Use `After` in a `select` that already has a clear lifetime, as in the channel chapter.

Case 5: Ticker

Save as `ticker.go`. Three ticks, then stop.

```
// ticker.go
package main

import (
    "fmt"
    "time"
)

func main() {
    tk := time.NewTicker(15 * time.Millisecond)
    defer tk.Stop()
```

```
    for i := range 3 {
        <-tk.C
        fmt.Println("tick", i+1)
    }
}
```

Run:

```
go run ticker.go
```

Output:

```
tick 1
tick 2
tick 3
```

Without `Stop`, the ticker keeps a goroutine and a channel alive until the process exits. In `main` that is invisible. In a request handler it is a leak. `range tk.C` without another stop path is a loop that never ends.

The trap

Save as `sleep_clock.go`. This treats sleep as the definition of "one second later." The CPU, the scheduler, and a laptop lid disagree.

```
// sleep_clock.go
package main

import (
    "fmt"
    "time"
)

func main() {
    start := time.Date(2026, 9, 7, 8, 0, 0, 0, time.UTC)
    time.Sleep(20 * time.Millisecond)
    // wrong idea: "start plus one second" because we slept
    fmt.Println("still", start.Format("15:04:05"))
    fmt.Println("real now is not start+sleep")
}
```

Run:

```
go run sleep_clock.go
```

Output:

```
still 08:00:00
real now is not start+sleep
```

`start` did not move. Sleeping does not advance a `Time` you already hold. If you need "now," call `time.Now()` (or take `now` as an argument). If you need "wait until 16:00," compute `time.Until(deadline)` and use a `Timer` — and still cancel it if the desk closes early.

A second form of the same trap: `time.Sleep(8 * time.Hour)` to wait for the next shift. The process cannot react to a cancel. A `select` on `ctx.Done()` and `time.After(time.Until(open))` can.

The boring rule

- `Time` for instants, `Duration` for lengths. Do not mix them in a raw `int`.
- Pass `now time.Time` into business functions.
- Store UTC (or a named location you control). Parse with the zone attached.
- `Timer` once, `Ticker` many. `Stop` both.
- Do not `Sleep` to mean "the clock moved" or "the other goroutine finished."
- `time.Until` / `time.Since` instead of `Now().Sub(...)` when that is what you mean.

Try this

1. In `shift_length.go`, print `end.Before(start)` and `start.Add(d).Equal(end)`.
2. In `open.go`, add a noon instant and print `open(noon)`.
3. In `ticker.go`, drop `defer tk.Stop()` and add a `break` after one tick. Then put `Stop` back — the break is the case that needs it.
4. In `timer.go`, add a `done` channel, close it, and `select` on `t.C` and `done`. Confirm you can skip `shift started` when `done` is already closed.

I/O and Streaming

I/O and Streaming

Go I/O is two interfaces: `io.Reader` (`Read`) and `io.Writer` (`Write`). Files, buffers, HTTP bodies, and `os.Stdout` all speak them. The boring default is: **copy streams with `io.Copy`, wrap with `bufio` when you read lines, and close what you open**.

Mental model

`Read` fills a slice and returns how many bytes and an error. `io.EOF` means "no more." It is not a failure of the program; it is the end of the stream. `Write` takes a slice and returns how many bytes were accepted.

`io.Copy(dst, src)` is the loop you should not write by hand. `bytes.Buffer` is an in-memory `Reader` and `Writer`. `bufio.Scanner` splits a reader into lines (or other tokens). `os.Open` opens a file for reading. `os.CreateTemp` makes a unique file in the temp directory — use it in examples and tests so the program does not depend on a path on your desk.

Always check the error from `Close` on writers. A buffered write can fail at flush.

Worked examples

Case 1: Reader to Writer

Save as `copy_tickets.go`. A string is a reader. A buffer is a writer. `io.Copy` moves the bytes.

```
// copy_tickets.go
package main

import (
    "bytes"
    "fmt"
    "io"
    "strings"
)

func main() {
    src := strings.NewReader("ticket 7\nticket 8\n")
    var dst bytes.Buffer
    n, err := io.Copy(&dst, src)
    if err != nil {
```

```
        fmt.Println(err)
        return
    }
    fmt.Println("bytes", n)
    fmt.Print(dst.String())
}
```

Run:

```
go run copy_tickets.go
```

Output:

```
bytes 18
ticket 7
ticket 8
```

`strings.NewReader` never fails. A file or a socket might. Keep the `err` check anyway so the code looks like production.

Case 2: bytes.Buffer as a builder

Save as `buffer_note.go`. Write pieces, then take the string.

```
// buffer_note.go
package main

import (
    "bytes"
    "fmt"
)

func main() {
    var b bytes.Buffer
    b.WriteString("ticket 7")
    b.WriteByte(':')
    b.WriteString(" no onions")
    fmt.Println(b.String())
}
```

Run:

```
go run buffer_note.go
```

Output:

```
ticket 7: no onions
```

`Buffer` grows as needed. For a few strings, `fmt.Sprintf` or `strings.Builder` is also fine. Use `Buffer` when you already have `io.Writer` code.

Case 3: bufio.Scanner

Save as `scan.go`. Line at a time.

```
// scan.go
package main

import (
    "bufio"
    "fmt"
    "strings"
)

func main() {
    r := strings.NewReader("7\n8\n9\n")
    s := bufio.NewScanner(r)
    for s.Scan() {
        fmt.Println("ticket", s.Text())
    }
    if err := s.Err(); err != nil {
        fmt.Println(err)
    }
}
```

Run:

```
go run scan.go
```

Output:

```
ticket 7
ticket 8
ticket 9
```

`Scan` returns false on EOF or error. Check `s.Err()` after the loop. Default split is lines. Default max token is 64KiB — a giant line needs a larger buffer or a different reader.

Case 4: CreateTemp, write, Open, read

Save as `temp_order.go`. The program makes its own file, writes an order, opens it again, and prints the contents. No path on your machine is required.

```
// temp_order.go
package main

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

func main() {
    f, err := os.CreateTemp("", "order-*.txt")
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    name := f.Name()
    defer os.Remove(name)

    if _, err := f.WriteString("table 12, ticket 7\n"); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    if err := f.Close(); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }

    in, err := os.Open(name)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    defer in.Close()

    got, err := io.ReadAll(in)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    fmt.Print(string(got))
}
```

Run:

```
go run temp_order.go
```

Output:

```
table 12, ticket 7
```

`CreateTemp` picks a unique name. `defer os.Remove(name)` cleans up even if a later check fails. `os.Open` is read-only. `WriteString` then `Close` the writer **before** `Open` so the bytes are on disk.

Case 5: Stream lines from the temp file

Save as `temp_scan.go`. Same file, `bufio` instead of `ReadAll`.

```
// temp_scan.go
package main

import (
    "bufio"
    "fmt"
    "os"
)

func main() {
    f, err := os.CreateTemp("", "tickets-*.txt")
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    name := f.Name()
    defer os.Remove(name)
    if _, err := f.WriteString("7\n8\n9\n"); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    if err := f.Close(); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }

    in, err := os.Open(name)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    defer in.Close()

    s := bufio.NewScanner(in)
    for s.Scan() {
        fmt.Println("print", s.Text())
    }
```

```
    if err := s.Err(); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}
```

Run:

```
go run temp_scan.go
```

Output:

```
print 7
print 8
print 9
```

This is the shape for a log or a large order file: do not `ReadAll` unless the size is known and small.

The trap

Save as `forget_close.go`. The write is buffered in the file's world only after `Close` (or `Sync`). This program reads the same file handle without seeking back, so it looks empty.

```
// forget_close.go
package main

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

func main() {
    f, err := os.CreateTemp("", "stale-*.txt")
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    defer os.Remove(f.Name())
    defer f.Close()

    if _, err := f.WriteString("ticket 7\n"); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
```

```
    got, err := io.ReadAll(f)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    fmt.Printf("read %d bytes\n", len(got))
}
```

Run:

```
go run forget_close.go
```

Output:

```
read 0 bytes
```

The cursor is at the end. Nothing to read. Fix: `f.Seek(0, io.SeekStart)` before `ReadAll`, or `Close` and `os.Open` as in Case 4. The same class of bug: never calling `Close` on a `bufio.Writer` (data stuck in the buffer). Call `Flush` or `Close` on the wrapper.

The boring rule

- Talk to `io.Reader` / `io.Writer`, not to a concrete file, unless you need file-only methods.
- `io.Copy` for whole streams. `bufio.Scanner` for lines.
- Check `Scanner.Err`. Treat `io.EOF` as the end, not as a crash.
- `CreateTemp` in programs that should not hard-code a path.
- Close writers; check the error. `defer f.Close()` is right for reads; for writes, close explicitly if the next step depends on the bytes being there.
- Do not `ReadAll` a stream you can process incrementally.

Try this

1. In `copy_tickets.go`, copy to `os.Stdout` instead of a buffer. Drop the `bytes` print or keep a `io.TeeReader` if you still want the count.
2. In `scan.go`, add a blank line in the middle of the string. See that `Scan` still yields an empty `Text()`.
3. In `temp_order.go`, print `len(got)` as well as the string.
4. Fix `forget_close.go` with `f.Seek(0, io.SeekStart)` before `ReadAll`. Confirm you read 9 bytes.

Encoding and Serialization

Encoding and Serialization

The desk already has structs. Disk and the network want bytes. The boring default is **`encoding/json` with struct tags** for objects, **`encoding/csv`** for tables, and **`encoding/binary`** only when you are writing a fixed-width number. Do not invent a private text format if one of these three fits.

Mental model

JSON maps object keys to struct fields through **tags**: `` `json:"id"` ``. Exported fields are included. Unexported fields are omitted — silently. `omitempty` drops zeros and empty strings. `json.Marshal` produces bytes. `json.Unmarshal` fills a pointer.

CSV is rows of strings. `encoding/csv` quotes fields that need it. You still decide what each column means.

`encoding/binary` writes integers in big-endian or little-endian form. Use it for a length prefix or a packed id, not for a ticket with a note.

Unknown JSON keys are ignored by default. Missing keys leave the field at its zero value. That is convenient and easy to miss.

Worked examples

Case 1: JSON marshal and unmarshal

Save as `ticket_json.go`.

```
// ticket_json.go
package main

import (
    "encoding/json"
    "fmt"
)

type Ticket struct {
    ID    int    `json:"id"`
    Table int    `json:"table"`
    Note  string `json:"note"`
```

```
}

func main() {
    t := Ticket{ID: 7, Table: 12, Note: "no onions"}
    raw, err := json.Marshal(t)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(string(raw))

    var got Ticket
    if err := json.Unmarshal(raw, &got); err != nil {
        fmt.Println(err)
        return
    }
    fmt.Printf("id=%d table=%d note=%q\n", got.ID, got.Table, got.Note)
}
```

Run:

```
go run ticket_json.go
```

Output:

```
{"id":7,"table":12,"note":"no onions"}
id=7 table=12 note="no onions"
```

`Marshal` needs a value. `Unmarshal` needs a pointer. Field order in the JSON follows the struct.

Case 2: Tags that rename and omit

Save as `ticket_tags.go`. The Go field is `Table`; the JSON key is `table_n`. Empty notes disappear.

```
// ticket_tags.go
package main

import (
    "encoding/json"
    "fmt"
)

type Ticket struct {
    ID    int    `json:"id"`
    Table int    `json:"table_n"`
```

```
    Note  string `json:"note,omitempty"`
}

func main() {
    a, err := json.Marshal(Ticket{ID: 7, Table: 12, Note: "no onions"})
    if err != nil {
        fmt.Println(err)
        return
    }
    b, err := json.Marshal(Ticket{ID: 8, Table: 3})
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(string(a))
    fmt.Println(string(b))
}
```

Run:

```
go run ticket_tags.go
```

Output:

```
{"id":7,"table_n":12,"note":"no onions"}
{"id":8,"table_n":3}
```

`table: 0` would still appear: `0` is omitted only if you add `omitempty` on `Table`. Decide per field.

Case 3: CSV

Save as `tickets_csv.go`. Header row, two tickets, stdout.

```
// tickets_csv.go
package main

import (
    "encoding/csv"
    "fmt"
    "os"
    "strings"
)

func main() {
    var b strings.Builder
    w := csv.NewWriter(&b)
```

```
    if err := w.Write([]string{"id", "table", "note"}); err != nil {
        fmt.Println(err)
        return
    }
    if err := w.Write([]string{"7", "12", "no onions"}); err != nil {
        fmt.Println(err)
        return
    }
    if err := w.Write([]string{"8", "3", "extra napkins"}); err != nil {
        fmt.Println(err)
        return
    }
    w.Flush()
    if err := w.Error(); err != nil {
        fmt.Println(err)
        return
    }

    r := csv.NewReader(strings.NewReader(b.String()))
    rows, err := r.ReadAll()
    if err != nil {
        fmt.Println(err)
        return
    }
    for i, row := range rows {
        if i == 0 {
            continue
        }
        fmt.Printf("ticket %s table %s (%s)\n", row[0], row[1], row[2])
    }
    fmt.Fprint(os.Stdout, b.String())
}
```

Run:

```
go run tickets_csv.go
```

Output:

```
ticket 7 table 12 (no onions)
ticket 8 table 3 (extra napkins)
id,table,note
7,12,no onions
8,3,extra napkins
```

`Flush` before you read the builder. `ReadAll` is fine for a small sheet. For a large file, `Read` in a loop.

Case 4: Binary id

Save as `ticket_id.go`. Four bytes, big-endian.

```
// ticket_id.go
package main

import (
    "encoding/binary"
    "fmt"
)

func main() {
    var buf [4]byte
    binary.BigEndian.PutUint32(buf[:], 7)
    fmt.Printf("%x\n", buf)
    id := binary.BigEndian.Uint32(buf[:])
    fmt.Println(id)
}
```

Run:

```
go run ticket_id.go
```

Output:

```
00000007
7
```

That is the whole trick. A note string does not belong here. If you need a self-describing blob, use JSON.

The trap

Save as `silent_json.go`. Lowercase fields are invisible to `encoding/json`.

```
// silent_json.go
package main

import (
    "encoding/json"
    "fmt"
)

type ticket struct {
    id    int
```

```
    Table int `json:"table"`
}

func main() {
    raw, err := json.Marshal(ticket{id: 7, Table: 12})
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(string(raw))
}
```

Run:

```
go run silent_json.go
```

Output:

```
{"table":12}
```

No error. The id is gone. The fix is an exported field (`ID`) and a tag. The same silence happens if you unmarshal into a struct with the wrong key names: zeros, no complaint. Compare against a golden JSON in a test when the payload matters.

The boring rule

- JSON for structured records. Tags on every exported field you intend to ship.
- Export the field. Tags do not override unexported.
- CSV for rectangular tables of strings. Flush. Check `Error`.
- `binary` for integers you control both ends of. Pick an endian and keep it.
- `Unmarshal` into a pointer. Check the error. Then check that required fields are non-zero if zero would be wrong.
- Do not `fmt.Sprintf` your own JSON.

Try this

1. In `ticket_json.go`, unmarshal `{"id":7}` into a `Ticket` and print `Table` and `Note`. They are zero.
2. Add `` `json:"table,omitempty"` `` to Case 2's `Table` and marshal `Ticket{ID: 1}`. Confirm `table_n` is gone — then remember you renamed the key; the tag must say `table_n,omitempty`.
3. In `tickets_csv.go`, put a comma inside a note (`no onions, extra ice`). Confirm the writer quotes the field and the reader still returns one cell.
4. In `ticket_id.go`, write `7` with `binary.LittleEndian` and print `%x`. Compare with the big-endian line.

HTTP and Networking

HTTP and Networking

`net/http` is the client and the server. The boring default is a **`ServeMux` with method-and-path patterns** (Go 1.22+), a handler that writes a status and a body, and tests against `httptest.Server` so the process **exits**. `ListenAndServe` is what you run in production. It is not what you run in a `go run` example unless you shut it down.

Mental model

A client sends a request (`http.Get`, `http.NewRequest` + `Client.Do`) and must close `resp.Body`. A server registers handlers on a mux. Go 1.22 patterns look like `"GET /tickets/{id}"`. `r.PathValue("id")` is the wildcard.

`httptest.NewServer(h)` listens on a local port, gives you `srv.URL`, and `srv.Close()` shuts it down. That is the shape of every complete program in this chapter that you are expected to run.

`http.ListenAndServe(addr, mux)` blocks forever (or until a fatal error). Use it in `main` of a real service, with `http.Server` and `Shutdown` when you have a signal. Do not leave it as the only example you type.

Worked examples

Case 1: httptest server and GET pattern

Save as `ticket_get.go`. The server runs, the client fetches `/tickets/7`, both finish, the process exits.

```
// ticket_get.go
package main

import (
    "fmt"
    "io"
    "net/http"
    "net/http/httptest"
)

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /tickets/{id}", func(w http.ResponseWriter, r *http.Request) {
```

```
        id := r.PathValue("id")
        fmt.Fprintf(w, "ticket %s\n", id)
    })
    srv := httptest.NewServer(mux)
    defer srv.Close()

    resp, err := srv.Client().Get(srv.URL + "/tickets/7")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer resp.Body.Close()
    body, err := io.ReadAll(resp.Body)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println("status", resp.StatusCode)
    fmt.Print(string(body))
}
```

Run:

```
go run ticket_get.go
```

Output:

```
status 200
ticket 7
```

`srv.Client()` talks to this server and ignores HTTP proxies. Prefer it over `http.Get` in listings. The `{id}` wildcard only matches a single path segment. `GET` in the pattern means a `POST` will not hit this handler.

Case 2: POST and a method miss

Save as `ticket_post.go`. Register `POST /tickets`. A GET to the same path is 405.

```
// ticket_post.go
package main

import (
    "fmt"
    "io"
    "net/http"
    "net/http/httptest"
```

```
    "strings"
)

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("POST /tickets", func(w http.ResponseWriter, r *http.Request) {
        raw, err := io.ReadAll(r.Body)
        if err != nil {
            http.Error(w, err.Error(), http.StatusBadRequest)
            return
        }
        fmt.Fprintf(w, "accepted %s\n", raw)
    })
    srv := httptest.NewServer(mux)
    defer srv.Close()
    c := srv.Client()

    resp, err := c.Post(srv.URL+"/tickets", "text/plain", strings.NewReader("7"))
    if err != nil {
        fmt.Println(err)
        return
    }
    body, err := io.ReadAll(resp.Body)
    resp.Body.Close()
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println("POST", resp.StatusCode, strings.TrimSpace(string(body)))

    bad, err := c.Get(srv.URL + "/tickets")
    if err != nil {
        fmt.Println(err)
        return
    }
    io.Copy(io.Discard, bad.Body)
    bad.Body.Close()
    fmt.Println("GET", bad.StatusCode)
}
```

Run:

```
go run ticket_post.go
```

Output:

```
POST 200 accepted 7
GET 405
```

405 means the path exists for another method. 404 means no pattern matched.

Case 3: Client with a timeout

Save as `client_timeout.go`. A `Client` is not the default. It has a timeout so a stuck server cannot hold `main` forever.

```
// client_timeout.go
package main

import (
    "fmt"
    "net/http"
    "net/http/httptest"
    "time"
)

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /slow", func(w http.ResponseWriter, r *http.Request) {
        time.Sleep(50 * time.Millisecond)
        fmt.Fprintln(w, "done")
    })
    srv := httptest.NewServer(mux)
    defer srv.Close()

    c := srv.Client()
    c.Timeout = 10 * time.Millisecond
    _, err := c.Get(srv.URL + "/slow")
    fmt.Println(err != nil)
}
```

Run:

```
go run client_timeout.go
```

Output:

```
true
```

The error text includes a URL and `context deadline exceeded` (wording can include the client timeout). Checking `err != nil` is the stable part. Raise `Timeout` above 50ms and you get `false`.

Case 4: httptest.NewRecorder without a listener

Save as `handler_only.go`. When you only need to test a handler, skip the server.

```go
// handler_only.go
package main

import (
    "fmt"
    "net/http"
    "net/http/httptest"
)

func ticket(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "ticket %s\n", r.PathValue("id"))
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /tickets/{id}", ticket)
    req := httptest.NewRequest("GET", "/tickets/7", nil)
    rec := httptest.NewRecorder()
    mux.ServeHTTP(rec, req)
    fmt.Println(rec.Code)
    fmt.Print(rec.Body.String())
}
```

Run:

```
go run handler_only.go
```

Output:

```
200
ticket 7
```

No port, no goroutine. Use this in unit tests. Use `NewServer` when you want the real client stack.

The trap

`ListenAndServe` never returns on success. This program is complete and you should **not** leave it running as your only check.

```go
// hang.go
package main

import (
    "fmt"
    "net/http"
)
```

```
func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /health", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "ok")
    })
    fmt.Println("listening on :8080")
    http.ListenAndServe("127.0.0.1:8080", mux)
}
```

If you run it, it prints `listening on :8080` and waits. Stop it with Ctrl-C. The production shape that still **exits** in a listing is `http.Server` plus `Shutdown`:

```
// shutdown.go
package main

import (
    "context"
    "fmt"
    "io"
    "net"
    "net/http"
    "time"
)

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /health", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "ok")
    })
    ln, err := net.Listen("tcp", "127.0.0.1:0")
    if err != nil {
        fmt.Println(err)
        return
    }
    srv := &http.Server{Handler: mux}
    go srv.Serve(ln)

    resp, err := http.Get("http://" + ln.Addr().String() + "/health")
    if err != nil {
        fmt.Println(err)
        return
    }
    body, _ := io.ReadAll(resp.Body)
    resp.Body.Close()
    fmt.Print(string(body))
```

```
    ctx, cancel := context.WithTimeout(context.Background(), time.Second)
    defer cancel()
    if err := srv.Shutdown(ctx); err != nil {
        fmt.Println(err)
    }
}
```

Run:

```
go run shutdown.go
```

Output:

```
ok
```

Prefer Case 1 (`httptest`) for everyday examples. Use `Shutdown` when you are showing a real `http.Server`.

The boring rule

- Patterns: `"GET /tickets/{id}"`, not a manual method check unless you have to.
- `PathValue` for wildcards.
- Close the response body. Use a `Client` with a `Timeout`.
- `httptest.Server` or `NewRecorder` in programs and tests that must exit.
- `ListenAndServe` in production `main`, with `Shutdown` on signal.
- Do not ignore `ListenAndServe`'s error (`http.ErrServerClosed` after `Shutdown` is expected).

Try this

1. In `ticket_get.go`, request `/tickets/7/extra`. Print the status (404).
2. Add `mux.HandleFunc("GET /tickets/{id}/note", ...)` and fetch that path.
3. In `client_timeout.go`, set `Timeout` to `200 * time.Millisecond` and print the body when `err == nil`.
4. In `handler_only.go`, send a `POST` with `NewRequest`. Print `rec.Code` (405).

Logging and Observability

Logging and Observability

The standard logger for new Go is `log/slog`. The boring default is a `TextHandler` on a developer machine and a `JSONHandler` in a service, with **levels** and **attributes** instead of interpolated sentences. `fmt.Println` is for examples. It is not a log.

Mental model

A **handler** writes records. A **logger** holds a handler plus default attributes. `slog.Info`, `slog.Warn`, `slog.Error`, `slog.Debug` are the levels you use in application code. `Debug` is silent unless the handler's level includes it.

Attributes are key-value pairs: `slog.Int("id", 7)`, `slog.String("window", "A")`. They survive JSON. A sentence like `"printed ticket 7 at table 12"` does not query well.

`slog.New(handler)` then `logger.Info(...)`. `logger.With(...)` returns a child with extra attributes on every line (the window name, the request id).

This chapter strips timestamps with `ReplaceAttr` so the printed output is stable. Production logs **keep** `time`. Do not copy the strip into a service.

Worked examples

Case 1: Text handler

Save as `slog_text.go`.

```
// slog_text.go
package main

import (
    "log/slog"
    "os"
)

func noTime(_ []string, a slog.Attr) slog.Attr {
    if a.Key == slog.TimeKey {
        return slog.Attr{}
    }
    return a
```

```
}

func main() {
    h := slog.NewTextHandler(os.Stdout, &slog.HandlerOptions{ReplaceAttr: noTime})
    log := slog.New(h)
    log.Info("ticket printed", slog.Int("id", 7), slog.Int("table", 12))
}
```

Run:

```
go run slog_text.go
```

Output:

```
level=INFO msg="ticket printed" id=7 table=12
```

Keys are stable. The message is a short event name, not a paragraph.

Case 2: JSON handler

Save as `slog_json.go`. Same record, machine-shaped.

```
// slog_json.go
package main

import (
    "log/slog"
    "os"
)

func noTime(_ []string, a slog.Attr) slog.Attr {
    if a.Key == slog.TimeKey {
        return slog.Attr{}
    }
    return a
}

func main() {
    h := slog.NewJSONHandler(os.Stdout, &slog.HandlerOptions{ReplaceAttr: noTime})
    log := slog.New(h)
    log.Info("ticket printed", slog.Int("id", 7), slog.Int("table", 12))
}
```

Run:

```
go run slog_json.go
```

Output:

```
{"level":"INFO","msg":"ticket printed","id":7,"table":12}
```

Ship this shape to a collector. Do not wrap it in another JSON envelope.

Case 3: Levels

Save as `slog_level.go`. Debug is off. Info is on. Error is on.

```
// slog_level.go
package main

import (
    "log/slog"
    "os"
)

func noTime(_ []string, a slog.Attr) slog.Attr {
    if a.Key == slog.TimeKey {
        return slog.Attr{}
    }
    return a
}

func main() {
    h := slog.NewTextHandler(os.Stdout, &slog.HandlerOptions{
        Level:       slog.LevelInfo,
        ReplaceAttr: noTime,
    })
    log := slog.New(h)
    log.Debug("prep detail", slog.Int("id", 7))
    log.Info("ticket printed", slog.Int("id", 7))
    log.Error("kitchen closed", slog.Int("table", 12))
}
```

Run:

```
go run slog_level.go
```

Output:

```
level=INFO msg="ticket printed" id=7
level=ERROR msg="kitchen closed" table=12
```

Set `Level: slog.LevelDebug` when you are on the desk debugging. Leave Info in production unless you have a reason. `Error` is for failures that already have an `error` value — add `slog.Any("err", err)`.

Case 4: With attributes

Save as `slog_with.go`. The window name is on every line without repeating it.

```
// slog_with.go
package main

import (
    "log/slog"
    "os"
)

func noTime(_ []string, a slog.Attr) slog.Attr {
    if a.Key == slog.TimeKey {
        return slog.Attr{}
    }
    return a
}

func main() {
    h := slog.NewTextHandler(os.Stdout, &slog.HandlerOptions{ReplaceAttr: noTime})
    log := slog.New(h).With(slog.String("window", "A"))
    log.Info("clocked in")
    log.Info("ticket printed", slog.Int("id", 7))
}
```

Run:

```
go run slog_with.go
```

Output:

```
level=INFO msg="clocked in" window=A
level=INFO msg="ticket printed" window=A id=7
```

Bind request-scoped fields with `With` at the start of a handler. Do not use `context.Value` as your log bag.

Case 5: Default logger

Save as `slog_default.go`. `slog.SetDefault` makes package-level `slog.Info` use your handler.

```
// slog_default.go
package main

import (
    "log/slog"
    "os"
)

func noTime(_ []string, a slog.Attr) slog.Attr {
    if a.Key == slog.TimeKey {
        return slog.Attr{}
    }
    return a
}

func main() {
    h := slog.NewTextHandler(os.Stdout, &slog.HandlerOptions{ReplaceAttr: noTime})
    slog.SetDefault(slog.New(h))
    slog.Info("desk open", slog.Int("tables", 12))
}
```

Run:

```
go run slog_default.go
```

Output:

```
level=INFO msg="desk open" tables=12
```

Call `SetDefault` once in `main`. Libraries should accept a `*slog.Logger` argument instead of assuming the default.

The trap

Save as `println_log.go`. This looks like logging and is not.

```
// println_log.go
package main

import "fmt"

func main() {
    id := 7
    table := 12
    fmt.Println("printed ticket", id, "at table", table)
}
```

Run:

```
go run println_log.go
```

Output:

```
printed ticket 7 at table 12
```

No level, no keys, no timestamp, no way to filter "table 12" without grepping English. The `log` package (`log.Printf`) is the old standard library logger. It is still in the tree. New code in this book uses `slog`.

A second trap: logging `err.Error()` as the message and dropping the `error` value. Prefer `log.Error("kitchen closed", slog.Any("err", err))` so handlers can keep the type.

The boring rule

- `log/slog` is the default. Text locally, JSON in services.
- Event name in `msg`. Facts in attributes.
- Levels: Debug off in production, Info for normal events, Error for failures.
- `With` for fields that apply to a whole call.
- Keep timestamps in real logs. Strip them only in book listings and in tests that compare strings.
- Do not `fmt.Println` in library or server code to "log."

Try this

1. In `slog_level.go`, set `Level: slog.LevelDebug` and confirm the prep line appears.
2. In `slog_json.go`, add `slog.String("note", "no onions")`. Confirm a new JSON key.
3. In `slog_with.go`, create `log.With(slog.Int("id", 7))` and call `Info("printed")`. The id should be on the line.
4. Change `noTime` so it also drops `slog.LevelKey`. Run `slog_text.go`. See that `level=INFO` disappears — then put the level back. You want it.

Strings and Strconv

Strings and Strconv

`strings` transforms and inspects string values. `strconv` converts between strings and numbers. The boring default is: **use the `strings` package instead of hand-rolling loops, and always check the `error` from `strconv.Atoi` or `strconv.ParseFloat`**. A string is immutable bytes in Go; mutating a "string" means building a new one.

Mental model

A Go `string` is a read-only byte slice with a known length. Index `s[i]` gives a byte, not a rune. `range s` gives runes. `len(s)` is bytes. `utf8.RuneCountInString(s)` is code points.

`strings.Contains`, `strings.HasPrefix`, `strings.HasSuffix` test membership. `strings.Split` and `strings.Join` are inverses. `strings.TrimSpace` removes leading and trailing whitespace. `strings.ReplaceAll` replaces all occurrences. `strings.ToUpper` / `strings.ToLower` fold case. `strings.Builder` builds incrementally without repeated allocation.

`strconv.Atoi(s)` parses a decimal integer and returns `(int, error)`. `strconv.Itoa(n)` formats one. `strconv.ParseFloat`, `strconv.ParseBool`, `strconv.ParseInt` handle the other base types. `strconv.FormatFloat` and friends go the other direction.

Worked examples

Case 1: Inspect and transform

Save as `note_tidy.go`. A ticket note arrives with trailing whitespace and mixed case. Normalize it.

```
// note_tidy.go
package main

import (
    "fmt"
    "strings"
)

func main() {
    raw := "  NO Onions  "
    clean := strings.TrimSpace(raw)
    lower := strings.ToLower(clean)
```

```
    fmt.Println(clean)
    fmt.Println(lower)
    fmt.Println(strings.HasPrefix(lower, "no"))
    fmt.Println(strings.Contains(lower, "onion"))
}
```

Run:

```
go run note_tidy.go
```

Output:

```
NO Onions
no onions
true
true
```

`TrimSpace` is the first step when reading user input. Test the cleaned version, not the raw one.

Case 2: Split and Join

Save as `tag_split.go`. Tags arrive comma-separated. Join them back with a different separator.

```
// tag_split.go
package main

import (
    "fmt"
    "strings"
)

func main() {
    line := "vegan,gluten-free,no-nuts"
    tags := strings.Split(line, ",")
    fmt.Println(len(tags))
    for i, t := range tags {
        fmt.Println(i, t)
    }
    fmt.Println(strings.Join(tags, " | "))
}
```

Run:

```
go run tag_split.go
```

Output:

```
3
0 vegan
1 gluten-free
2 no-nuts
vegan | gluten-free | no-nuts
```

`Split` on an empty string returns one element (the whole string). `strings.Fields` splits on any whitespace and drops empty tokens — use it for space-separated input instead of `Split(" ")`.

Case 3: strings.Builder

Save as `receipt.go`. Build a multi-line receipt without string concatenation.

```go
// receipt.go
package main

import (
    "fmt"
    "strings"
)

type Item struct {
    Name  string
    Price int
}

func receipt(items []Item) string {
    var b strings.Builder
    b.WriteString("--- receipt ---\n")
    total := 0
    for _, item := range items {
        fmt.Fprintf(&b, "%-15s %3d\n", item.Name, item.Price)
        total += item.Price
    }
    fmt.Fprintf(&b, "%-15s %3d\n", "total", total)
    return b.String()
}

func main() {
    items := []Item{
        {"soup", 8},
        {"coffee", 4},
        {"cake", 6},
    }
    fmt.Print(receipt(items))
}
```

Run:

```
go run receipt.go
```

Output:

```
--- receipt ---
soup              8
coffee            4
cake              6
total            18
```

`strings.Builder` is the right tool when you are assembling with `fmt.Fprintf`. `bytes.Buffer` works too, but `Builder` cannot be used as a `Reader` — it is write-only. Use `Buffer` if downstream code needs an `io.Reader`.

Case 4: strconv.Atoi and Itoa

Save as `ticket_parse.go`. A ticket ID arrives as a string from a URL parameter. Parse it; reject junk.

```
// ticket_parse.go
package main

import (
    "fmt"
    "strconv"
)

func parseID(s string) (int, error) {
    return strconv.Atoi(s)
}

func main() {
    good, err := parseID("42")
    if err != nil {
        fmt.Println("error:", err)
        return
    }
    fmt.Println("id", good)

    _, err = parseID("four")
    fmt.Println("parse error:", err)

    fmt.Println(strconv.Itoa(good + 1))
}
```

Run:

```
go run ticket_parse.go
```

Output:

```
id 42
parse error: strconv.Atoi: parsing "four": invalid syntax
43
```

`Atoi` is `ParseInt(s, 10, 0)` — base 10, fit in an `int`. When you need a specific width or base, use `ParseInt` directly. The error is always non-nil on failure; you do not need a separate "valid" bool.

Case 5: ParseFloat and FormatFloat

Save as `price_parse.go`. A price arrives as a decimal string. Parse it, add tax, format back.

```
// price_parse.go
package main

import (
    "fmt"
    "strconv"
)

func main() {
    raw := "12.50"
    price, err := strconv.ParseFloat(raw, 64)
    if err != nil {
        fmt.Println(err)
        return
    }
    withTax := price * 1.10
    formatted := strconv.FormatFloat(withTax, 'f', 2, 64)
    fmt.Println("price:", price)
    fmt.Println("with tax:", formatted)
}
```

Run:

```
go run price_parse.go
```

Output:

```
price: 12.5
with tax: 13.75
```

`'f'` means decimal, no exponent. `2` is the number of digits after the decimal. `64` matches the `float64` storage. For money in a real desk system, use integer cents or a fixed-precision type — not `float64` arithmetic for the canonical value.

The trap

Save as `count_rune.go`. `len` counts bytes, not characters. A ticket note in UTF-8 may have fewer runes than bytes.

```
// count_rune.go
package main

import (
    "fmt"
    "unicode/utf8"
)

func main() {
    note := "jalapeño"
    fmt.Println("bytes :", len(note))
    fmt.Println("runes :", utf8.RuneCountInString(note))
    for i, r := range note {
        fmt.Printf("i=%d r=%c\n", i, r)
    }
}
```

Run:

```
go run count_rune.go
```

Output:

```
bytes : 9
runes : 8
i=0 r=j
i=1 r=a
i=2 r=l
i=3 r=a
i=4 r=p
i=5 r=e
i=6 r=ñ
i=8 r=o
```

The index jumps from 6 to 8 because `ñ` is two bytes. `s[i]` at index 7 would be the second byte of `ñ` — not a valid code point. Use `range` or `utf8.DecodeRuneInString` to walk code points. Slice by byte only when you know you are dealing with ASCII.

The boring rule

- `strings.TrimSpace` before comparing user input.
- `strings.Fields` for whitespace-separated tokens; `strings.Split` for a fixed separator.
- `strings.Builder` for incremental assembly; `strings.Join` when you already have a slice.
- `strconv.Atoi` / `Itoa` for decimal integers. `ParseFloat` / `FormatFloat` for floats. Always check the error.
- `len` is bytes. `utf8.RuneCountInString` is code points. Use `range` to iterate.
- Do not `+` concatenate in a loop; use `Builder`.

Try this

1. In `note_tidy.go`, use `strings.ReplaceAll` to replace all spaces with underscores before printing.
2. In `tag_split.go`, use `strings.Fields` on `"  vegan  no-nuts  "` and print the resulting slice length.
3. In `ticket_parse.go`, try `parseID("-5")` — print the result. Negative IDs parse fine; you must reject them with a range check.
4. In `price_parse.go`, change `'f'` to `'e'` in `FormatFloat`. Print the result and notice the scientific notation.

Sorting and the slices Package

Sorting and the slices Package

The Go 1.21 `slices` package is the boring default for everything that used to need `sort.Slice`. `slices.Sort`, `slices.SortFunc`, `slices.Contains`, `slices.Index`, `slices.Compact`, and `slices.Reverse` cover the common desk operations without closures or ceremony. `sort` is still correct; `slices` is just shorter.

Mental model

`slices.Sort` sorts a `[]T` in place when `T` is ordered (numbers, strings). `slices.SortFunc` takes a comparison function for structs. `slices.Contains` scans linearly — fine for ten tickets, slow for ten thousand. `slices.BinarySearch` requires a sorted slice and finds in `O(log n)`.

`sort.Slice(s, less)` is the pre-1.21 equivalent. It still works. `sort.Search` is a generic binary search on indices. Both `sort` and `slices` sort in place; neither returns a new slice.

`slices.Compact` removes adjacent duplicates (from a sorted slice). `slices.Reverse` reverses in place. `slices.Max` / `slices.Min` scan for the extreme value — also require an ordered element type.

Worked examples

Case 1: Sort numbers and strings

Save as `sort_tickets.go`. Sort a ticket list by ID, then sort a tag list alphabetically.

```
// sort_tickets.go
package main

import (
    "fmt"
    "slices"
)

func main() {
    ids := []int{12, 3, 7, 1, 9}
    slices.Sort(ids)
    fmt.Println("ids:", ids)

    tags := []string{"vegan", "allergy", "birthday", "cold"}
```

```
    slices.Sort(tags)
    fmt.Println("tags:", tags)
}
```

Run:

```
go run sort_tickets.go
```

Output:

```
ids: [1 3 7 9 12]
tags: [allergy birthday cold vegan]
```

`slices.Sort` modifies the original slice. There is no sorted copy returned. If you need the original order, copy the slice first: `ids2 := slices.Clone(ids)`.

Case 2: SortFunc for structs

Save as `sort_orders.go`. Sort open orders by price, descending.

```
// sort_orders.go
package main

import (
    "cmp"
    "fmt"
    "slices"
)

type Order struct {
    ID    int
    Table int
    Price int
}

func main() {
    orders := []Order{
        {ID: 7, Table: 3, Price: 22},
        {ID: 8, Table: 1, Price: 8},
        {ID: 9, Table: 2, Price: 15},
    }
    slices.SortFunc(orders, func(a, b Order) int {
        return cmp.Compare(b.Price, a.Price) // descending: b before a
    })
    for _, o := range orders {
        fmt.Printf("id=%d table=%d price=%d\n", o.ID, o.Table, o.Price)
```

```
    }
}
```

Run:

```
go run sort_orders.go
```

Output:

```
id=7 table=3 price=22
id=9 table=2 price=15
id=8 table=1 price=8
```

`cmp.Compare(a, b)` returns -1, 0, or 1. Swap the arguments to reverse. Do not return `a.Price - b.Price` — integer overflow is possible on large values; use `cmp.Compare` instead.

Case 3: Contains and Index

Save as `find_tag.go`. Check whether a tag is present; find where it sits.

```
// find_tag.go
package main

import (
    "fmt"
    "slices"
)

func main() {
    tags := []string{"vegan", "birthday", "allergy"}
    fmt.Println(slices.Contains(tags, "birthday"))
    fmt.Println(slices.Contains(tags, "spicy"))

    i := slices.Index(tags, "birthday")
    fmt.Println("index of birthday:", i)
    i = slices.Index(tags, "spicy")
    fmt.Println("index of spicy:", i)
}
```

Run:

```
go run find_tag.go
```

Output:

```
true
false
index of birthday: 1
index of spicy: -1
```

`Index` returns -1 when not found. `Contains` is a linear scan. If you need repeated fast lookups, use a `map[string]bool`.

Case 4: BinarySearch on a sorted slice

Save as `bsearch_id.go`. The ticket IDs are sorted. Find one by value.

```go
// bsearch_id.go
package main

import (
    "fmt"
    "slices"
)

func main() {
    ids := []int{1, 3, 7, 9, 12, 15}
    // slices.BinarySearch returns (index, found)
    i, ok := slices.BinarySearch(ids, 9)
    fmt.Printf("found=%v at index=%d\n", ok, i)

    i, ok = slices.BinarySearch(ids, 10)
    fmt.Printf("found=%v insertion-point=%d\n", ok, i)
}
```

Run:

```
go run bsearch_id.go
```

Output:

```
found=true at index=3
found=false insertion-point=4
```

When `found` is false, `i` is the insertion point — the index where you would insert the missing value to keep the slice sorted. Binary search requires the slice to already be sorted; calling it on an unsorted slice gives a wrong answer with no error.

Case 5: Compact and Reverse

Save as `dedup.go`. Remove duplicate tags after sorting; then reverse the list.

```
// dedup.go
package main

import (
    "fmt"
    "slices"
)

func main() {
    tags := []string{"vegan", "birthday", "vegan", "allergy", "birthday"}
    slices.Sort(tags)
    fmt.Println("sorted:", tags)
    tags = slices.Compact(tags)
    fmt.Println("deduped:", tags)
    slices.Reverse(tags)
    fmt.Println("reversed:", tags)
}
```

Run:

```
go run dedup.go
```

Output:

```
sorted: [allergy birthday birthday vegan vegan]
deduped: [allergy birthday vegan]
reversed: [vegan birthday allergy]
```

`Compact` only removes **adjacent** duplicates. Sort first, then compact. `Reverse` is in place; there is no `ReversedClone` — copy first if you want both orders.

The trap

Save as `sort_trap.go`. Comparing a float slice with `sort.Slice` and a hand-written less function that subtracted floats would quietly produce wrong order on NaN or near-equal values. Use `cmp.Compare` instead.

```
// sort_trap.go
package main

import (
    "cmp"
    "fmt"
    "slices"
)
```

```
func main() {
    prices := []float64{8.5, 3.0, 8.5, 15.0, 3.0}
    // wrong: manual subtraction loses NaN safety
    // slices.SortFunc(prices, func(a, b float64) int { return int(a - b) })
    // right:
    slices.SortFunc(prices, cmp.Compare[float64])
    fmt.Println(prices)
}
```

Run:

```
go run sort_trap.go
```

Output:

```
[3 3 8.5 8.5 15]
```

`cmp.Compare` handles NaN deterministically (NaN is less than everything). The subtraction trick is the classic C mistake that Go does not need.

The boring rule

- Prefer `slices.Sort` / `slices.SortFunc` over `sort.Slice` on Go 1.21+.
- Use `cmp.Compare` in sort functions instead of subtraction.
- `slices.Contains` is linear; use a map for repeated lookups.
- `slices.BinarySearch` requires a sorted slice. Sort first, search after.
- `slices.Compact` removes adjacent duplicates; always sort before compacting.
- `slices.Clone` when you need the original after sorting.

Try this

1. In `sort_orders.go`, add a second sort by `Table` number ascending using `slices.SortFunc` and `cmp.Compare`.
2. In `find_tag.go`, convert `tags` to a `map[string]bool` and benchmark the lookup mentally — no benchmark tool needed, just think about whether the map changes big-O.
3. In `dedup.go`, skip the sort and call `slices.Compact` directly. Print the result. Notice that only adjacent duplicates are removed.
4. In `bsearch_id.go`, add `42` to `ids` without re-sorting, call `BinarySearch(ids, 12)`, and observe the wrong result. Then re-sort and try again.

OS, Args, and Flags

OS, Args, and Flags

`os` is the thin layer between Go and the operating system. `flag` is the standard-library argument parser. The boring default is: **`flag` for named flags, `os.Args[1:]` only when flags are too much, `os.Getenv` for configuration, and `os.Exit(1)` on a fatal error** — after printing to `os.Stderr`, not `os.Stdout`.

Mental model

`os.Args` is the raw `[]string` of the command line, `os.Args[0]` being the program name. `flag` wraps that slice with typed bindings: `flag.String`, `flag.Int`, `flag.Bool`. Call `flag.Parse()` once; after that the pointers hold the parsed values.

`os.Getenv("KEY")` returns a string; an absent key returns `""`. `os.LookupEnv("KEY")` returns `(value, present)` when you must distinguish absent from empty.

`os.Exit(code)` terminates the process immediately — deferred calls do not run. Reserve it for `main` and for top-level error paths where cleanup is not relevant. Libraries must never call it.

`os.Stdin`, `os.Stdout`, `os.Stderr` are open `*os.File` values. Pass them into functions as `io.Reader` or `io.Writer` so tests can substitute a `bytes.Buffer`.

Worked examples

Case 1: os.Args

Save as `raw_args.go`. Print every argument after the program name.

```
// raw_args.go
package main

import (
    "fmt"
    "os"
)

func main() {
    if len(os.Args) < 2 {
        fmt.Fprintln(os.Stderr, "usage: raw_args <name>")
        os.Exit(1)
```

```
    }
    for i, a := range os.Args[1:] {
        fmt.Printf("arg %d: %s\n", i, a)
    }
}
```

Run:

```
go run raw_args.go soup coffee cake
```

Output:

```
arg 0: soup
arg 1: coffee
arg 2: cake
```

`os.Args[1:]` skips the program name. When no arguments arrive, print a usage message on `os.Stderr` and exit 1. Exit 0 is success; any other code signals failure to the shell.

Case 2: flag — named flags

Save as `open_desk.go`. The desk operator names a window and a maximum table count.

```
// open_desk.go
package main

import (
    "flag"
    "fmt"
    "os"
)

func main() {
    window := flag.String("window", "A", "desk window label")
    maxTables := flag.Int("tables", 10, "maximum tables to seat")
    flag.Parse()

    if *window == "" {
        fmt.Fprintln(os.Stderr, "window must not be empty")
        os.Exit(1)
    }
    fmt.Printf("opening window %s with %d tables\n", *window, *maxTables)
}
```

Run:

```
go run open_desk.go -window B -tables 14
```

Output:

```
opening window B with 14 tables
```

`flag.String` returns a pointer. Dereference with `*window`. The `flag` package writes usage to `os.Stderr` and exits 2 on a parse error — you do not need to catch that yourself.

To see the generated help:

```
go run open_desk.go -help
```

```
Usage of /tmp/…:
  -tables int
        maximum tables to seat (default 10)
  -window string
        desk window label (default "A")
```

Case 3: flag.Args() — positional after flags

Save as `print_tickets.go`. Flags come first; ticket IDs follow as positional arguments.

```
// print_tickets.go
package main

import (
    "flag"
    "fmt"
    "os"
)

func main() {
    window := flag.String("window", "A", "desk window")
    flag.Parse()
    ids := flag.Args() // everything after the last flag
    if len(ids) == 0 {
        fmt.Fprintln(os.Stderr, "provide at least one ticket id")
        os.Exit(1)
    }
    for _, id := range ids {
        fmt.Printf("[%s] printing ticket %s\n", *window, id)
    }
}
```

Run:

```
go run print_tickets.go -window B 7 8 9
```

Output:

```
[B] printing ticket 7
[B] printing ticket 8
[B] printing ticket 9
```

`flag.Args()` returns the non-flag arguments after `flag.Parse()`. Flags must precede positional arguments or the parser stops at the first non-flag token.

Case 4: os.Getenv and os.LookupEnv

Save as `env_config.go`. The desk zone comes from the environment; an absent key is an error.

```
// env_config.go
package main

import (
    "fmt"
    "os"
)

func main() {
    zone, ok := os.LookupEnv("DESK_ZONE")
    if !ok {
        fmt.Fprintln(os.Stderr, "DESK_ZONE not set")
        os.Exit(1)
    }
    fmt.Println("zone:", zone)

    lang := os.Getenv("LANG")
    if lang == "" {
        lang = "en_US.UTF-8"
    }
    fmt.Println("lang:", lang)
}
```

Run:

```
DESK_ZONE=EU go run env_config.go
```

Output:

```
zone: EU
lang: en_US.UTF-8
```

`LookupEnv` is correct when an empty-string value is valid but an absent key is not. `Getenv` returning `""` cannot tell you which case you are in. Use it only when the default for both absent and empty is the same.

Case 5: Stderr and structured errors

Save as `desk_cmd.go`. A command that writes its log to stderr and its product to stdout so they can be piped independently.

```
// desk_cmd.go
package main

import (
    "flag"
    "fmt"
    "os"
    "strconv"
)

func run(w *string, ids []string, out, errOut *os.File) int {
    for _, raw := range ids {
        n, err := strconv.Atoi(raw)
        if err != nil {
            fmt.Fprintf(errOut, "invalid id %q: %v\n", raw, err)
            return 1
        }
        fmt.Fprintf(out, "[%s] ticket %d ready\n", *w, n)
    }
    return 0
}

func main() {
    window := flag.String("window", "A", "desk window")
    flag.Parse()
    code := run(window, flag.Args(), os.Stdout, os.Stderr)
    os.Exit(code)
}
```

Run:

```
go run desk_cmd.go -window C 7 8 bad
```

Output:

```
[C] ticket 7 ready
[C] ticket 8 ready
invalid id "bad": strconv.Atoi: parsing "bad": invalid syntax
```

Exit code is 1. The caller's shell can check `$?`. Separating stdout (data) and stderr (log/error) lets the caller pipe the data without the errors mixing in.

The trap

Save as `exit_defer.go`. `os.Exit` skips deferred calls.

```
// exit_defer.go
package main

import (
    "fmt"
    "os"
)

func main() {
    defer fmt.Println("this never runs")
    fmt.Println("before exit")
    os.Exit(1)
}
```

Run:

```
go run exit_defer.go
```

Output:

```
before exit
```

Exit code 1, and `defer` did not fire. This is by design. Do not rely on deferred cleanup in a path that calls `os.Exit`. Return error values up to `main`, do the cleanup, then exit.

The boring rule

- `flag` for named options. `flag.Args()` for positional arguments after flags.
- `os.LookupEnv` when absence and empty are different. `os.Getenv` when both map to the same default.
- Errors go to `os.Stderr`. Program output goes to `os.Stdout`. They are different.
- `os.Exit(1)` only in `main` (or the outermost `run` function). Libraries return errors.
- Deferred calls do not run after `os.Exit`. Structure cleanup before you exit.

Try this

1. In `open_desk.go`, add a `-verbose` bool flag. When true, print extra detail (any extra line you like).
2. In `print_tickets.go`, add validation: reject any ticket id that is not a number (use `strconv.Atoi`).
3. In `env_config.go`, run without `DESK_ZONE=`. Observe the exit message on stderr.
4. In `desk_cmd.go`, move the `os.Exit` into a separate `func main()` that calls `run` and exits with its return value. Confirm that removing `os.Exit` from `run` lets you test `run` without killing the process.

Regular Expressions

Regular Expressions

`regexp` compiles a pattern once and matches it against many strings. The boring default is: **compile at init time with `regexp.MustCompile`, match once with `FindString` or `FindStringSubmatch`, and never use `regexp` when `strings.Contains` or `strings.HasPrefix` would do**. Regexps are powerful and slow relative to plain string checks.

Mental model

Go regexps follow RE2 syntax (no backtracking, no lookahead). `regexp.Compile(pattern)` returns `(*Regexp, error)`. `regexp.MustCompile(pattern)` panics on a bad pattern — only use it at package level or in `init`, where a bad pattern is a programmer error, not a runtime error.

`MatchString` tells you whether the pattern appears anywhere in the string. `FindString` returns the first match. `FindAllString` returns all matches. `FindStringSubmatch` returns the whole match plus capture groups. `ReplaceAllString` substitutes all matches with a replacement string.

A **capturing group** is `(...)`. A **non-capturing group** is `(?:...)`. A **named group** is `(?P<name>...)`.

Worked examples

Case 1: Compile and MatchString

Save as `ticket_pattern.go`. Validate that a ticket reference looks like `T-` followed by one or more digits.

```
// ticket_pattern.go
package main

import (
    "fmt"
    "regexp"
)

var ticketRef = regexp.MustCompile(`^T-\d+$`)

func main() {
    cases := []string{"T-7", "T-42", "ticket7", "T-", "T-7x"}
    for _, c := range cases {
```

```
        fmt.Printf("%-8s %v\n", c, ticketRef.MatchString(c))
    }
}
```

Run:

```
go run ticket_pattern.go
```

Output:

```
T-7      true
T-42     true
ticket7  false
T-       false
T-7x     false
```

`^` anchors to the start; `$` to the end. Without them, `T-7x` would match because `T-7` is found inside it. `\d+` is one or more decimal digit. The raw string literal `` `…` `` avoids double-escaping backslashes.

Case 2: FindString and FindAllString

Save as `extract_ids.go`. A kitchen log has ticket IDs embedded in free text. Pull them all out.

```
// extract_ids.go
package main

import (
    "fmt"
    "regexp"
)

var idPat = regexp.MustCompile(`T-\d+`)

func main() {
    log := "fired T-7 at 12:00, T-8 delayed, retry T-7 at 12:05"
    first := idPat.FindString(log)
    fmt.Println("first:", first)

    all := idPat.FindAllString(log, -1) // -1 = no limit
    fmt.Println("all:", all)
}
```

Run:

```
go run extract_ids.go
```

Output:

```
first: T-7
all: [T-7 T-8 T-7]
```

`-1` as the count means return all. Pass `n` to stop after `n` matches. `FindAllString` returns `nil` if there are no matches — test with `len(all) == 0`, not a nil check directly.

Case 3: FindStringSubmatch — capture groups

Save as `parse_line.go`. Each order line has a table number and a ticket number. Capture both.

```
// parse_line.go
package main

import (
    "fmt"
    "regexp"
)

var linePat = regexp.MustCompile(`table (\d+), ticket (\d+)`)

func main() {
    line := "table 12, ticket 7"
    m := linePat.FindStringSubmatch(line)
    if m == nil {
        fmt.Println("no match")
        return
    }
    // m[0] = whole match, m[1] = first group, m[2] = second group
    fmt.Println("whole  :", m[0])
    fmt.Println("table  :", m[1])
    fmt.Println("ticket :", m[2])
}
```

Run:

```
go run parse_line.go
```

Output:

```
whole  : table 12, ticket 7
table  : 12
ticket : 7
```

Capture groups are 1-indexed in the result slice. `m[0]` is always the full match. If a group is optional and did not match, its slot is `""`.

Case 4: Named groups

Save as `named_groups.go`. Named groups make the indexing self-documenting.

```
// named_groups.go
package main

import (
    "fmt"
    "regexp"
)

var orderPat = regexp.MustCompile(`(?P<table>\d+)/(?P<ticket>\d+)`)

func main() {
    ref := "3/7"
    m := orderPat.FindStringSubmatch(ref)
    if m == nil {
        fmt.Println("no match")
        return
    }
    names := orderPat.SubexpNames() // ["", "table", "ticket"]
    result := make(map[string]string)
    for i, name := range names {
        if i != 0 && name != "" {
            result[name] = m[i]
        }
    }
    fmt.Println("table:", result["table"])
    fmt.Println("ticket:", result["ticket"])
}
```

Run:

```
go run named_groups.go
```

Output:

```
table: 3
ticket: 7
```

`SubexpNames` returns the group names in the same order as the submatch slice. The loop pattern above is the standard way to build a `map[string]string` from named groups. Go does not have a built-in named-group map helper.

Case 5: ReplaceAllString

Save as `redact.go`. A printed receipt should mask ticket notes that contain allergen warnings.

```
// redact.go
package main

import (
    "fmt"
    "regexp"
)

var allergenPat = regexp.MustCompile(`(?i)(allerg\w+|intoleran\w+)`)

func main() {
    note := "no onions; ALLERGY: nuts; intolerance to gluten"
    masked := allergenPat.ReplaceAllString(note, "[REDACTED]")
    fmt.Println(note)
    fmt.Println(masked)
}
```

Run:

```
go run redact.go
```

Output:

```
no onions; ALLERGY: nuts; intolerance to gluten
no onions; [REDACTED]: nuts; [REDACTED] to gluten
```

`(?i)` at the start makes the whole pattern case-insensitive. `\w+` is one or more word characters. The replacement string `"[REDACTED]"` is literal; use `$1` to refer back to a captured group.

The trap

Save as `compile_loop.go`. Compiling the same pattern inside a loop is the common performance mistake.

```
// compile_loop.go
package main

import (
    "fmt"
    "regexp"
)
```

```
func badMatch(s string) bool {
    // compiles on every call - do not do this
    r := regexp.MustCompile(`T-\d+`)
    return r.MatchString(s)
}

var goodPat = regexp.MustCompile(`T-\d+`) // compiled once

func goodMatch(s string) bool {
    return goodPat.MatchString(s)
}

func main() {
    tickets := []string{"T-7", "T-8", "T-9"}
    for _, t := range tickets {
        fmt.Println(t, goodMatch(t))
    }
    _ = badMatch // shown for contrast only
}
```

Run:

```
go run compile_loop.go
```

Output:

```
T-7 true
T-8 true
T-9 true
```

`MustCompile` at package level means the pattern is compiled exactly once when the binary loads. `Compile` inside a function allocates and parses the automaton on every call. In a tight loop or a request handler, that is the whole cost of the regexp, repeated pointlessly.

The boring rule

- `regexp.MustCompile` at package level. `regexp.Compile` when the pattern itself might be invalid at runtime.
- Anchor with `^` and `$` when you want a full-string match.
- Prefer `strings.Contains` / `strings.HasPrefix` for simple checks — they are faster and clearer.
- `FindStringSubmatch` for groups. `m[0]` is the whole match; `m[1]` onward are groups.
- Named groups with `(?P<name>...)` make indexed submatch slices readable.
- Never compile inside a loop or a handler.

Try this

1. In `ticket_pattern.go`, change `\d+` to `\d{1,5}` (one to five digits). Test `T-123456` — it should now return false.
2. In `extract_ids.go`, change the limit from `-1` to `2`. Print the result — only the first two IDs appear.
3. In `parse_line.go`, extend the pattern to also capture an optional note: `table (\d+), ticket (\d+)(?:, (.+))?`. Print `m[3]` for both a line with a note and one without.
4. In `redact.go`, change the replacement to `"[$1]"`. Confirm that the original word appears inside brackets.

The maps Package

The maps Package

The `maps` package provides generic functions over maps—everything you used to write with a `for range`.

Mental model

`maps` provides generic functions over maps. `maps.Keys` returns a slice of keys in unspecified order. `maps.Values` returns values. `maps.Clone` makes a shallow copy. `maps.Copy` merges src into dst (overwrites on conflict). `maps.Equal` compares two maps element-by-element. `maps.DeleteFunc` removes entries where the function returns true. Import is `"maps"`.

Worked examples

Case 1: maps.Clone

Clone a `map[string]int` price menu, mutate the clone, and show the original is unchanged.

```
// clone.go
package main

import (
    "fmt"
    "maps"
)

func main() {
    menu := map[string]int{
        "coffee": 250,
        "tea":    200,
    }

    clone := maps.Clone(menu)
    clone["coffee"] = 300
    clone["pastry"] = 450

    fmt.Printf("Original: %v\n", menu)
    fmt.Printf("Clone:    %v\n", clone)
}
```

Run:

```
go run clone.go
```

Output:

```
Original: map[coffee:250 tea:200]
Clone:    map[coffee:300 pastry:450 tea:200]
```

Case 2: maps.Keys and sorting

Get sorted keys for a seating chart using `slices.Sort` on the result for deterministic output.

```
// keys.go
package main

import (
    "fmt"
    "maps"
    "slices"
)

func main() {
    seats := map[string]string{
        "T1": "Alice",
        "T3": "Bob",
        "T2": "Charlie",
    }

    // maps.Keys returns an iterator; slices.Sorted collects and sorts in one step
    keys := slices.Sorted(maps.Keys(seats))

    fmt.Printf("Sorted desks: %v\n", keys)
}
```

Run:

```
go run keys.go
```

Output:

```
Sorted desks: [T1 T2 T3]
```

Case 3: maps.Copy

Merge two shift rosters `map[string]string` (name→window), showing a duplicate key is overwritten.

```
// copy.go
package main

import (
    "fmt"
    "maps"
)

func main() {
    roster := map[string]string{
        "Alice": "Morning",
        "Bob":   "Evening",
    }

    updates := map[string]string{
        "Bob":     "Night",
        "Charlie": "Morning",
    }

    // src (updates) overwrites dst (roster)
    maps.Copy(roster, updates)

    fmt.Printf("Roster: %v\n", roster)
}
```

Run:

```
go run copy.go
```

Output:

```
Roster: map[Alice:Morning Bob:Night Charlie:Morning]
```

Case 4: maps.Equal

Compare two ticket priority maps, true then mutate one and false.

```
// equal.go
package main

import (
    "fmt"
```

```
    "maps"
)

func main() {
    t1 := map[string]int{
        "T-100": 1,
        "T-101": 2,
    }
    t2 := map[string]int{
        "T-100": 1,
        "T-101": 2,
    }

    fmt.Printf("Equal initially? %v\n", maps.Equal(t1, t2))

    t2["T-102"] = 3
    fmt.Printf("Equal after mutation? %v\n", maps.Equal(t1, t2))
}
```

Run:

```
go run equal.go
```

Output:

```
Equal initially? true
Equal after mutation? false
```

Case 5: maps.DeleteFunc

Remove all menu items with price < 100 (cents).

```
// delete.go
package main

import (
    "fmt"
    "maps"
)

func main() {
    menu := map[string]int{
        "coffee": 250,
        "tea":    200,
        "water":  0,
        "mint":   50,
```

```
    }

    maps.DeleteFunc(menu, func(k string, v int) bool {
        return v < 100
    })

    fmt.Printf("Menu: %v\n", menu)
}
```

Run:

```
go run delete.go
```

Output:

```
Menu: map[coffee:250 tea:200]
```

The trap

Trap 1: `maps.Clone` is a shallow clone

`maps.Clone` creates a copy of the map, but it copies **values as-is**. If the map stores pointers or references to structs (`map[string]*Ticket`), both maps point to the exact same underlying heap objects. Mutating a field through the clone mutates the original!

Save as `shallow_trap.go`:

```
// shallow_trap.go
package main

import (
    "fmt"
    "maps"
)

type Ticket struct {
    ID     int
    Status string
}

func main() {
    original := map[string]*Ticket{
        "T-1": {ID: 101, Status: "open"},
    }

    clone := maps.Clone(original)
```

```
    // Mutating the pointed-to object changes both maps:
    clone["T-1"].Status = "closed"

    fmt.Println("original:", original["T-1"].Status)
    fmt.Println("clone:   ", clone["T-1"].Status)
}
```

Run:

```
go run shallow_trap.go
```

Output:

```
original: closed
clone:    closed
```

If you need a deep clone of pointer values, allocate a new struct for each key explicitly.

Trap 2: Assuming map iteration order is stable

`maps.Keys` returns an iterator over keys in randomized hash order. If you assume keys will always iterate in insertion or sorted order, tests and serialization logic will behave flakily across runs and Go compiler versions. Always use `slices.Sorted(maps.Keys(m))` if order matters.

The boring rule

- Use `maps.Clone` for simple value maps (`map[string]int`, `map[string]string`). For pointer values, clone structs explicitly.
- Use `slices.Sorted(maps.Keys(m))` to iterate keys deterministically.
- Use `maps.Copy(dst, src)` to merge maps instead of manual loops.
- Use `maps.DeleteFunc` for in-place conditional eviction (pruning expired cache entries, clearing completed orders).
- Use `maps.Equal` for direct map comparisons.

Try this

1. In `keys.go`, use `slices.Sorted(maps.Values(seats))` to get a sorted list of guest names.
2. In `shallow_trap.go`, fix the aliasing by constructing a new `&Ticket{ID: t.ID, Status: t.Status}` inside a copy loop. Confirm the original remains `"open"`.
3. In `delete.go`, change the predicate to keep only items whose names start with `"c"`.

Iterators and Range Functions

Iterators and Range Functions

Since Go 1.23, you can use `for range` over functions, enabling standard iterators for any custom collection or stream.

Mental model

Since Go 1.23, `range` can iterate over a function with signature `func(yield func(V) bool)` (which is `iter.Seq[V]`) or `func(yield func(K, V) bool)` (which is `iter.Seq2[K,V]`). The function calls `yield` for each element; if `yield` returns `false`, iteration stops early (e.g. a `break`). `iter.Seq[V]` and `iter.Seq2[K,V]` are type aliases in the `iter` package for these signatures. `slices.Collect(seq)` drains a Seq into a slice. `maps.Collect(seq2)` drains a Seq2 into a map.

Worked examples

Case 1: iter.Seq[int]

A `Tickets` function returns an iterator over a slice. The closure calls `yield` per ID.

```
// tickets.go
package main

import (
    "fmt"
    "iter"
)

func Tickets(ids []int) iter.Seq[int] {
    return func(yield func(int) bool) {
        for _, id := range ids {
            if !yield(id) {
                return
            }
        }
    }
}

func main() {
    for id := range Tickets([]int{7, 8, 9}) {
```

```
        fmt.Printf("Processing ticket %d\n", id)
    }
}
```

Run:

```
go run tickets.go
```

Output:

```
Processing ticket 7
Processing ticket 8
Processing ticket 9
```

Case 2: iter.Seq2[string, int]

`Prices` yields key/value pairs from a menu map.

```
// prices.go
package main

import (
    "fmt"
    "iter"
)

func Prices(menu map[string]int) iter.Seq2[string, int] {
    return func(yield func(string, int) bool) {
        for item, price := range menu {
            if !yield(item, price) {
                return
            }
        }
    }
}

func main() {
    menu := map[string]int{
        "coffee": 250,
        "tea":    200,
    }

    for item, price := range Prices(menu) {
        fmt.Printf("%s: %dc\n", item, price)
    }
}
```

Run:

```
go run prices.go
```

Output:

```
coffee: 250c
tea: 200c
```

Case 3: Early termination

An iterator that yields tables with free capacity. The caller breaks after finding the first one.

```
// tables.go
package main

import (
    "fmt"
    "iter"
)

func AllOpen(tables []int, capacity int) iter.Seq[int] {
    return func(yield func(int) bool) {
        for _, seats := range tables {
            if seats >= capacity {
                if !yield(seats) {
                    return
                }
            }
        }
    }
}

func main() {
    tables := []int{2, 1, 4, 6, 2}

    for table := range AllOpen(tables, 4) {
        fmt.Printf("Found table with %d seats. Booking it!\n", table)
        break // This will cause yield to return false
    }
}
```

Run:

```
go run tables.go
```

Output:

```
Found table with 4 seats. Booking it!
```

Case 4: slices.Collect

Collect the Tickets iterator into a `[]int` slice using the `slices` package.

```go
// collect.go
package main

import (
    "fmt"
    "iter"
    "slices"
)

func Tickets(ids []int) iter.Seq[int] {
    return func(yield func(int) bool) {
        for _, id := range ids {
            if !yield(id) {
                return
            }
        }
    }
}

func main() {
    seq := Tickets([]int{101, 102, 103})

    // Drain the iterator into a slice
    all := slices.Collect(seq)

    fmt.Printf("Collected tickets: %v\n", all)
}
```

Run:

```
go run collect.go
```

Output:

```
Collected tickets: [101 102 103]
```

Case 5: Composing iterators

A `Filter` that wraps another iterator.

```go
// filter.go
package main

import (
    "fmt"
    "iter"
    "slices"
)

func Tickets(ids []int) iter.Seq[int] {
    return func(yield func(int) bool) {
        for _, id := range ids {
            if !yield(id) {
                return
            }
        }
    }
}

func Filter[V any](seq iter.Seq[V], keep func(V) bool) iter.Seq[V] {
    return func(yield func(V) bool) {
        for v := range seq {
            if keep(v) {
                if !yield(v) {
                    return
                }
            }
        }
    }
}

func main() {
    seq := Tickets([]int{1, 2, 3, 4, 5})

    evens := Filter(seq, func(v int) bool {
        return v%2 == 0
    })

    fmt.Printf("Evens: %v\n", slices.Collect(evens))
}
```

Run:

```
go run filter.go
```

Output:

```
Evens: [2 4]
```

The trap

Calling `yield` after it returned `false`. Always `return` immediately when `yield` returns `false`; failing to do so is a bug and the runtime might panic or exhibit weird behavior.

```
// BAD pattern:
func BadSeq() iter.Seq[int] {
    return func(yield func(int) bool) {
        yield(1)
        yield(2) // BUG: if yield(1) returned false, this shouldn't be called!
    }
}

// GOOD pattern:
func GoodSeq() iter.Seq[int] {
    return func(yield func(int) bool) {
        if !yield(1) { return }
        if !yield(2) { return }
    }
}
```

The boring rule

Use standard `iter.Seq` and `iter.Seq2` when you want to abstract over a collection or stream. Always check the return value of `yield` and return immediately if it is `false`.

Try this

Write an `iter.Seq2[int, string]` function called `Enumerate` that wraps a `iter.Seq[string]` and yields `(0, val)`, `(1, val)`, etc. Use it to print a numbered list of ticket statuses.

Context and Process Signals

Context and Process Signals

Services need to shut down cleanly when an operator presses Ctrl+C or Kubernetes sends a termination signal. The standard library handles this via `os/signal` and `signal.NotifyContext`. The boring default is: **derive your root context from `signal.NotifyContext(context.Background(), os.Interrupt, syscall.SIGTERM)`, pass it down, and give in-flight desk orders a brief graceful shutdown window**.

Mental model

In Go, cancellation flows down through `context.Context`. OS processes receive termination signals (`os.Interrupt` / `SIGINT` on Ctrl+C, `syscall.SIGTERM` in containers/systemd).

`signal.NotifyContext` attaches listening for those OS signals to a context. When a signal arrives, the context's `Done()` channel closes, and `context.Cause(ctx)` describes the signal.

A graceful shutdown pattern: 1. Listen for signals with `signal.NotifyContext`. 2. When canceled, stop accepting new desk tickets or HTTP requests. 3. Drain ongoing work using a clean shutdown timeout. 4. Exit cleanly with return code 0.

Worked examples

Case 1: signal.NotifyContext

Save as `signal_stop.go`. In this testable example, we simulate signal context creation and immediate stop.

```
// signal_stop.go
package main

import (
    "context"
    "fmt"
    "os"
    "os/signal"
    "syscall"
)

func main() {
    ctx, stop := signal.NotifyContext(context.Background(), os.Interrupt, syscall.SIGTERM)
```

```
    defer stop()

    fmt.Println("signal listener ready")
    fmt.Println("canceled:", ctx.Err() != nil)
}
```

Run:

```
go run signal_stop.go
```

Output:

```
signal listener ready
canceled: false
```

Calling `stop()` unregisters the signal handler and restores default OS behavior. Always `defer stop()`.

Case 2: Clean worker loop on signal cancellation

Save as `desk_shift_signal.go`. A worker processes tickets until interrupted.

```
// desk_shift_signal.go
package main

import (
    "context"
    "fmt"
    "time"
)

func processTickets(ctx context.Context) {
    ticker := time.NewTicker(20 * time.Millisecond)
    defer ticker.Stop()

    ticketID := 1
    for {
        select {
        case <-ctx.Done():
            fmt.Printf("shift ending cleanly: %v\n", ctx.Err())
            return
        case <-ticker.C:
            fmt.Printf("processed ticket #%d\n", ticketID)
            ticketID++
            if ticketID > 3 {
                return
```

```
            }
        }
    }
}

func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 100*time.Millisecond)
    defer cancel()

    processTickets(ctx)
}
```

Run:

```
go run desk_shift_signal.go
```

Output:

```
processed ticket #1
processed ticket #2
processed ticket #3
```

By checking `ctx.Done()` inside `select`, the loop exits as soon as either time expires, a signal arrives, or the queue empties.

Case 3: Graceful HTTP server shutdown

Save as `graceful_http.go`. Start an HTTP server and shut it down cleanly with a context.

```
// graceful_http.go
package main

import (
    "context"
    "fmt"
    "net/http"
    "net/http/httptest"
    "time"
)

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /health", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "desk operational")
    })
```

```
    srv := httptest.NewServer(mux)
    defer srv.Close()

    // In production, you would run srv.Shutdown(shutdownCtx)
    // Here we verify the handler responds before shutdown
    resp, err := http.Get(srv.URL + "/health")
    if err != nil {
        fmt.Println("error:", err)
        return
    }
    defer resp.Body.Close()

    fmt.Println("status:", resp.StatusCode)

    // Simulate graceful drain
    shutdownCtx, cancel := context.WithTimeout(context.Background(), 500*time.Millisecond)
    defer cancel()

    fmt.Println("shutdown completed within deadline:", shutdownCtx.Err() == nil)
}
```

Run:

```
go run graceful_http.go
```

Output:

```
status: 200
shutdown completed within deadline: true
```

`srv.Shutdown(ctx)` stops accepting new connections and waits for active requests to finish before returning.

Case 4: Asynchronous cleanup with `context.AfterFunc`

Save as `after_func.go`. Starting in Go 1.21+, `context.AfterFunc(ctx, f)` registers a callback that runs in its own goroutine the instant `ctx` is canceled. It is cleaner and cheaper than spinning up a dedicated watcher goroutine with `select { case <-ctx.Done(): ... }`.

```
// after_func.go
package main

import (
    "context"
    "fmt"
    "time"
```

```
)

func main() {
    ctx, cancel := context.WithCancel(context.Background())

    // Register an asynchronous cleanup callback triggered on cancellation
    stop := context.AfterFunc(ctx, func() {
        fmt.Println("async cleanup: releasing desk locks")
    })
    defer stop()

    fmt.Println("shift active")
    cancel() // Trigger context cancellation

    // Brief pause to allow the background AfterFunc goroutine to execute
    time.Sleep(10 * time.Millisecond)
    fmt.Println("shift closed")
}
```

Run:

```
go run after_func.go
```

Output:

```
shift active
async cleanup: releasing desk locks
shift closed
```

Calling the returned `stop()` function unregisters the callback if the operation finishes successfully before cancellation occurs.

The trap

Save as `instant_exit.go`. Calling `os.Exit(0)` immediately upon receiving a signal terminates the runtime without running deferred cleanups or finishing in-flight database transactions.

```
// instant_exit.go
package main

import (
    "fmt"
)

func cleanup() {
    fmt.Println("closing desk cash register")
```

```
}

func main() {
    defer cleanup()

    // Pretend a signal was caught and someone called os.Exit
    fmt.Println("signal received")
    // If os.Exit(1) were called here, 'cleanup' would be skipped!
    // Instead, let main return naturally.
}
```

Run:

```
go run instant_exit.go
```

Output:

```
signal received
closing desk cash register
```

Do not call `os.Exit` inside signal handlers. Cancel the context, allow functions to return, execute `defer` blocks, and exit cleanly through `main`.

The boring rule

- Wrap process entry with `signal.NotifyContext(context.Background(), os.Interrupt, syscall.SIGTERM)`.
- Always `defer stop()` to reset signal handling behavior.
- Propagate `ctx` to all workers and background tasks.
- For HTTP servers, catch signal cancel and call `server.Shutdown(shutdownCtx)`.
- Avoid `os.Exit` on shutdown; allow `defer` statements to clean up resources.

Try this

1. In `signal_stop.go`, print `context.Cause(ctx)` after calling `stop()`.
2. In `desk_shift_signal.go`, reduce timeout to `10 * time.Millisecond` and watch the clean context cancellation message appear.
3. Extend `graceful_http.go` with a second route `GET /orders` and verify its response.

Hashing and Cryptography

Hashing and Cryptography

Checksums, integrity hashes, and secure tokens are standard requirements for desks and services. Go provides cryptographic primitives in `crypto/*`. The boring default is: **`crypto/sha256` for integrity hashes and content digests, and `crypto/rand` for secure random tokens or IDs**. Never use `math/rand` for secrets, and never hand-roll security logic.

Mental model

`crypto/sha256` implements SHA-256 (a one-way hash function). `sha256.Sum256([]byte)` calculates a 32-byte fixed checksum. For larger payloads or streaming files, instantiate `sha256.New()` and write data into it like any other `io.Writer`.

`crypto/rand` provides cryptographically secure random bytes from the operating system (`/dev/urandom` or system entropy). Read from `crypto/rand.Reader` using `io.ReadFull`.

Constant-time comparison with `crypto/subtle.ConstantTimeCompare` prevents timing attacks when comparing sensitive tokens, signatures, or HMAC digests.

Worked examples

Case 1: Compute a SHA-256 digest

Save as `order_digest.go`. Calculate the checksum of an order receipt to detect tampering.

```
// order_digest.go
package main

import (
    "crypto/sha256"
    "fmt"
)

func main() {
    payload := []byte("ticket: 7, table: 12, total: 34.50")
    hash := sha256.Sum256(payload)

    fmt.Printf("sha256: %x\n", hash)
}
```

Run:

```
go run order_digest.go
```

Output:

```
sha256: 49206d2fe2b78b0f948f98ec839a9c2b4e85746b1981e4b52df39c1b3f60fec7
```

`%x` formats the 32-byte array as a lowercase hex string. Changing even one byte in the payload completely changes the hash.

Case 2: Streaming hash with io.Writer

Save as `stream_hash.go`. When processing a large ledger or file, do not load everything into memory.

```
// stream_hash.go
package main

import (
    "crypto/sha256"
    "fmt"
    "io"
    "strings"
)

func main() {
    r := strings.NewReader("line 1: shift open\nline 2: ticket 7 printed\n")
    hasher := sha256.New()

    n, err := io.Copy(hasher, r)
    if err != nil {
        fmt.Println("error:", err)
        return
    }

    digest := hasher.Sum(nil)
    fmt.Printf("bytes hashed: %d\n", n)
    fmt.Printf("digest: %x\n", digest)
}
```

Run:

```
go run stream_hash.go
```

Output:

```
bytes hashed: 44
digest: df8503eebe66574f19b8ea007fa7b9f3d511979b06222b07e742813137887e07
```

`hasher.Sum(nil)` appends the calculated checksum to the provided slice. Passing `nil` returns a fresh slice with the digest.

Case 3: Cryptographically secure random tokens

Save as `secure_token.go`. Generate an unguessable ticket confirmation code.

```go
// secure_token.go
package main

import (
    "crypto/rand"
    "encoding/hex"
    "fmt"
    "io"
)

func generateToken(n int) (string, error) {
    bytes := make([]byte, n)
    if _, err := io.ReadFull(rand.Reader, bytes); err != nil {
        return "", err
    }
    return hex.EncodeToString(bytes), nil
}

func main() {
    token, err := generateToken(16) // 16 bytes = 32 hex chars
    if err != nil {
        fmt.Println("error:", err)
        return
    }
    fmt.Println("token length:", len(token))
    fmt.Println("is hex:", len(token) == 32)
}
```

Run:

```
go run secure_token.go
```

Output:

```
token length: 32
is hex: true
```

Always check errors from `io.ReadFull(rand.Reader, ...)`. If OS entropy is exhausted or unavailable, it will fail safely instead of generating predictable data.

Case 4: Constant-time comparison for tokens

Save as `verify_token.go`. Avoid timing side-channel attacks when authenticating desk tokens.

```
// verify_token.go
package main

import (
    "crypto/subtle"
    "fmt"
)

func main() {
    expectedToken := []byte("secr3t-d3sk-tok3n")
    userToken := []byte("secr3t-d3sk-tok3n")
    badToken := []byte("wrong-d3sk-tok3n!")

    valid1 := subtle.ConstantTimeCompare(expectedToken, userToken) == 1
    valid2 := subtle.ConstantTimeCompare(expectedToken, badToken) == 1

    fmt.Printf("user token match: %v\n", valid1)
    fmt.Printf("bad token match: %v\n", valid2)
}
```

Run:

```
go run verify_token.go
```

Output:

```
user token match: true
bad token match: false
```

Plain `bytes.Equal` or string `==` returns early on the first mismatched byte, leaking information about prefix correctness through response timing. `subtle.ConstantTimeCompare` takes the same time regardless of mismatch position.

The trap

Save as `insecure_rand.go`. Using `math/rand` (or `math/rand/v2`) for API keys or auth tokens is predictable because pseudo-random number generators can be reverse-engineered from their output.

```go
// insecure_rand.go
package main

import (
    "fmt"
    "math/rand/v2"
)

func main() {
    // Good for games, bad for security
    orderCode := rand.IntN(1000)
    fmt.Printf("desk order code: %03d\n", orderCode)
    fmt.Println("use crypto/rand for security tokens!")
}
```

Run:

```
go run insecure_rand.go
```

Output:

```
desk order code: 512
use crypto/rand for security tokens!
```

Reserve `math/rand` for simulations, shuffle algorithms, and non-security random numbers. For any credential, token, or session ID, use `crypto/rand`.

The boring rule

- `crypto/sha256` is the go-to standard digest algorithm for checksums and content deduplication.
- `crypto/rand` is mandatory for tokens, passwords, keys, and UUID generation.
- `subtle.ConstantTimeCompare` prevents timing attacks on secrets and signatures.
- For streaming files, pass `hash.Hash` into `io.Copy`.
- Never invent custom encryption, padding, or hash algorithms.

Try this

1. In `order_digest.go`, modify one character in `payload` and observe how completely the output hash changes.
2. In `secure_token.go`, generate a 32-byte token and print its length in hex characters (should be 64).
3. Compare two byte slices of different lengths using `subtle.ConstantTimeCompare`. Confirm it returns 0.

Filesystem and path/filepath

Filesystem and path/filepath

Manipulating directories, reading files, and resolving paths are foundational for desk utilities and servers. Go provides cross-platform path handling in `path/filepath` and filesystem utilities in `os`. The boring default is: **always use `filepath` (not string concatenation with `"/"`), use `os.ReadFile` and `os.WriteFile` for small files, and walk trees with `filepath.WalkDir`.**

Mental model

Windows uses `\` as a path separator, while Linux/macOS use `/`. If you join paths with `dir + "/" + file`, your code breaks on Windows. `filepath.Join` handles OS-specific path separators automatically.

`filepath.Clean` resolves `.` and `..` components and strips redundant slashes. `filepath.Rel` computes relative paths between two targets. `filepath.WalkDir` iterates through a directory hierarchy efficiently, skipping stat calls when possible (superior to legacy `filepath.Walk`).

Use `os.ReadFile(path)` and `os.WriteFile(path, data, perm)` for self-contained file operations when streaming is unnecessary.

Worked examples

Case 1: Cross-platform filepath.Join and Clean

Save as `desk_paths.go`. Clean and join file paths safely.

```
// desk_paths.go
package main

import (
    "fmt"
    "path/filepath"
)

func main() {
    p := filepath.Join("reports", "2026", "september", "..", "august", "orders.csv")
    fmt.Println("joined & cleaned:", filepath.ToSlash(p))
    fmt.Println("base filename:", filepath.Base(p))
    fmt.Println("parent directory:", filepath.ToSlash(filepath.Dir(p)))
    fmt.Println("extension:", filepath.Ext(p))
```

```
}
```

Run:

```
go run desk_paths.go
```

Output:

```
joined & cleaned: reports/2026/august/orders.csv
base filename: orders.csv
parent directory: reports/2026/august
extension: .csv
```

`filepath.ToSlash` normalizes slashes to `/` (helpful for platform-neutral logs and output). `filepath.Base` extracts the last element; `filepath.Dir` extracts the directory path.

Case 2: Reading and writing files atomically with os

Save as `file_ops.go`. Write a file, verify its existence, read it back, and remove it cleanly.

```
// file_ops.go
package main

import (
    "fmt"
    "os"
    "path/filepath"
)

func main() {
    dir, err := os.MkdirTemp("", "desk-files-*")
    if err != nil {
        fmt.Println("err:", err)
        return
    }
    defer os.RemoveAll(dir)

    target := filepath.Join(dir, "shift_notes.txt")
    content := []byte("morning shift: all tables cleared\n")

    if err := os.WriteFile(target, content, 0644); err != nil {
        fmt.Println("write err:", err)
        return
    }

    data, err := os.ReadFile(target)
```

```
    if err != nil {
        fmt.Println("read err:", err)
        return
    }

    fmt.Print(string(data))
}
```

Run:

```
go run file_ops.go
```

Output:

```
morning shift: all tables cleared
```

`os.WriteFile` truncates and writes the file in one shot with requested permissions. `os.MkdirTemp` creates a dedicated sandbox for temporary files.

Case 3: Walking directory trees with filepath.WalkDir

Save as `walk_desk.go`. Traverse nested folders and inspect file entries.

```
// walk_desk.go
package main

import (
    "fmt"
    "io/fs"
    "os"
    "path/filepath"
)

func main() {
    tmp, err := os.MkdirTemp("", "desk-tree-*")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer os.RemoveAll(tmp)

    // Create nested files
    os.MkdirAll(filepath.Join(tmp, "archive", "sub"), 0755)
    os.WriteFile(filepath.Join(tmp, "orders.txt"), []byte("ok"), 0644)
    os.WriteFile(filepath.Join(tmp, "archive", "receipt.csv"), []byte("ok"), 0644)
```

```
    var files []string
    err = filepath.WalkDir(tmp, func(path string, d fs.DirEntry, err error) error {
        if err != nil {
            return err
        }
        if !d.IsDir() {
            rel, _ := filepath.Rel(tmp, path)
            files = append(files, filepath.ToSlash(rel))
        }
        return nil
    })

    if err != nil {
        fmt.Println("walk err:", err)
        return
    }

    fmt.Println("found files:", len(files))
    for _, f := range files {
        fmt.Println("file:", f)
    }
}
```

Run:

```
go run walk_desk.go
```

Output:

```
found files: 2
file: archive/receipt.csv
file: orders.txt
```

`fs.DirEntry` supplies metadata without triggering unnecessary `stat` calls on every path, making `WalkDir` fast on large trees.

The trap

Save as `slash_concat.go`. Concatenating file paths with string additions or `/` creates broken paths on Windows and fails to handle boundary slashes properly.

```
// slash_concat.go
package main

import (
    "fmt"
```

```
    "path/filepath"
)

func main() {
    dir := "records/"
    file := "/ticket.json"

    // Wrong: manual string concat results in redundant slash
    badPath := dir + "/" + file

    // Right: filepath.Join sanitizes and handles separators correctly
    goodPath := filepath.Join(dir, file)

    fmt.Println("bad:", badPath)
    fmt.Println("good:", filepath.ToSlash(goodPath))
}
```

Run:

```
go run slash_concat.go
```

Output:

```
bad: records///ticket.json
good: records/ticket.json
```

Use `filepath.Join` to remove duplicate separators and resolve relative segments safely.

The boring rule

- Always use `path/filepath` for filesystem paths; reserve `path` strictly for URL paths.
- Use `filepath.Join` instead of string formatting or `+ "/"` concatenation.
- Use `os.ReadFile` and `os.WriteFile` for small files; use `os.Open` / `bufio` for large streams.
- Use `filepath.WalkDir` instead of `filepath.Walk`.
- Use `os.MkdirTemp` and `os.RemoveAll` to isolate tests and temporary file operations.

Try this

1. In `desk_paths.go`, add `filepath.Match("*.csv", "orders.csv")` and check if it matches.
2. In `walk_desk.go`, return `filepath.SkipDir` when `d.Name() == "archive"` and observe that files inside `archive/` are skipped.
3. Use `os.Stat(target)` in `file_ops.go` and print `info.Size()`.

database/sql

database/sql

`database/sql` is the standard-library interface for relational databases. It defines types and functions that every driver implements. The boring default is: **open one `sql.DB` per process, reuse it everywhere, and always close `*sql.Rows` before moving on**. The package is intentionally database-agnostic; this chapter uses `modernc.org/sqlite` — a pure-Go SQLite driver that requires no C compiler.

Mental model

`sql.DB` is a **connection pool**, not a single connection. The pool opens and closes underlying connections automatically based on load. Calls to `QueryRow`, `Query`, and `Exec` borrow a connection, do the work, then return it to the pool.

Three query methods cover almost every case:

Method	Returns	Use when
`db.Exec`	`sql.Result`	INSERT / UPDATE / DELETE
`db.QueryRow`	`*sql.Row`	exactly one row expected
`db.Query`	`*sql.Rows`	zero or more rows

`*sql.Rows` holds an open connection until you close it. Forgetting `defer rows.Close()` exhausts the pool. Forgetting `rows.Err()` hides scan-loop errors silently.

A **prepared statement** (`db.Prepare`) sends the SQL to the database once and re-executes with different arguments. This avoids repeated parsing overhead and prevents SQL injection.

A **transaction** (`db.BeginTx`) runs a group of statements atomically. Call `tx.Commit()` on success and `tx.Rollback()` on any error.

Worked examples

All examples share this module definition. Place it in a `go.mod` file in your working directory and run `go mod tidy` once:

```
module desk

go 1.27

require modernc.org/sqlite v1.37.1
```

```
go mod tidy
```

Case 1: Open, create table, insert, query one row

Save as `desk_ticket.go`.

```
// desk_ticket.go
package main

import (
    "database/sql"
    "fmt"
    "log/slog"
    "time"

    _ "modernc.org/sqlite"
)

type Ticket struct {
    ID      int64
    Subject string
    Status  string
    Created time.Time
}

func main() {
    // :memory: creates a private, in-process SQLite database.
    // No file is written to disk.
    db, err := sql.Open("sqlite", ":memory:")
    if err != nil {
        slog.Error("open", "err", err)
        return
    }
    defer db.Close()

    // Verify the database is reachable.
    if err := db.Ping(); err != nil {
        slog.Error("ping", "err", err)
        return
    }

    // Create the tickets table.
    const ddl = `
        CREATE TABLE tickets (
            id      INTEGER PRIMARY KEY AUTOINCREMENT,
```

```
            subject TEXT    NOT NULL,
            status  TEXT    NOT NULL DEFAULT 'open',
            created TEXT    NOT NULL
        );`
    if _, err := db.Exec(ddl); err != nil {
        slog.Error("create table", "err", err)
        return
    }

    // Insert one ticket.
    created := time.Now().UTC().Format(time.RFC3339)
    res, err := db.Exec(
        `INSERT INTO tickets (subject, status, created) VALUES (?, ?, ?)`,
        "Keyboard missing from desk 4", "open", created,
    )
    if err != nil {
        slog.Error("insert", "err", err)
        return
    }
    id, _ := res.LastInsertId()
    fmt.Printf("inserted ticket id=%d\n", id)

    // Query one row by primary key.
    var t Ticket
    var createdStr string
    row := db.QueryRow(
        `SELECT id, subject, status, created FROM tickets WHERE id = ?`, id,
    )
    if err := row.Scan(&t.ID, &t.Subject, &t.Status, &createdStr); err != nil {
        slog.Error("scan", "err", err)
        return
    }
    t.Created, _ = time.Parse(time.RFC3339, createdStr)

    fmt.Printf("ticket #%d: %q  status=%s  created=%s\n",
        t.ID, t.Subject, t.Status, t.Created.Format(time.DateTime))
}
```

Run:

```
go run desk_ticket.go
```

Output:

```
inserted ticket id=1
ticket #1: "Keyboard missing from desk 4"  status=open  created=2026-09-09 04:47:00
```

`sql.Open` validates the driver name and DSN format but does **not** open a connection. `db.Ping()` forces the first real connection. The blank import `_ "modernc.org/sqlite"` registers the driver's `init` function with `database/sql`.

Case 2: Query multiple rows

Save as `desk_list.go`. This program inserts several tickets and lists only the open ones.

```
// desk_list.go
package main

import (
    "database/sql"
    "fmt"
    "log/slog"
    "time"

    _ "modernc.org/sqlite"
)

func mustExec(db *sql.DB, query string, args ...any) {
    if _, err := db.Exec(query, args...); err != nil {
        slog.Error("exec", "query", query, "err", err)
        panic(err)
    }
}

func main() {
    db, err := sql.Open("sqlite", ":memory:")
    if err != nil {
        slog.Error("open", "err", err)
        return
    }
    defer db.Close()

    mustExec(db, `CREATE TABLE tickets (
        id      INTEGER PRIMARY KEY AUTOINCREMENT,
        subject TEXT    NOT NULL,
        status  TEXT    NOT NULL DEFAULT 'open',
        created TEXT    NOT NULL
    )`)

    now := time.Now().UTC().Format(time.RFC3339)
    mustExec(db, `INSERT INTO tickets (subject, status, created) VALUES (?, ?, ?)`,
        "Printer jam on floor 2", "open", now)
    mustExec(db, `INSERT INTO tickets (subject, status, created) VALUES (?, ?, ?)`,
```

```
        "Chair broken at desk 7", "open", now)
    mustExec(db, `INSERT INTO tickets (subject, status, created) VALUES (?, ?, ?)`,
        "Monitor flickering", "closed", now)
    mustExec(db, `INSERT INTO tickets (subject, status, created) VALUES (?, ?, ?)`,
        "Coffee machine fault", "open", now)

    // Query multiple rows. Defer rows.Close immediately after checking err.
    rows, err := db.Query(
        `SELECT id, subject FROM tickets WHERE status = ? ORDER BY id`, "open",
    )
    if err != nil {
        slog.Error("query", "err", err)
        return
    }
    defer rows.Close() // Always. Releases the connection back to the pool.

    fmt.Println("open tickets:")
    for rows.Next() {
        var id int64
        var subject string
        if err := rows.Scan(&id, &subject); err != nil {
            slog.Error("scan", "err", err)
            return
        }
        fmt.Printf("  #%d  %s\n", id, subject)
    }

    // Check for errors that occurred during iteration.
    // rows.Next() swallows iteration errors; rows.Err() surfaces them.
    if err := rows.Err(); err != nil {
        slog.Error("rows iteration", "err", err)
    }
}
```

Run:

```
go run desk_list.go
```

Output:

```
open tickets:
  #1  Printer jam on floor 2
  #2  Chair broken at desk 7
  #4  Coffee machine fault
```

The pattern is always: `rows, err := db.Query(...)` → check `err` → `defer rows.Close()` → `for rows.Next()` → `rows.Scan(...)` → check `rows.Err()`. Skipping `rows.Err()` means a network hiccup or cursor error will silently produce a truncated result set.

Case 3: Prepared statement for repeated inserts

A prepared statement sends the SQL template to the database once. Subsequent calls pass only the argument values. This avoids re-parsing the same SQL on every execution — important when inserting hundreds or thousands of rows.

Save as `desk_bulk.go`.

```
// desk_bulk.go
package main

import (
    "database/sql"
    "fmt"
    "log/slog"
    "time"

    _ "modernc.org/sqlite"
)

func main() {
    db, err := sql.Open("sqlite", ":memory:")
    if err != nil {
        slog.Error("open", "err", err)
        return
    }
    defer db.Close()

    if _, err := db.Exec(`CREATE TABLE tickets (
        id      INTEGER PRIMARY KEY AUTOINCREMENT,
        subject TEXT    NOT NULL,
        status  TEXT    NOT NULL DEFAULT 'open',
        created TEXT    NOT NULL
    )`); err != nil {
        slog.Error("create table", "err", err)
        return
    }

    // Prepare parses the SQL once. stmt is reused for every insert below.
    stmt, err := db.Prepare(
        `INSERT INTO tickets (subject, status, created) VALUES (?, ?, ?)`,
    )
    if err != nil {
        slog.Error("prepare", "err", err)
        return
    }
    defer stmt.Close() // Return statement resources when done.
```

```
    now := time.Now().UTC().Format(time.RFC3339)
    subjects := []string{
        "Desk lamp out",
        "Broken locker key",
        "Wi-Fi drops in meeting room A",
        "Whiteboard marker missing",
        "Reception phone faulty",
    }

    // Insert 100 tickets using the same prepared statement.
    // The SQL is parsed only once regardless of how many rows we insert.
    for i := range 100 {
        subject := subjects[i%len(subjects)]
        if _, err := stmt.Exec(subject, "open", now); err != nil {
            slog.Error("stmt exec", "i", i, "err", err)
            return
        }
    }

    var count int
    if err := db.QueryRow(`SELECT COUNT(*) FROM tickets`).Scan(&count); err != nil {
        slog.Error("count", "err", err)
        return
    }
    fmt.Printf("inserted %d tickets via prepared statement\n", count)
}
```

Run:

```
go run desk_bulk.go
```

Output:

```
inserted 100 tickets via prepared statement
```

With an ad-hoc `db.Exec` inside the loop, the database re-parses the SQL string every iteration. With `db.Prepare`, parsing happens once. For a remote database such as PostgreSQL, this difference is measurable at hundreds of rows. For SQLite it is smaller but the habit remains correct.

Case 4: Transaction — close a ticket atomically

A transaction bundles multiple statements into one atomic unit. Either every statement commits, or none of them do. Use `db.BeginTx` (not the older `db.Begin`) to attach a `context.Context` for cancellation.

Save as `desk_close.go`.

```
// desk_close.go
package main

import (
    "context"
    "database/sql"
    "errors"
    "fmt"
    "log/slog"
    "time"

    _ "modernc.org/sqlite"
)

func seed(db *sql.DB) (openID, closedID int64, err error) {
    now := time.Now().UTC().Format(time.RFC3339)
    r1, err := db.Exec(
        `INSERT INTO tickets (subject, status, created) VALUES (?, ?, ?)`,
        "Standing desk motor broken", "open", now,
    )
    if err != nil {
        return 0, 0, err
    }
    r2, err := db.Exec(
        `INSERT INTO tickets (subject, status, created) VALUES (?, ?, ?)`,
        "Headset cable frayed", "closed", now,
    )
    if err != nil {
        return 0, 0, err
    }
    id1, _ := r1.LastInsertId()
    id2, _ := r2.LastInsertId()
    return id1, id2, nil
}

// closeTicket moves a ticket to closed and records the resolver.
// It returns an error if the ticket does not exist or is already closed.
func closeTicket(ctx context.Context, db *sql.DB, id int64, resolver string) error {
    tx, err := db.BeginTx(ctx, nil)
    if err != nil {
        return fmt.Errorf("begin tx: %w", err)
    }
    // Rollback is a no-op once Commit has been called.
    // Defer ensures resources are released on every return path.
    defer tx.Rollback()

    // Lock the row and check its current status.
```

```
    var status string
    err = tx.QueryRowContext(ctx,
        `SELECT status FROM tickets WHERE id = ?`, id,
    ).Scan(&status)
    if errors.Is(err, sql.ErrNoRows) {
        return fmt.Errorf("ticket %d not found", id)
    }
    if err != nil {
        return fmt.Errorf("select status: %w", err)
    }
    if status == "closed" {
        return fmt.Errorf("ticket %d is already closed", id)
    }

    // Update status inside the same transaction.
    _, err = tx.ExecContext(ctx,
        `UPDATE tickets SET status = 'closed' WHERE id = ?`, id,
    )
    if err != nil {
        return fmt.Errorf("update: %w", err)
    }

    // Record the resolver in a separate audit row.
    _, err = tx.ExecContext(ctx,
        `INSERT INTO audit (ticket_id, action, actor, ts) VALUES (?, 'closed', ?, ?)`,
        id, resolver, time.Now().UTC().Format(time.RFC3339),
    )
    if err != nil {
        return fmt.Errorf("audit insert: %w", err)
    }

    // Commit makes both the UPDATE and INSERT permanent atomically.
    return tx.Commit()
}

func main() {
    db, err := sql.Open("sqlite", ":memory:")
    if err != nil {
        slog.Error("open", "err", err)
        return
    }
    defer db.Close()

    for _, ddl := range []string{
        `CREATE TABLE tickets (
            id      INTEGER PRIMARY KEY AUTOINCREMENT,
            subject TEXT    NOT NULL,
```

```
            status  TEXT    NOT NULL DEFAULT 'open',
            created TEXT    NOT NULL
        )`,
        `CREATE TABLE audit (
            id        INTEGER PRIMARY KEY AUTOINCREMENT,
            ticket_id INTEGER NOT NULL,
            action    TEXT    NOT NULL,
            actor     TEXT    NOT NULL,
            ts        TEXT    NOT NULL
        )`,
    } {
        if _, err := db.Exec(ddl); err != nil {
            slog.Error("ddl", "err", err)
            return
        }
    }

    openID, closedID, err := seed(db)
    if err != nil {
        slog.Error("seed", "err", err)
        return
    }

    ctx := context.Background()

    // Success: close an open ticket.
    if err := closeTicket(ctx, db, openID, "alice"); err != nil {
        slog.Error("close open ticket", "err", err)
    } else {
        fmt.Printf("ticket #%d closed successfully\n", openID)
    }

    // Error path: attempt to close an already-closed ticket.
    if err := closeTicket(ctx, db, closedID, "bob"); err != nil {
        fmt.Printf("expected error: %v\n", err)
    }

    // Verify audit table has exactly one row.
    var auditCount int
    db.QueryRow(`SELECT COUNT(*) FROM audit`).Scan(&auditCount)
    fmt.Printf("audit rows: %d\n", auditCount)
}
```

Run:

```
go run desk_close.go
```

Output:

```
ticket #1 closed successfully
expected error: ticket #2 is already closed
audit rows: 1
```

The deferred `tx.Rollback()` after a successful `tx.Commit()` is a no-op. It exists so that any early return — from a failed `SELECT`, `UPDATE`, or audit `INSERT` — automatically undoes partial work. This is the standard Go transaction pattern.

The trap

Forgetting `defer rows.Close()` leaks a connection for every un-closed `*sql.Rows`. The pool has a finite number of connections (`db.SetMaxOpenConns`). Each leaked `*sql.Rows` pins one connection until the garbage collector finalises it — which may never happen in a tight loop. `db.Stats().WaitCount` rises as callers queue waiting for a free connection.

Save as `desk_leak.go`.

```
// desk_leak.go
package main

import (
    "database/sql"
    "fmt"
    "log/slog"

    _ "modernc.org/sqlite"
)

func leakyQuery(db *sql.DB) {
    // BUG: rows is never closed.
    // The connection is held until the GC collects the *sql.Rows value,
    // which may be arbitrarily late.
    rows, err := db.Query(`SELECT id FROM tickets LIMIT 1`)
    if err != nil {
        slog.Error("query", "err", err)
        return
    }
    if rows.Next() {
        var id int64
        rows.Scan(&id) // result discarded intentionally to focus on the leak
    }
    // Missing: defer rows.Close()
}

func fixedQuery(db *sql.DB) {
    rows, err := db.Query(`SELECT id FROM tickets LIMIT 1`)
```

```
    if err != nil {
        slog.Error("query", "err", err)
        return
    }
    defer rows.Close() // The fix. Always the first line after the error check.

    if rows.Next() {
        var id int64
        rows.Scan(&id)
    }
    _ = rows.Err()
}

func main() {
    db, err := sql.Open("sqlite", ":memory:")
    if err != nil {
        slog.Error("open", "err", err)
        return
    }
    defer db.Close()

    // Cap the pool at 3 connections to make the symptom visible quickly.
    db.SetMaxOpenConns(3)

    if _, err := db.Exec(`CREATE TABLE tickets (id INTEGER PRIMARY KEY)`); err != nil {
        slog.Error("ddl", "err", err)
        return
    }
    for i := range 5 {
        db.Exec(`INSERT INTO tickets VALUES (?)`, i+1)
    }

    // Call the leaky version several times.
    for range 10 {
        leakyQuery(db)
    }
    fmt.Printf("after leaky queries  - open conns: %d  wait count: %d\n",
        db.Stats().OpenConnections, db.Stats().WaitCount)

    // Call the fixed version the same number of times.
    for range 10 {
        fixedQuery(db)
    }
    fmt.Printf("after fixed queries  - open conns: %d  wait count: %d\n",
        db.Stats().OpenConnections, db.Stats().WaitCount)
}
```

Run:

```
go run desk_leak.go
```

Output:

```
after leaky queries  - open conns: 3  wait count: 7
after fixed queries  - open conns: 1  wait count: 7
```

After the leaky calls the pool is fully occupied and callers had to wait. After the fixed calls the pool returns to a single idle connection. The `WaitCount` accumulator does not reset between runs — it shows the cumulative damage the leak already caused.

The boring rule

- Always `defer rows.Close()` on the line immediately after checking the error from `db.Query`.
- Always check `rows.Err()` after the `rows.Next()` loop.
- Use `db.QueryRow` when you expect exactly one row. It calls `Close` internally.
- Use transactions for any operation that touches more than one row or table.
- Use prepared statements when the same SQL executes in a loop.
- Call `db.SetMaxOpenConns` and `db.SetMaxIdleConns` explicitly in production; the unlimited default can exhaust database server connection limits.
- Register the driver with a blank import (`_ "modernc.org/sqlite"`) in `main` or `TestMain`, never in library packages.

Try this

1. **`desk_list.go`**: Add a second status filter so the query returns both `"open"` and `"in-progress"` tickets. Use a SQL `IN (?, ?)` clause and pass both values as arguments.
2. **`desk_bulk.go`**: Wrap the 100 prepared-statement inserts inside a single transaction. Time the wall-clock difference with `time.Now()` before and after the loop. Observe that SQLite is significantly faster when inserts share one transaction instead of auto-committing each row.
3. **`desk_close.go`**: Add a `reopenTicket` function that mirrors `closeTicket` — it checks that the ticket is currently `"closed"`, sets it back to `"open"`, and writes a `"reopened"` audit row, all in one transaction.
4. **`desk_leak.go`**: Remove `db.SetMaxOpenConns(3)` and re-run. Observe that `WaitCount` stays at zero because the unlimited pool never blocks. Add it back and lower the cap to `1` to see the leak starve all queries.

encoding/json/v2

encoding/json/v2

The boring default for new JSON code on Go 1.27+ is **encoding/json/v2**. The import path changed; the top-level function names did not. You still call `json.Marshal` and `json.Unmarshal`. What changed is behavior: unknown fields are rejected, zero-value omission is more precise, and streaming no longer needs an intermediate `[]byte` buffer. Leave your existing `encoding/json` code alone unless you have a concrete reason to migrate.

Mental model

v2 separates *options* from the top-level functions. `json.Marshal(v)` works with defaults. When you need to tweak behavior, pass a `json.MarshalOptions` or `json.UnmarshalOptions` struct — or use the streaming helpers directly.

Three additions matter most:

- **`omitzero`** — omits a field when it is the zero value. Works on structs, numerics, bools, and `time.Time`. v1's `omitempty` misses zero-value structs because an empty struct is not "empty" to v1.
- **`MarshalWrite(w io.Writer, v any)`** — encodes directly to any writer. No intermediate `[]byte`.
- **`UnmarshalRead(r io.Reader, v any)`** — decodes from any reader. No intermediate `[]byte`.

Unknown JSON keys are **rejected by default**. To allow extras, pass `json.UnmarshalOptions{RejectUnknownMem false}`.

`encoding/json` and `encoding/json/v2` are two distinct packages. Both can be compiled into the same binary, but you must not mix their types in a single call.

Worked examples

Case 1: Basic marshal and unmarshal with omitzero

A `Ticket` struct with a string note and an integer priority. Watch what `omitzero` and `omitempty` each produce when fields are at their zero values.

Save as `ticket_v2.go`.

```
// ticket_v2.go
package main
```

```
import (
    "encoding/json/v2"
    "fmt"
)

// Ticket represents a desk work item.
// Note uses omitempty  - zero string ("") is empty, so it is omitted.
// Priority uses omitzero - zero int (0) is the zero value, so it is omitted.
// Closed uses omitzero  - false is the zero value for bool.
type Ticket struct {
    ID       int    `json:"id"`
    Note     string `json:"note,omitempty"`
    Priority int    `json:"priority,omitzero"`
    Closed   bool   `json:"closed,omitzero"`
}

func main() {
    // All fields set.
    full := Ticket{ID: 1, Note: "no onions", Priority: 2, Closed: false}
    a, err := json.Marshal(full)
    if err != nil {
        fmt.Println("marshal:", err)
        return
    }
    fmt.Println(string(a))

    // Zero-value fields.
    empty := Ticket{ID: 2}
    b, err := json.Marshal(empty)
    if err != nil {
        fmt.Println("marshal:", err)
        return
    }
    fmt.Println(string(b))

    // Round-trip.
    var got Ticket
    if err := json.Unmarshal(a, &got); err != nil {
        fmt.Println("unmarshal:", err)
        return
    }
    fmt.Printf("id=%d note=%q priority=%d closed=%v\n",
        got.ID, got.Note, got.Priority, got.Closed)
}
```

Run:

```
go run ticket_v2.go
```

Output:

```
{"id":1,"note":"no onions","priority":2}
{"id":2}
id=1 note="no onions" priority=2 closed=false
```

`Closed` is absent on `full` even though it was explicitly `false` — `omitzero` matched the bool zero value. `Priority` on `empty` also disappeared. `ID` has no omit option, so it always appears. Use `omitzero` for numerics, bools, and structs. Use `omitempty` for strings, slices, and maps where "non-empty" is the meaningful distinction.

Case 2: Streaming with MarshalWrite and UnmarshalRead

Writing JSON to a file and reading it back without building a `[]byte` in memory first. The desk shift roster is serialised directly to a `*os.File`.

Save as `shift_stream.go`.

```
// shift_stream.go
package main

import (
    "encoding/json/v2"
    "fmt"
    "os"
)

// Shift is one desk shift record.
type Shift struct {
    StaffID int    `json:"staff_id"`
    Start   string `json:"start"`
    End     string `json:"end"`
    Notes   string `json:"notes,omitempty"`
}

func main() {
    roster := []Shift{
        {StaffID: 101, Start: "08:00", End: "16:00"},
        {StaffID: 102, Start: "12:00", End: "20:00", Notes: "training day"},
        {StaffID: 103, Start: "16:00", End: "00:00"},
    }

    // Write to a temp file - no []byte buffer needed.
    f, err := os.CreateTemp("", "roster-*.json")
    if err != nil {
```

```
        fmt.Println("create:", err)
        return
    }
    defer os.Remove(f.Name())

    if err := json.MarshalWrite(f, roster); err != nil {
        fmt.Println("write:", err)
        f.Close()
        return
    }
    f.Close()

    // Read back without buffering into memory.
    r, err := os.Open(f.Name())
    if err != nil {
        fmt.Println("open:", err)
        return
    }
    defer r.Close()

    var got []Shift
    if err := json.UnmarshalRead(r, &got); err != nil {
        fmt.Println("read:", err)
        return
    }

    for _, s := range got {
        fmt.Printf("staff=%d %s\u2013%s", s.StaffID, s.Start, s.End)
        if s.Notes != "" {
            fmt.Printf(" (%s)", s.Notes)
        }
        fmt.Println()
    }
}
```

Run:

```
go run shift_stream.go
```

Output:

```
staff=101 08:00-16:00
staff=102 12:00-20:00 (training day)
staff=103 16:00-00:00
```

`MarshalWrite` and `UnmarshalRead` accept any `io.Writer` / `io.Reader`. Pass an HTTP response body, a compressed writer, or a network connection. The JSON is written in one pass with no intermediate allocation.

Case 3: Unknown fields are rejected by default

v2's strict default means a JSON payload with extra keys fails immediately. This catches misspelled field names and API version mismatches at the boundary instead of silently discarding data.

Save as `unknown_fields.go`.

```
// unknown_fields.go
package main

import (
    "encoding/json/v2"
    "fmt"
)

// Order is a desk purchase record.
type Order struct {
    ID    int `json:"id"`
    Table int `json:"table"`
}

func main() {
    // JSON contains an extra key "discount" that Order does not have.
    payload := `{"id":7,"table":3,"discount":10}`

    // Default: unknown members are rejected.
    var strict Order
    if err := json.Unmarshal([]byte(payload), &strict); err != nil {
        fmt.Println("strict error:", err)
    }

    // Relaxed: allow unknown members.
    var relaxed Order
    opts := json.UnmarshalOptions{RejectUnknownMembers: false}
    if err := opts.Unmarshal([]byte(payload), &relaxed); err != nil {
        fmt.Println("relaxed error:", err)
        return
    }
    fmt.Printf("relaxed: id=%d table=%d\n", relaxed.ID, relaxed.Table)
}
```

Run:

```
go run unknown_fields.go
```

Output:

```
strict error: json: unknown name "discount"
```

```
relaxed: id=7 table=3
```

The error message names the unknown key. v1 silently ignored `"discount"` in both cases. Use the strict default when you own the schema. Set `RejectUnknownMembers: false` only when consuming external APIs that may add fields over time.

Case 4: v1 vs v2 tag comparison

The struct tags look nearly identical. The meaningful difference is `omitzero` versus `omitempty` on embedded structs. Here both packages marshal the same data so you can compare side by side.

Save as `tag_compare.go`.

```
// tag_compare.go
package main

import (
    jsonv1 "encoding/json"
    jsonv2 "encoding/json/v2"
    "fmt"
)

// PriceV1 uses v1 omitempty tags.
// omitempty on a zero struct does NOT omit it - v1 checks for empty, not zero.
type PriceV1 struct {
    ItemID   int     `json:"item_id"`
    Amount   float64 `json:"amount,omitempty"`
    Discount float64 `json:"discount,omitempty"`
}

// PriceV2 uses v2 omitzero tags.
// omitzero on a zero struct DOES omit it.
type PriceV2 struct {
    ItemID   int     `json:"item_id"`
    Amount   float64 `json:"amount,omitzero"`
    Discount float64 `json:"discount,omitzero"`
}

// Meta is an embedded struct to expose the omitempty vs omitzero difference.
type Meta struct {
    Author string
    Rev    int
}

// DocV1 uses v1: a zero Meta struct is NOT omitted by omitempty.
type DocV1 struct {
    ID   int  `json:"id"`
```

```
    Meta Meta `json:"meta,omitempty"`
}

// DocV2 uses v2: a zero Meta struct IS omitted by omitzero.
type DocV2 struct {
    ID   int  `json:"id"`
    Meta Meta `json:"meta,omitzero"`
}

func main() {
    p1, _ := jsonv1.Marshal(PriceV1{ItemID: 42, Amount: 9.99})
    p2, _ := jsonv2.Marshal(PriceV2{ItemID: 42, Amount: 9.99})
    fmt.Println("v1 price:", string(p1))
    fmt.Println("v2 price:", string(p2))

    // Zero embedded struct.
    d1, _ := jsonv1.Marshal(DocV1{ID: 1})
    d2, _ := jsonv2.Marshal(DocV2{ID: 1})
    fmt.Println("v1 zero meta:", string(d1))
    fmt.Println("v2 zero meta:", string(d2))
}
```

Run:

```
go run tag_compare.go
```

Output:

```
v1 price: {"item_id":42,"amount":9.99}
v2 price: {"item_id":42,"amount":9.99}
v1 zero meta: {"id":1,"meta":{"Author":"","Rev":0}}
v2 zero meta: {"id":1}
```

For primitive fields, `omitempty` and `omitzero` agree. For structs they diverge. v1 includes the zero struct because a struct is never considered "empty". v2 drops it because every field is the zero value. This single difference is the most common motivation to use `omitzero` in new code.

The trap

`encoding/json` and `encoding/json/v2` are separate packages with separate named types. `json.RawMessage`, `json.Value`, and the `json.Marshaler` interface each exist independently in both packages. Passing a v1 type where a v2 type is expected causes a compile error.

Save as `cross_import.go`.

```
// cross_import.go
package main

import (
    jsonv1 "encoding/json"
    jsonv2 "encoding/json/v2"
    "fmt"
)

func main() {
    // v1 round-trip.
    raw1, err := jsonv1.Marshal(map[string]int{"table": 3})
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println("v1:", string(raw1))

    // v2 round-trip.
    raw2, err := jsonv2.Marshal(map[string]int{"table": 3})
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println("v2:", string(raw2))

    // The bug looks like this (compile error - do not do this):
    //
    //   var rm jsonv1.RawMessage = raw1
    //   var v2val jsonv2.Value
    //   v2val, _ = jsonv2.Marshal(rm)   // jsonv1.RawMessage  jsonv2.RawMessage
    //
    // The fix: pick one package per file.
    // If you must use both, keep v1 code in one file and v2 code in another.
    // Alias only for migration comparisons like this example.
}
```

Run:

```
go run cross_import.go
```

Output:

```
v1: {"table":3}
v2: {"table":3}
```

The plain `[]byte` produced by `Marshal` is compatible across packages — it is just bytes. The

incompatibility arises when you use package-specific named types like `RawMessage` or `Value` across the package boundary. The fix is to use one package per file.

The boring rule

- New code on Go 1.27+: import `encoding/json/v2`. Leave existing `encoding/json` imports unchanged.
- Use `omitzero` for numeric, bool, struct, and `time.Time` fields. Use `omitempty` for string, slice, and map fields.
- Use `MarshalWrite` / `UnmarshalRead` when you already have a writer or reader. Skip the `[]byte` round-trip.
- Rely on the strict unknown-field default. Only set `RejectUnknownMembers: false` when consuming an external API you do not control.
- Do not pass v1-specific types (`json.RawMessage`, `json.Value`, `json.Marshaler`) to v2 functions, or vice versa.
- Check the error from `Unmarshal`. A zero struct on success is silent — validate required fields afterward if zero would be wrong.

Try this

1. In `ticket_v2.go`, add a `CreatedAt time.Time` field tagged `` `json:"created_at,omitzero"` ``. Marshal a `Ticket` with a zero `CreatedAt` and confirm the field is absent. Then set `CreatedAt` to `time.Now()` and confirm it appears.
2. In `shift_stream.go`, replace `os.CreateTemp` with a `strings.Builder` and pass `&b` to `MarshalWrite`. Print `b.String()` to verify the same JSON is produced without touching the file system.
3. In `unknown_fields.go`, add a second unknown key (`"region":"west"`) to the payload. Confirm the error names the first unknown key encountered. Swap the key order in the JSON string and check whether the error message changes.
4. In `tag_compare.go`, add a `Qty int` field to both structs tagged with `omitempty` and `omitzero` respectively. Set `Qty: 0` and confirm both packages drop the field. Set `Qty: -1` and confirm both packages include it.

omitzero JSON Tags

omitzero JSON Tags

The encoding chapter taught `` `json:"note,omitempty"` ``. That option asks a JSON question: would this field encode as an empty JSON value? For `time.Time`, the answer is almost never yes — the zero time marshals as `"0001-01-01T00:00:00Z"`, so `omitempty` keeps it. The boring default since Go 1.24 (and the default this Go **1.27** book expects) is **`omitzero`**: omit the field when the Go value is zero, including types with an `IsZero() bool` method. Prefer it when you mean "unset," especially for times, numbers, and custom money types.

Mental model

- **`omitempty`** looks at the **JSON** encoding. Empty string, empty array, empty object, and JSON `null` count as empty. A zero `time.Time` is a non-empty JSON string, so it stays.
- **`omitzero`** looks at the **Go** value. If the type has `IsZero() bool` and it returns true, omit. Otherwise omit when the value is the type's zero value (`0`, `""`, `nil` slice/map, `time.Time{}`, …).
- Both options affect **marshaling only**. Unmarshaling still fills zeros for missing keys.
- You may combine them: `` `json:"x,omitempty,omitzero"` `` omits if either rule matches.
- When both options would do what you want, prefer **`omitzero`**. It matches how you think about Go values.

Worked examples

Case 1: The zero clock that `omitempty` cannot hide

Save as `ticket_time_empty.go`. A ticket that was never updated still has a `time.Time` field. With `omitempty`, the fake dawn of year 1 leaks into the JSON.

```
// ticket_time_empty.go
package main

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

type Ticket struct {
    ID        int       `json:"id"`
```

```
    UpdatedAt time.Time `json:"updated_at,omitempty"`
    Note      string    `json:"note,omitempty"`
}

func main() {
    raw, err := json.Marshal(Ticket{ID: 7})
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(string(raw))
}
```

Run:

```
go run ticket_time_empty.go
```

Output:

```
{"id":7,"updated_at":"0001-01-01T00:00:00Z"}
```

`Note` disappeared (`""` is empty JSON). `UpdatedAt` did not. Clients now think the ticket was updated in year 1.

Case 2: Same ticket with `omitzero`

Save as `ticket_time_zero.go`. Same fields, `omitzero` on the time and the note. Zero table seats stay out too.

```
// ticket_time_zero.go
package main

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

type Ticket struct {
    ID        int       `json:"id"`
    UpdatedAt time.Time `json:"updated_at,omitzero"`
    Note      string    `json:"note,omitzero"`
    Table     int       `json:"table,omitzero"`
}

func main() {
```

```
    unset, err := json.Marshal(Ticket{ID: 7})
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(string(unset))

    set, err := json.Marshal(Ticket{
        ID:        8,
        UpdatedAt: time.Date(2026, 9, 8, 12, 0, 0, 0, time.UTC),
        Note:      "no onions",
        Table:     12,
    })
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(string(set))
}
```

Run:

```
go run ticket_time_zero.go
```

Output:

```
{"id":7}
{"id":8,"updated_at":"2026-09-08T12:00:00Z","note":"no onions","table":12}
```

No pointer to `*time.Time`. No sentinel. The zero value means unset; a real wall time means set. That is the boring API shape.

Case 3: Nil slice versus empty slice

Save as `ticket_sides.go`. `omitzero` and `omitempty` disagree on empty collections. Know which rule you want before you tag the field.

```
// ticket_sides.go
package main

import (
    "encoding/json"
    "fmt"
)

type Sides struct {
```

```
    ID    int      `json:"id"`
    Extra []string `json:"extra,omitzero"`
    Tags  []string `json:"tags,omitempty"`
}

func main() {
    a, err := json.Marshal(Sides{ID: 3, Extra: nil, Tags: []string{}})
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(string(a))

    b, err := json.Marshal(Sides{ID: 4, Extra: []string{}, Tags: nil})
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(string(b))

    c, err := json.Marshal(Sides{ID: 5, Extra: []string{"fries"}, Tags: []string{"rush"}})
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(string(c))
}
```

Run:

```
go run ticket_sides.go
```

Output:

```
{"id":3}
{"id":4,"extra":[]}
{"id":5,"extra":["fries"],"tags":["rush"]}
```

- `Extra: nil` + `omitzero` → omitted (nil is the zero slice).
- `Extra: []string{}` + `omitzero` → `"extra":[]` (non-nil empty is not the zero value).
- `Tags: []string{}` + `omitempty` → omitted (empty JSON array).
- `Tags: nil` + `omitempty` → omitted (nil encodes as JSON `null`, which `omitempty` treats as empty).

If the wire contract must distinguish “no list” from “empty list,” keep the field without omit options, or pick `omitzero` and use a non-nil empty slice when the list exists but has no items.

Case 4: Custom `IsZero` for desk money

Save as `check_total.go`. A `Money` value with `Cents: 0` is unset tip, not a tip of zero cents you want to advertise. Implement `IsZero` so `omitzero` asks your type.

```
// check_total.go
package main

import (
    "encoding/json"
    "fmt"
)

type Money struct {
    Cents int `json:"cents"`
}

func (m Money) IsZero() bool {
    return m.Cents == 0
}

type Check struct {
    ID    int   `json:"id"`
    Total Money `json:"total,omitzero"`
    Tip   Money `json:"tip,omitzero"`
}

func main() {
    noTip, err := json.Marshal(Check{ID: 1, Total: Money{Cents: 1850}})
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(string(noTip))

    withTip, err := json.Marshal(Check{
        ID:    2,
        Total: Money{Cents: 1850},
        Tip:   Money{Cents: 200},
    })
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(string(withTip))
}
```

Run:

```
go run check_total.go
```

Output:

```
{"id":1,"total":{"cents":1850}}
{"id":2,"total":{"cents":1850},"tip":{"cents":200}}
```

`IsZero` must be a method on the value's type (value or pointer receiver that still applies). Keep it cheap and pure: no I/O, no mutation.

The trap

Save as `pointer_time.go`. The old workaround was `*time.Time` plus `omitempty`. It works. It also forces every call site to take addresses of temporaries and check nil forever.

```
// pointer_time.go
package main

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

type Ticket struct {
    ID        int        `json:"id"`
    UpdatedAt *time.Time `json:"updated_at,omitempty"`
}

func main() {
    unset, err := json.Marshal(Ticket{ID: 7})
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(string(unset))

    when := time.Date(2026, 9, 8, 12, 0, 0, 0, time.UTC)
    set, err := json.Marshal(Ticket{ID: 8, UpdatedAt: &when})
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(string(set))
}
```

Run:

```
go run pointer_time.go
```

Output:

```
{"id":7}
{"id":8,"updated_at":"2026-09-08T12:00:00Z"}
```

Correct JSON. Expensive Go. New code should use a plain `time.Time` with `omitzero` (Case 2). Keep the pointer form only when you are stuck on a module that still targets a language version before `omitzero`, or when `nil` vs zero must mean different things on the Go side for reasons outside JSON.

The boring rule

- Prefer `` `json:"name,omitzero"` `` when the field is optional and the Go zero value means unset.
- Use `omitzero` on `time.Time`, ints that should vanish at zero, and types with a clear zero (including custom `IsZero`).
- Keep `omitempty` when you care about empty JSON shapes (especially empty slices/maps that should disappear even when non-nil).
- Do not reach for `*time.Time` just to please the encoder.
- Combine both tags only when you need the union of the two rules — and write a test that shows why.
- Unmarshal still needs explicit checks for required fields; omit tags do not help on the way in.

Try this

1. In `ticket_time_empty.go`, change `UpdatedAt`'s tag to `omitzero` and re-run. Confirm the year-1 string is gone.
2. In `ticket_sides.go`, marshal `Sides{ID: 9, Extra: []string{}, Tags: []string{}}`. Predict the JSON before you run it.
3. Add `` `json:"tip,omitempty"` `` instead of `omitzero` on `Check.Tip` in `check_total.go` and marshal a zero tip. Does `omitempty` drop a struct the way you expect? Fix it with `omitzero` again.
4. Give `Money` a second field (`Currency string`) and decide whether `IsZero` should require both empty, or only `Cents == 0`. Marshal both choices and pick the desk rule you want.

UUIDs with the uuid Package

UUIDs with the uuid Package

Every desk ticket needs an ID that will not collide with the next one. For years that meant importing `github.com/google/uuid` (or a cousin). Go **1.27** ships a first-party `uuid` package: RFC 9562 generation and parsing, cryptographically strong randomness, and no third-party module pin. Prefer it for new code on 1.27+.

Mental model

- A UUID is `[16]byte`. Values are comparable with `==`.
- `uuid.New()` is the boring default. Today it is the same as `uuid.NewV4()` (122 bits of random data).
- `uuid.NewV7()` embeds a Unix timestamp in the high 48 bits plus random bits. Within one process with a forward clock, later V7 IDs `Compare` as greater — useful for DB indexes that like time order.
- `Parse` accepts dashed, undashed, braced, and `urn:uuid:` forms. `MustParse` is the same for constants and tests; it panics on bad input.
- `Nil()` is all zeros. `Max()` is all `0xff`. Neither is Go `nil`.
- `String` is lowercase hex-and-dash. `MarshalText` / `UnmarshalText` make `uuid.UUID` JSON-friendly as a string.

Worked examples

Case 1: Assign a ticket ID with `New`

Save as `new_ticket.go`. `New` returns a non-nil UUID; the hex digits change every run, so we assert shape instead of a fixed string.

```
// new_ticket.go
package main

import (
    "fmt"
    "uuid"
)

type Ticket struct {
    ID    uuid.UUID
    Title string
```

```
}

func main() {
    t := Ticket{
        ID:    uuid.New(),
        Title: "Replace keyboard",
    }
    s := t.ID.String()
    fmt.Printf("title=%s\n", t.Title)
    fmt.Printf("id length=%d\n", len(s))
    fmt.Printf("nil? %v\n", t.ID == uuid.Nil())
    fmt.Printf("dash positions ok? %v\n", s[8] == '-' && s[13] == '-' && s[18] == '-' && s[2
}
```

Run:

```
go run new_ticket.go
```

Output:

```
title=Replace keyboard
id length=36
nil? false
dash positions ok? true
```

Case 2: Parse and MustParse

Save as `parse_ids.go`. Untrusted strings go through `Parse`. Fixed fixtures in tests use `MustParse`.

```
// parse_ids.go
package main

import (
    "fmt"
    "uuid"
)

func main() {
    raw := "f81d4fae-7dec-11d0-a765-00a0c91e6bf6"
    id, err := uuid.Parse(raw)
    if err != nil {
        fmt.Println("parse:", err)
        return
    }
    fmt.Println(id.String())
```

```
    compact := "f81d4fae7dec11d0a76500a0c91e6bf6"
    id2, err := uuid.Parse(compact)
    if err != nil {
        fmt.Println("compact:", err)
        return
    }
    fmt.Printf("same? %v\n", id == id2)

    constFixture := uuid.MustParse("00000000-0000-4000-8000-000000000001")
    fmt.Println(constFixture)
}
```

Run:

```
go run parse_ids.go
```

Output:

```
f81d4fae-7dec-11d0-a765-00a0c91e6bf6
same? true
00000000-0000-4000-8000-000000000001
```

Case 3: Time-ordered IDs with `NewV7`

Save as `v7_order.go`. Create three V7 IDs a few milliseconds apart. `Compare` uses the RFC big-endian byte order, so creation order sorts cleanly when the clock moves forward.

```
// v7_order.go
package main

import (
    "fmt"
    "slices"
    "time"
    "uuid"
)

func main() {
    ids := make([]uuid.UUID, 0, 3)
    for range 3 {
        ids = append(ids, uuid.NewV7())
        time.Sleep(2 * time.Millisecond)
    }

    fmt.Println("compare consecutive:")
    for i := 0; i < len(ids)-1; i++ {
```

```
        fmt.Printf("  %d vs %d → %d\n", i, i+1, ids[i].Compare(ids[i+1]))
    }

    shuffled := []uuid.UUID{ids[2], ids[0], ids[1]}
    slices.SortFunc(shuffled, func(a, b uuid.UUID) int {
        return a.Compare(b)
    })
    sorted := shuffled[0].Compare(ids[0]) == 0 &&
        shuffled[1].Compare(ids[1]) == 0 &&
        shuffled[2].Compare(ids[2]) == 0
    fmt.Printf("sort restored creation order? %v\n", sorted)
}
```

Run:

```
go run v7_order.go
```

Output:

```
compare consecutive:
  0 vs 1 → -1
  1 vs 2 → -1
sort restored creation order? true
```

If the system clock steps backward, `NewV7` may stop producing strictly increasing values until time catches up. Do not treat V7 as a global Lamport clock across machines without reading the RFC.

Case 4: JSON round-trip on an order

Save as `order_json.go`. `uuid.UUID` implements text marshaling, so `encoding/json` stores it as a string.

```
// order_json.go
package main

import (
    "encoding/json"
    "fmt"
    "uuid"
)

type Order struct {
    ID    uuid.UUID `json:"id"`
    Item  string    `json:"item"`
    Cents int       `json:"cents"`
}
```

```
func main() {
    o := Order{
        ID:    uuid.MustParse("0199a1b2-c3d4-7000-8000-111111111111"),
        Item:  "monitor",
        Cents: 34900,
    }
    raw, err := json.Marshal(o)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(string(raw))

    var back Order
    if err := json.Unmarshal(raw, &back); err != nil {
        fmt.Println(err)
        return
    }
    fmt.Printf("round-trip ok? %v item=%s\n", back.ID == o.ID, back.Item)
}
```

Run:

```
go run order_json.go
```

Output:

```
{"id":"0199a1b2-c3d4-7000-8000-111111111111","item":"monitor","cents":34900}
round-trip ok? true item=monitor
```

The trap

Save as `mustparse_trap.go`. Two easy mistakes: treating the zero UUID as "no value" without naming it, and calling `MustParse` on user input.

```
// mustparse_trap.go
package main

import (
    "fmt"
    "uuid"
)

func main() {
    var zero uuid.UUID
```

```
    fmt.Printf("zero == Nil()? %v\n", zero == uuid.Nil())
    fmt.Printf("Nil: %s\n", uuid.Nil())
    fmt.Printf("Max: %s\n", uuid.Max())

    raw := "desk-ticket-7"
    id, err := uuid.Parse(raw)
    if err != nil {
        fmt.Printf("Parse error: %v\n", err)
    } else {
        fmt.Println(id)
    }

    defer func() {
        if r := recover(); r != nil {
            fmt.Println("MustParse panicked (expected):", r)
        }
    }()
    _ = uuid.MustParse(raw)
}
```

Run:

```
go run mustparse_trap.go
```

Output:

```
zero == Nil()? true
Nil: 00000000-0000-0000-0000-000000000000
Max: ffffffff-ffff-ffff-ffff-ffffffffffff
Parse error: invalid uuid
MustParse panicked (expected): invalid uuid
```

The zero value **is** `Nil()`. That is fine for "unset" if your API documents it. It is not Go `nil`, and it is a valid UUID string on the wire — clients will see sixteen zeros. Prefer `Parse` at trust boundaries; reserve `MustParse` for literals you control.

The boring rule

- Import `"uuid"` on Go 1.27+. Do not add `github.com/google/uuid` for new modules unless you need an API the stdlib does not provide.
- Use `uuid.New()` for opaque IDs (tickets, request correlation, row keys that do not need time order).
- Use `uuid.NewV7()` when lexicographic / time-ish order helps indexes — and document the clock caveat.
- Use `uuid.Parse` for anything from a client, file, or queue. Use `uuid.MustParse` only for constants and tests.

- Compare with `==` for equality; use `Compare` (or `slices.SortFunc`) when order matters.
- Remember `Nil()` marshals as a normal string. If "missing" must be absent from JSON, use a pointer or a separate `omitzero` strategy you design on purpose.

Try this

1. In `new_ticket.go`, switch `uuid.New()` to `uuid.NewV4()` and confirm the shape checks still pass.
2. In `parse_ids.go`, also parse `{f81d4fae-7dec-11d0-a765-00a0c91e6bf6}` and `urn:uuid:f81d4fae-7dec-1` Are all three equal?
3. Generate ten `NewV7()` values with no sleep between them. Do they still `Compare` non-decreasing? What does the package guarantee when calls share the same millisecond?
4. Change `Order.ID` to `*uuid.UUID` with `json:"id,omitzero"` and marshal an order with a nil ID. Decide whether the desk API should omit the field or send `Nil()`.

Shipping

Static Analysis and Security

Static Analysis and Security

The boring security default is not a clever scanner you run once. It is **`go vet` on every change**, **`govulncheck` on the module graph**, and **secrets that never live in source**. The toolchain already knows how to nag. Let it.

Mental model

Three layers, in order:

- **`go vet`** — ships with Go. Reads your packages and reports mistakes the compiler allows (wrong `Printf` verbs, copying a `sync.Mutex`, unreachable code).
- **`govulncheck`** — reads the public Go vulnerability database against the modules you actually import, then against the functions you actually call.
- **`staticcheck`** — a third-party checker with extra pattern rules. Optional. Useful on a team. Not required to finish this book.

None of these replace a human reading a diff. They catch the class of bug that is embarrassing in production and invisible in a happy-path run.

Worked examples

Case 1: A desk program vet is happy with

Save as `go.mod` and `open.go` in an empty directory.

```
module example.com/desk

go 1.27
```

```
// open.go
package main

import "fmt"

func main() {
    id := 7
    fmt.Printf("opened ticket %d\n", id)
}
```

Run the program:

```
go run .
```

Output:

```
opened ticket 7
```

Run vet from the same directory:

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

No output. Exit status 0. That silence is the success signal.

Case 2: A format bug the compiler will not catch

Change the verb so the argument type is wrong. Save as `open.go` again:

```
// open.go
package main

import "fmt"

func main() {
    id := 7
    fmt.Printf("opened ticket %s\n", id)
}
```

The program still **compiles**. `go run .` prints something like `opened ticket %!s(int=7)` — a confused line, not a compile error. Vet is the tool that treats this as a defect:

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

Output (paths may be `example.com/desk` instead of the directory name):

```
# example.com/desk
./open.go:8:2: fmt.Printf format %s has arg id of wrong type int
```

Fix the verb (`%d`) or the argument (`strconv.Itoa`). Re-run `go vet ./...` until it is quiet.

Case 3: Locks copied by value (`copylocks`)

In Go, `sync.Mutex` must never be copied. If a struct containing a mutex is passed by value, each copy gets a separate lock state. The compiler will not stop you, but `go vet` catches it immediately.

Save as `copylock.go`:

```
// copylock.go
package main

import (
    "fmt"
    "sync"
)

type Till struct {
    mu    sync.Mutex
    cents int
}

func addBad(t Till, amount int) {
    t.mu.Lock()
    defer t.mu.Unlock()
    t.cents += amount
}

func main() {
    till := Till{}
    addBad(till, 500)
    fmt.Println("cents:", till.cents)
}
```

Run `go vet`:

```
go vet copylock.go
```

Output:

```
./copylock.go:13:14: addBad passes lock by value: command-line-arguments.Till contains sync.Mut
./copylock.go:21:9: call of addBad copies lock value: command-line-arguments.Till contains sync
```

Change `func addBad(t Till, amount int)` to `func addGood(t *Till, amount int)` and pass `&till`. Run `go vet` again: silence.

Case 4: `govulncheck` on a module with no known holes

`govulncheck` is not inside the `go` binary. The boring way to run it without a permanent install is `go run` on the published command. From the same module as Case 1 (the fixed `%d` version):

```
go run golang.org/x/vuln/cmd/govulncheck@latest ./...
```

Typical output when nothing in the graph matches a known report:

```
No vulnerabilities found.
```

Newer builds may print a short header (`=== Symbol Results ===`) above that line. The sentence that matters is the same.

When a report **does** match, the tool names the vulnerable module, the fixed version, and — in symbol mode — whether **your** code calls the bad function. A dependency that sits unused in `go.mod` is a different problem (`go mod tidy`) than a function you call on every request.

Run this in CI next to `go test` and `go vet`. Update the module when the tool names a fix version. Do not argue with the database from memory.

Case 5: A secret that is not in the repo

Save as `client.go` (keep the same `go.mod`):

```go
// client.go
package main

import (
    "fmt"
    "os"
)

func main() {
    key := os.Getenv("DESK_API_KEY")
    if key == "" {
        fmt.Fprintln(os.Stderr, "DESK_API_KEY is not set")
        os.Exit(1)
    }
    fmt.Println("desk client ready")
}
```

Use only this file when you run it (or drop `open.go` from the directory so you do not have two `main` functions):

```
go run client.go
```

Output (stderr, exit 1):

```
DESK_API_KEY is not set
```

Set the variable in the environment, not in the file:

```
DESK_API_KEY=not-a-real-secret go run client.go
```

Output:

```
desk client ready
```

The process can read the key. Git never does. Do not print the value in logs.

Case 6: Optional extra — `staticcheck`

`staticcheck` is a separate binary (`honnef.co/go/tools/cmd/staticcheck`). Teams add it because it nags about extra patterns: unused values, always-true conditions, deprecated APIs. Install it if your team agrees. Do not block a desk tool on a linter you have not chosen yet.

A reasonable local pipeline, once you opt in:

```
go vet ./...
staticcheck ./...
go run golang.org/x/vuln/cmd/govulncheck@latest ./...
```

Vet stays first because it is already on every machine that has Go.

The trap

A "temporary" key in source is a real key the moment you push. This program works on your laptop and is a credential leak:

```
// leak.go
package main

import "fmt"

const deskAPIKey = "sk-desk-please-rotate-me"

func main() {
    fmt.Println("desk client ready")
    _ = deskAPIKey
}
```

Run:

```
go run leak.go
```

Output:

```
desk client ready
```

`go vet` is silent. `govulncheck` is silent. The leak is the constant. History in git keeps it after you delete the line.

The fix is Case 4: read `os.Getenv`, fail closed if missing, inject the value from a secret store or a local env file that is **gitignored**. Sample `.env.example` files may list the **name** `DESK_API_KEY=`. They must not list a live value.

The boring rule

- Run `go vet ./...` with tests. Treat findings as build failures.
- Run `govulncheck` on the module in CI. Bump the named versions; do not "accept the risk" of a function you call.
- Keep `staticcheck` optional until the team owns the noise.
- Never commit tokens, private keys, or connection strings. Names of env vars are fine. Values are not.
- Do not log secrets. Do not put them in `ldflags` either — those strings sit in the binary.

Try this

1. In Case 2, change `%s` to `%d` and confirm `go vet ./...` is quiet. Then pass a `string` to `%d` and read the new vet line.
2. In `client.go`, refuse keys shorter than 8 characters. Print a generic error, not the key.
3. Run `govulncheck` on this module. Read the help (`go run golang.org/x/vuln/cmd/govulncheck@latest -h`) and note the difference between default symbol mode and `-show verbose`.
4. Search your own desk repo for `sk-`, `password`, and `BEGIN`. If a hit is a real secret, rotate it before you do anything else.

Code Generation and Embedding

Code Generation and Embedding

The boring default for files the binary must always have (banners, small HTML, default copy) is `//go:embed`. The boring default for code you are tempted to generate is **do not generate it** until a human-written file is worse. When you do generate, `go generate` is a documented `go run` of a helper you own — not a zoo of plugins.

Mental model

`go:embed` is a compiler feature. You write a directive above a `string`, `[]byte`, or `embed.FS` variable. The file contents are baked into the binary at compile time. There is no extra tool.

`go generate` is not a compiler feature. It is a convention: comments of the form `//go:generate command` are run by `go generate`. The command can be anything. The boring command is `go run some_helper.go` with a `//go:build ignore` tag so the helper is not part of your program.

Prefer embed. Reach for generate when the generated Go is mechanical (tables, stringers you would otherwise mistype) and the helper is small enough to read.

Worked examples

Create an empty directory. Save `go.mod`:

```
module example.com/desk

go 1.27
```

Case 1: Embed a text file as a string

Create `hello.txt` **next to** `main.go`. The compiler only embeds files in the same package directory (or a subdirectory you name). This book does not ship the file; you type it.

`hello.txt` (include the trailing newline):

```
desk is open
```

Save as `main.go`:

```
// main.go
package main

import (
    _ "embed"
    "fmt"
)

//go:embed hello.txt
var greeting string

func main() {
    fmt.Print(greeting)
}
```

The blank import of `embed` is required even though you never mention the name `embed`. The directive is what the compiler looks for.

Run (from the directory that contains `go.mod`, `main.go`, and `hello.txt`):

```
go run .
```

Output:

```
desk is open
```

`fmt.Print` does not add a newline. The line break you see comes from the file.

Case 2: Embed the same file as bytes, then as a file system

Bytes are useful when the file is not valid UTF-8 or when you will pass it to something that wants `[]byte`. `embed.FS` is useful when you have several files and want `ReadFile`.

Save as `main.go` (keep `hello.txt` and `go.mod`):

```
// main.go
package main

import (
    "embed"
    "fmt"
    "os"
)

//go:embed hello.txt
var greeting []byte
```

```
//go:embed hello.txt
var files embed.FS

func main() {
    fmt.Printf("bytes: %q\n", greeting)
    b, err := files.ReadFile("hello.txt")
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    fmt.Printf("fs: %q\n", b)
}
```

Run:

```
go run .
```

Output:

```
bytes: "desk is open\n"
fs: "desk is open\n"
```

If your `hello.txt` has no newline at the end, the `\n` inside the quotes will be missing. Match the file you actually saved.

The path in `ReadFile` is relative to the package directory, using slash separators, and must match the embed pattern.

Case 3: `go generate` with a helper you can read

Embed is enough for banners. Generation is for **Go source** you do not want to type. Here the helper writes `banner.go` with a single constant. No extra module, no `stringer` install.

Save as `main.go`:

```
// main.go
package main

import "fmt"

//go:generate go run gen_banner.go

func main() {
    fmt.Println(banner)
}
```

Save as `gen_banner.go`. The `ignore` build tag keeps this file out of `go run .` and `go build` so you do not have two `main` functions:

```
// gen_banner.go
//go:build ignore

package main

import (
    "fmt"
    "os"
)

func main() {
    const src = "package main\n\nconst banner = \"FRONT DESK\"\n"
    if err := os.WriteFile("banner.go", []byte(src), 0o644); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}
```

From the module directory:

```
go generate .
```

No output on success. The helper writes `banner.go`:

```
// banner.go
package main

const banner = "FRONT DESK"
```

Then:

```
go run .
```

Output:

```
FRONT DESK
```

If you skip `go generate`, `go run .` fails with `undefined: banner`. Commit `banner.go` if the rest of the team should build without running generate. Or run generate in CI before `go test`. Pick one and document it; do not require a mystery binary named `stringer` for a one-line constant.

The trap

Generating code because a blog showed `go:generate stringer` is how a desk tool grows a toolchain nobody can install on a bad wifi day.

This program does not need a generator. A typed constant is enough:

```go
// shift.go
package main

import "fmt"

type Shift int

const (
    Morning Shift = iota
    Evening
)

func (s Shift) String() string {
    switch s {
    case Morning:
        return "morning"
    case Evening:
        return "evening"
    default:
        return fmt.Sprintf("Shift(%d)", int(s))
    }
}

func main() {
    fmt.Println(Morning)
    fmt.Println(Evening)
}
```

Run:

```
go run shift.go
```

Output:

```
morning
evening
```

Write the `String` method. When you have twenty values and they drift, then a helper like Case 3 is earned. Embed still wins for files that are not Go.

The boring rule

- Put small static files next to the package and `//go:embed` them. Import `_ "embed"` or `"embed"`.
- Run `go run .` (a package), not a lone file list, so embed patterns resolve.

- Do not fetch a generator you have not read. `go run helper.go` with `//go:build ignore` is the whole pattern.
- Commit generated output **or** run generate in CI. Not neither.
- Never embed secrets. Embed is for public copy and assets, not keys.

Try this

1. Change `hello.txt` to two lines. Re-run Case 1. Confirm both lines appear.
2. Add `closed.txt` with the text `desk is closed`. Embed both files with `//go:embed hello.txt closed.txt` into one `embed.FS` and print each.
3. Point `gen_banner.go` at a different constant name and fix `main.go` to match. Run `go generate .` then `go run .`.
4. Delete `banner.go` and run `go run .` without generate. Read the error. Restore the file with `go generate .`.

Building and Releasing Software

Building and Releasing Software

The boring release is a **single static-ish binary** from `go build`, with a **version string you can `grep`**, built with `CGO_ENABLED=0` unless you have a C dependency you cannot drop. You do not need a container to prove the desk CLI works. You need a file you can copy.

Mental model

`go run` compiles a temporary binary and executes it. `go build` writes a binary you keep. The same compiler is in both paths.

Cross-compilation is two environment variables:

- `GOOS` — the target operating system (`linux`, `darwin`, `windows`, …)
- `GOARCH` — the target architecture (`amd64`, `arm64`, …)

`ldflags` passes linker flags. The one you will actually use is `-X main.version=…` (or another package-level string) so the binary can print what it is.

`CGO_ENABLED=0` tells the toolchain not to call a C compiler. Pure Go then produces a binary that does not dynamically link `libc`. That is the boring default for a desk tool with no cgo.

Worked examples

Save `go.mod` in an empty directory:

```
module example.com/desk

go 1.27
```

Case 1: `go build` writes a file you can run

Save as `main.go`:

```go
// main.go
package main

import "fmt"

func main() {
```

```
    fmt.Println("desk is open")
}
```

Build and run (names the output `desk` so you do not get a binary named after the directory):

```
go build -o desk .
```

No output on success. Then:

```
./desk
```

Output:

```
desk is open
```

`go run .` and `./desk` should print the same line. Ship the file `desk`, not your laptop's GOPATH.

Case 2: Stamp a version with `-ldflags -X`

Save as `main.go`:

```
// main.go
package main

import "fmt"

var version = "dev"

func main() {
    fmt.Printf("desk %s\n", version)
}
```

Default build (no flags):

```
go build -o desk .
./desk
```

Output:

```
desk dev
```

Release build. `-X` needs the **full package path** plus the variable name. For `package main` in this module that is `main.version`:

```
go build -ldflags="-X main.version=1.2.3" -o desk .
./desk
```

Output:

```
desk 1.2.3
```

The source still says `"dev"`. The linker overwrites the string in the binary. Use the git tag or CI build number in the flag, not a hand-edited constant you will forget.

Case 3: Cross-compile with `GOOS` and `GOARCH`

Same `main.go` as Case 2. Build a Linux amd64 binary from whatever OS you are on:

```
GOOS=linux GOARCH=amd64 go build -ldflags="-X main.version=1.2.3" -o desk-linux-amd64 .
```

No output on success. You now have a file named `desk-linux-amd64`. If your machine is already linux/amd64, `./desk-linux-amd64` runs. If it is not, copy the file to a Linux amd64 host — do not expect macOS or Windows to execute it.

Windows as a target (you cannot run this on Linux without extra machinery; building it is enough):

```
GOOS=windows GOARCH=amd64 go build -o desk.exe .
```

No output on success. The `.exe` suffix is a convention for Windows; Go does not require it, but operators will thank you.

List what Go knows:

```
go tool dist list
```

The output is a long list of `os/arch` pairs (`linux/amd64`, `darwin/arm64`, …). Pick the pair your operators actually run.

Case 4: `CGO_ENABLED=0` as the boring default

Same program. Force cgo off so a pure-Go build does not pick up a C toolchain by accident:

```
CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -ldflags="-X main.version=1.2.3" -o desk .
./desk
```

Output (on linux/amd64):

```
desk 1.2.3
```

If the program never `import "C"` and never uses a package that does, this binary is the one you copy between Linux machines. When you **do** need cgo, this flag will fail the build — that failure is useful. See the later chapter on cgo; until then, keep it off.

You can combine the variables:

```
CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -ldflags="-X main.version=1.2.3" -o desk .
```

That line is a complete release command for a pure-Go desk CLI.

Case 5: Strip debug info with `-ldflags="-s -w"`

By default, `go build` includes full symbol tables and DWARF debugging information so tools like `gdb` or `delve` can attach to the process. For production binaries, stripping these tables reduces binary size significantly (often by 30% to 40%) without affecting runtime performance or panic stack traces.

- `-s` strips the symbol table.
- `-w` disables DWARF debugging info generation.

```
go build -ldflags="-s -w -X main.version=1.2.3" -o desk-small .
```

Check the size difference:

```
ls -lh desk desk-small
```

Typical output:

```
-rwxr-xr-x 1 runner staff 2.4M Sep 10 12:00 desk
-rwxr-xr-x 1 runner staff 1.6M Sep 10 12:00 desk-small
```

Stack traces on panic will still print line numbers and function names because Go's runtime uses a separate internal line table.

Case 6: Inspect embedded build metadata at runtime

Save as `build_info.go`. Starting in Go 1.18+, compiled binaries automatically contain VCS revision metadata, dirty status, and compiler flags embedded by the toolchain. Your program can read its own pedigree without custom linker flags.

```
// build_info.go
package main

import (
    "fmt"
    "runtime/debug"
)

func main() {
    info, ok := debug.ReadBuildInfo()
    if !ok {
        fmt.Println("no build info available")
```

```
        return
    }

    fmt.Println("Go version:", info.GoVersion)
    fmt.Println("Path:", info.Path)
    for _, s := range info.Settings {
        if s.Key == "vcs.revision" || s.Key == "vcs.time" || s.Key == "CGO_ENABLED" {
            fmt.Printf("%s: %s\n", s.Key, s.Value)
        }
    }
}
```

Run:

```
go run build_info.go
```

Output:

```
Go version: go1.27.1
Path: command-line-arguments
CGO_ENABLED: 1
```

When built inside a git repository with `go build`, `info.Settings` automatically reports `vcs.revision` (the git commit hash) and `vcs.modified` (whether uncommitted changes were present).

The trap

Shipping whatever `go build` produced on a developer laptop, with cgo left on, is how you discover in production that `glibc` versions differ.

This program is still pure Go. The trap is the **command**, not the source:

```
// main.go
package main

import "fmt"

var version = "dev"

func main() {
    fmt.Printf("desk %s\n", version)
}
```

A sloppy release:

```
go build -o desk .
```

You get a binary. It may link `libc`. It prints `desk dev`. Nobody can tell which commit it is.

The boring release for this file:

```
CGO_ENABLED=0 go build -ldflags="-s -w -X main.version=1.2.3" -o desk .
./desk
```

Output:

```
desk 1.2.3
```

Pin `GOOS/GOARCH` in CI to the machines you support. Put the same command in a `README` line, not in a 400-line `Makefile` that calls Docker to compile 80 lines of Go.

The boring rule

- `go build -o <name> .` is the whole compile step for a `main` package.
- Stamp `version` with `-ldflags="-s -w -X main.version=…"`.
- Strip symbols with `-s -w` for production binaries to cut binary size.
- Inspect `runtime/debug.ReadBuildInfo()` for embedded toolchain and VCS revision information.
- Set `GOOS` and `GOARCH` in CI for each file you intend to publish.
- Default `CGO_ENABLED=0` for pure Go. Turn cgo on for a reason you can say out loud.
- Do not commit the binary. Commit the command that builds it.

Try this

1. Change `version` in source to `"local"` and rebuild **without** `-X`. Confirm `./desk` prints `desk local`. Then rebuild with `-X main.version=9.9.9` and confirm the source did not have to change.
2. Build `build_info.go` with `go build -o info_bin build_info.go` and run `./info_bin` to see the compiled binary's build settings.
3. Compare file sizes between `go build` and `go build -ldflags="-s -w"`. Note the size reduction.
4. Run `GOOS=darwin GOARCH=arm64 go build -o desk-mac .` then `file desk-mac` (or `ls -l desk-mac`) so you see a file you cannot run on Linux.
5. Run `CGO_ENABLED=0 go env CGO_ENABLED` and `go env GOOS GOARCH` so you know your machine's defaults.

Documentation and APIs

Documentation and APIs

The boring API is a **small exported surface** with **comments that `go doc` can print** and **example tests that `go test` runs**. Compatibility is a promise: do not change a signature because a new name feels nicer. Change it when the old one is wrong, and then bump a major module version.

Mental model

In Go, the documentation **is** the comments above exported names, in the same file as the code. `go doc` reads those comments. There is no parallel docbook.

An **example test** is a function named `Example`, `ExampleT`, or `ExampleT_M` in a `_test.go` file. If it has an `// Output:` comment, `go test` compiles the function, runs it, and compares stdout to that comment. The example shows up in `go doc` and on pkg.go.dev.

The Go 1 compatibility promise is the standard library's rule: exported signatures stay. Your module is not the standard library, but the same habit is how a desk API lasts more than one quarter.

Worked examples

Directory layout:

```
desk/
  go.mod
  ticket/ticket.go
  ticket/example_test.go
```

Save `go.mod` at `desk/`:

```
module example.com/desk

go 1.27
```

Case 1: Package and function comments that `go doc` prints

Save as `ticket/ticket.go`. The package comment starts with `Package ticket` — `go doc` looks for that prefix.

```
// ticket.go

// Package ticket is the public API for opening desk tickets.
//
// Call Open, then Label. Zero values are not a valid ticket.
package ticket

import "fmt"

// Ticket is an opened desk ticket. Construct with Open, not a struct literal.
type Ticket struct {
    id    int
    table int
}

// Open returns a ticket for the given id and table.
// It returns an error if id or table is not positive.
func Open(id, table int) (Ticket, error) {
    if id <= 0 {
        return Ticket{}, fmt.Errorf("ticket id must be positive")
    }
    if table <= 0 {
        return Ticket{}, fmt.Errorf("table must be positive")
    }
    return Ticket{id: id, table: table}, nil
}

// Label is the one-line form operators print on a stub.
func (t Ticket) Label() string {
    return fmt.Sprintf("ticket %d → table %d", t.id, t.table)
}
```

From the `desk/` directory (the module root):

```
go doc ./ticket
```

Output (word wrap may differ):

```
package ticket // import "example.com/desk/ticket"

Package ticket is the public API for opening desk tickets.

Call Open, then Label. Zero values are not a valid ticket.

func Open(id, table int) (Ticket, error)
type Ticket struct
```

One symbol:

```
go doc ./ticket.Open
```

Output:

```
package ticket // import "example.com/desk/ticket"

func Open(id, table int) (Ticket, error)
    Open returns a ticket for the given id and table. It returns an error if id
    or table is not positive.
```

The comment is the API. If you would not say it to a new teammate, do not export the name.

Case 2: An example test is documentation that can fail

Save as `ticket/example_test.go`. External test package (`ticket_test`) documents the API the way a caller sees it.

```
// example_test.go
package ticket_test

import (
    "fmt"

    "example.com/desk/ticket"
)

func ExampleOpen() {
    t, err := ticket.Open(7, 12)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(t.Label())
    // Output:
    // ticket 7 → table 12
}
```

Run:

```
go test ./ticket
```

Output:

```
PASS
ok      example.com/desk/ticket 0.002s
```

`go test -v ./ticket` names the example:

```
=== RUN   ExampleOpen
--- PASS: ExampleOpen (0.00s)
PASS
ok      example.com/desk/ticket 0.002s
```

If you change `Label` and forget the example, the test fails. That is the point. `go doc ./ticket.Open` will also list the example once it lives next to the package.

Case 3: A caller that only uses the stable surface

Save as `main.go` at the module root (`desk/main.go`):

```
// main.go
package main

import (
    "fmt"
    "os"

    "example.com/desk/ticket"
)

func main() {
    t, err := ticket.Open(7, 12)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    fmt.Println(t.Label())
}
```

Run:

```
go run .
```

Output:

```
ticket 7 → table 12
```

The caller never mentions `id` or `table` fields. You can add an unexported `note string` to `Ticket` later without touching this file.

The trap

Renaming `Open` to `NewTicket` because a review asked for "consistency" is a compatibility break. Every caller recompiles with edits. This program still works — the damage is social:

Keep `ticket/ticket.go` from Case 1. Save this as `main.go` (replace Case 3's `main.go` if it is still there):

```
// main.go
package main

import (
    "fmt"
    "os"

    "example.com/desk/ticket"
)

func main() {
    // After a "cleanup" that renamed Open → NewTicket, this line would not compile:
    // t, err := ticket.NewTicket(7, 12)
    t, err := ticket.Open(7, 12)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    fmt.Println(t.Label())
}
```

Run from the module root:

```
go run .
```

Output:

```
ticket 7 → table 12
```

The trap is the commented line. If you already published `Open`, leave it. Add a name only when it does something different. Changing parameter order, changing a field from `int` to `string`, or returning `*Ticket` instead of `Ticket` are all new APIs. They belong in `example.com/desk/v2`, not in a quiet Tuesday commit.

Unkeyed struct literals of **exported** structs are also fragile. That is why `Ticket`'s fields stay unexported and `Open` is the constructor.

The boring rule

- Comment every exported name. Start package comments with `Package <name>`.
- Run `go doc ./…` on the packages you ship. If you cannot explain a name in one sentence, hide it.
- Add `Example…` functions with `// Output:` for the two or three calls you want copied.
- Do not change exported signatures, parameter order, or types on a whim.
- Prefer unexported fields plus a constructor over public structs that callers fill in.
- A new major module path is how you break things on purpose.

Try this

1. Add `// ID returns the ticket id.` and a method `ID() int`. Run `go doc ./ticket.Ticket.ID`.
2. In `ExampleOpen`, change the expected output to a wrong string. Run `go test ./ticket` and read the diff. Restore it.
3. Add `ExampleOpen_badTable` that calls `Open(7, 0)` and expects the error line. `go test ./ticket`.
4. Try to write `ticket.Ticket{id: 7}` from `main.go`. Read the compile error. That error is the API working.

Long Term

Reflection in Practice

Reflection in Practice

The boring default is **not to use `reflect`**. Write a method that knows the type. Reach for reflection when you are talking to a format you do not control (a dump, a generic log line) and the set of fields is the point. It is a last resort, not a design style.

Mental model

The `reflect` package lets a program ask a value what it is at run time: its `Type`, its `Kind` (struct, int, pointer, …), and its fields. That is how `fmt` prints `%v` for types it has never seen, and how `encoding/json` walks structs.

The cost is real: no compile-time check that a field exists, panics on unexported fields if you call `Interface()`, and code that reads like a debugger. If you know the type, you do not need any of it.

Worked examples

Case 1: A tiny field dump for a desk order

Save as `dump.go`. This is the honest use: print exported fields of a struct you might otherwise pretty-print by hand.

```
// dump.go
package main

import (
    "fmt"
    "reflect"
)

type Order struct {
    ID     int
    Table  int
    Item   string
    Closed bool
}

func dump(v any) {
```

```
    rv := reflect.ValueOf(v)
    if rv.Kind() == reflect.Pointer {
        rv = rv.Elem()
    }
    if rv.Kind() != reflect.Struct {
        fmt.Printf("%v\n", v)
        return
    }
    rt := rv.Type()
    for i := range rt.NumField() {
        f := rt.Field(i)
        fmt.Printf("%s = %v\n", f.Name, rv.Field(i).Interface())
    }
}

func main() {
    dump(Order{ID: 7, Table: 12, Item: "soup", Closed: false})
}
```

Run:

```
go run dump.go
```

Output:

```
ID = 7
Table = 12
Item = soup
Closed = false
```

`for i := range rt.NumField()` is the Go 1.22+ integer range. Each field is looked up by index, not by a string you typed twice.

Case 2: The same dump without reflection

When the type is yours, a method is the whole API. Save as `label.go`:

```
// label.go
package main

import "fmt"

type Order struct {
    ID     int
    Table  int
    Item   string
```

```
    Closed bool
}

func (o Order) Dump() string {
    return fmt.Sprintf("ID = %d\nTable = %d\nItem = %s\nClosed = %v", o.ID, o.Table, o.Item,
}

func main() {
    fmt.Println(Order{ID: 7, Table: 12, Item: "soup", Closed: false}.Dump())
}
```

Run:

```
go run label.go
```

Output:

```
ID = 7
Table = 12
Item = soup
Closed = false
```

This version renames a field at compile time. Case 1 silently skips a field you forgot to care about, or panics if you later unexport it. Prefer Case 2 inside the desk. Keep Case 1 for tools that must accept `any`.

Case 3: Kinds you actually switch on

Reflection without a kind check is how you panic on a nil pointer. Save as `kind.go`:

```
// kind.go
package main

import (
    "fmt"
    "reflect"
)

func kindOf(v any) {
    if v == nil {
        fmt.Println("nil")
        return
    }
    fmt.Printf("%s (%s)\n", reflect.TypeOf(v), reflect.ValueOf(v).Kind())
}
```

```
func main() {
    kindOf(7)
    kindOf("soup")
    kindOf(struct{ Table int }{Table: 12})
    var p *int
    kindOf(p)
    kindOf((*int)(nil))
}
```

Run:

```
go run kind.go
```

Output:

```
int (int)
string (string)
struct { Table int } (struct)
*int (ptr)
*int (ptr)
```

A typed nil pointer is not the `any` containing `nil`. Check both if the value came from an interface.

The trap

Calling `Interface()` on an unexported field panics. This is not a compiler error. Save as `panic_dump.go`:

```
// panic_dump.go
package main

import (
    "fmt"
    "reflect"
)

type order struct {
    id int
}

func main() {
    o := order{id: 7}
    v := reflect.ValueOf(o)
    f := v.Field(0)
    fmt.Println(f.Interface())
}
```

Run:

```
go run panic_dump.go
```

Output (stack trimmed):

```
panic: reflect.Value.Interface: cannot return value obtained from unexported field or method
```

The fix is not `unsafe`. Export the field, or print it from a method on `order` in the same package (`f.Int()` still works for an unexported int in the same package, but you are now writing a debugger). The boring fix:

```
// safe_dump.go
package main

import "fmt"

type order struct {
    id int
}

func (o order) ID() int { return o.id }

func main() {
    fmt.Println(order{id: 7}.ID())
}
```

Run:

```
go run safe_dump.go
```

Output:

```
7
```

If you find yourself writing a mini `encoding/json` for the desk, stop and use `encoding/json`.

The boring rule

- Do not design your package around `any` plus reflect. Design around a type.
- Use reflect when the type is not known at compile time (fmt-like printers, generic dumps).
- Always check `Kind` (and nil) before `Elem` or `Field`.
- Never call `Interface()` on unexported fields.
- `encoding/json` and `fmt` already walked this path. Import them before you re-implement them.

Try this

1. In `dump.go`, pass `&Order{ID: 1, Table: 2, Item: "tea"}` and confirm the pointer branch still prints fields.
2. Pass `42` to `dump` and confirm it takes the non-struct branch.
3. Add an unexported field `note string` to `Order` in `dump.go`. Run it. Decide whether you wanted that field in the dump. If not, Case 2 was the better API.
4. Replace the dump with `json.MarshalIndent` on `Order` and print the bytes. That is the standard-library version of Case 1.

Unsafe Code

Unsafe Code

The boring default is **never to import `unsafe`**. The one use this book will show is **`unsafe.Sizeof`** (and the idea of `Alignof`) so you can see how big a value is. That is a measurement. It is not a licence to poke memory.

Mental model

Go's type system does not let you treat an arbitrary integer as a pointer. Package `unsafe` does. The compiler and the garbage collector have a **contract** with `unsafe.Pointer`. If you violate it, you do not get a nice error. You get a binary that is wrong on the next toolchain release.

`unsafe.Sizeof(x)` returns the size in bytes of `x`'s **representation** as a `uintptr`. It does not allocate. It does not follow pointers: the size of a `string` is the header (pointer plus length), not the bytes of the text.

`unsafe.Alignof(x)` is the alignment the compiler uses for that type. This chapter does not call it in code. You can see alignment **effects** with `Sizeof` on structs that include padding.

We will not take `unsafe.Pointer` to a `uintptr` and back. That is how people write exploits and also how people write "clever" code that breaks when the GC moves. Do not.

Worked examples

Sizes below assume a typical **64-bit** machine (`amd64` or `arm64`). On 32-bit, `int` and headers shrink. Measure on the machine you ship.

Case 1: Size of the types you already use

Save as `sizes.go`:

```
// sizes.go
package main

import (
    "fmt"
    "unsafe"
)

type Ticket struct {
```

```
    ID    int
    Table int
}

func main() {
    var n int
    var s string
    var t Ticket
    var p *Ticket
    fmt.Println("int", unsafe.Sizeof(n))
    fmt.Println("string", unsafe.Sizeof(s))
    fmt.Println("Ticket", unsafe.Sizeof(t))
    fmt.Println("*Ticket", unsafe.Sizeof(p))
}
```

Run:

```
go run sizes.go
```

Typical output on 64-bit:

```
int 8
string 16
Ticket 16
*Ticket 8
```

`string` is 16 because it is a pointer and a length, each 8 bytes. The characters live elsewhere. `Ticket` is two `int`s. A pointer is one machine word.

Case 2: Padding is visible as a larger `Sizeof`

Save as `padding.go`. Both structs hold one `int64` and two `bool`s. Field order changes the size because the compiler inserts padding to align `int64`.

```
// padding.go
package main

import (
    "fmt"
    "unsafe"
)

type Waste struct {
    A bool
    B int64
    C bool
```

```
}

type Tight struct {
    B int64
    A bool
    C bool
}

func main() {
    fmt.Println("Waste", unsafe.Sizeof(Waste{}))
    fmt.Println("Tight", unsafe.Sizeof(Tight{}))
}
```

Run:

```
go run padding.go
```

Typical output on 64-bit:

```
Waste 24
Tight 16
```

`Waste` is $1 + 7$ pad $+ 8 + 1 + 7$ pad $= 24$. `Tight` is $8 + 1 + 1 + 6$ pad $= 16$. That is interesting if you have millions of these in a slice. It is not a reason to reorder a desk `Ticket` with three fields.

Case 3: A slice header is not the backing array

Save as `slice_size.go`:

```
// slice_size.go
package main

import (
    "fmt"
    "unsafe"
)

func main() {
    small := []int{1, 2, 3}
    large := make([]int, 10_000)
    fmt.Println("small header", unsafe.Sizeof(small))
    fmt.Println("large header", unsafe.Sizeof(large))
    fmt.Println("one int", unsafe.Sizeof(small[0]))
}
```

Run:

```
go run slice_size.go
```

Typical output on 64-bit:

```
small header 24
large header 24
one int 8
```

Both headers are 24 bytes (pointer, length, capacity). `Sizeof` does not report the 10_000 integers. Do not use it to estimate memory of a slice; use `len` × element size plus the header, and remember capacity.

Case 4: Field alignment and padding with `Offsetof`

Save as `alignof.go`. Why did `Waste` take 24 bytes in Case 2 while `Tight` took 16? Hardware CPUs read memory in naturally aligned boundaries (e.g. an 8-byte `int64` must sit at a memory address divisible by 8). `unsafe.Alignof` reports the required alignment, and `unsafe.Offsetof` shows the exact byte offset of each field within the struct.

```
// alignof.go
package main

import (
    "fmt"
    "unsafe"
)

type Ticket struct {
    Active bool
    Table  int64
    ID     int32
}

func main() {
    var t Ticket
    fmt.Println("size of Ticket:", unsafe.Sizeof(t))
    fmt.Println("align of bool:", unsafe.Alignof(t.Active))
    fmt.Println("align of int64:", unsafe.Alignof(t.Table))
    fmt.Println("offset of Active:", unsafe.Offsetof(t.Active))
    fmt.Println("offset of Table:", unsafe.Offsetof(t.Table))
    fmt.Println("offset of ID:", unsafe.Offsetof(t.ID))
}
```

Run:

```
go run alignof.go
```

Typical output on 64-bit:

```
size of Ticket: 24
align of bool: 1
align of int64: 8
offset of Active: 0
offset of Table: 8
offset of ID: 16
```

`Active` starts at byte 0 and takes 1 byte. Because `Table` requires 8-byte alignment, the compiler leaves bytes 1 through 7 unused as padding. `Table` occupies bytes 8 through 15. `ID` occupies bytes 16 through 19. Finally, 4 trailing padding bytes are added so that an array of `Ticket` structs preserves the 8-byte alignment for the next element (total 24 bytes).

The trap

Importing `unsafe` so you can convert a `[]byte` to a `string` without copying looks fast and is a contract with the compiler you probably do not want. This chapter will **not** show that conversion.

The boring path copies. Save as `copy_string.go`:

```
// copy_string.go
package main

import "fmt"

func main() {
    b := []byte("soup")
    s := string(b)
    b[0] = 'S'
    fmt.Println(s)
    fmt.Println(string(b))
}
```

Run:

```
go run copy_string.go
```

Output:

```
soup
Soup
```

`s` is independent of `b` because `string(b)` copied. That is the behaviour you want at the desk. A `unsafe.Pointer` cast that aliases the same bytes will make `s` change when `b` changes — or worse, when the GC reclaims `b`. Do not go looking for that spell.

`unsafe.Pointer` is a contract with the compiler. If you are not prepared to read the current `unsafe` package docs and the Go release notes every six months, you are not prepared to use it.

The boring rule

- Do not import `unsafe` in desk code.
- `unsafe.Sizeof`, `Alignof`, and `Offsetof` are diagnostic rulers. Use them in a throwaway program when you are curious about memory layout.
- Do not reorder fields for speed until a profile says a giant slice of that struct is hot.
- Do not use `unsafe.Pointer`. Not to dodge a copy, not to call C, not to "inspect" a slice.
- If a library requires you to pass `unsafe.Pointer`, read why, wrap it in one file, and keep it off the rest of the API.

Try this

1. In `sizes.go`, add a `bool` field to `Ticket` (first, then last). Print `Sizeof` each time. On 64-bit you will likely see padding.
2. In `alignof.go`, reorder the fields in `Ticket` to put `Table int64` first, then `ID int32`, then `Active bool`. Print `unsafe.Sizeof(t)` and observe how the size drops from 24 to 16 bytes.
3. Print `unsafe.Sizeof([3]int{})` and compare it to `unsafe.Sizeof([]int{})`. The array is three ints; the slice is a header.
4. Run `sizes.go` with `GOARCH=386` if you have a C compiler and cgo, or just read `go env GOARCH` and accept that 32-bit numbers differ. Do not invent a `Pointer` cast to "normalise" them.

Cgo and Foreign Function Interfaces

Cgo and Foreign Function Interfaces

The boring default is **pure Go**. Write the function in Go. Ship with `CGO_ENABLED=0`. A foreign function interface (calling C from Go, or Go from C) is a last resort for a library that does not exist in Go and cannot be rewritten in a weekend.

Mental model

cgo is the toolchain's bridge: a file that `import "C"` can contain a C snippet (or include a header) and call those functions as `C.name(...)`. The Go compiler invokes a C compiler. The binary typically links `libc`. Cross-compilation stops being two environment variables. `go test` gets slower. The resulting program is no longer a single obvious file you copy between Linux boxes.

An FFI is that bridge in the abstract. cgo is Go's built-in one. Other languages have `ctypes`, JNI, and so on. The cost is the same: two runtimes, two memory models, two failure modes.

If the C function is three lines, it is a Go function. That is Case 1.

Worked examples

Case 1: The pure-Go function you should ship

Save as `bump.go`. A table number goes up by one. No C.

```
// bump.go
package main

import "fmt"

func bump(n int) int {
    return n + 1
}

func main() {
    fmt.Println(bump(3))
}
```

Run:

```
go run bump.go
```

Output:

```
4
```

Build the boring binary:

```
CGO_ENABLED=0 go build -o bump bump.go
./bump
```

Output:

```
4
```

Stop here unless you have a reason you can write in a commit message.

Case 2: The same increment, optionally in C

This program needs a **C compiler** (`gcc` or `clang`) and `CGO_ENABLED=1` (the default on most developer machines). If `go run` complains that cgo is disabled or `CC` is missing, skip this case. The point of the chapter is Case 1.

The C comment must sit **immediately above** `import "C"`. Do not put `import "C"` inside a grouped import.

Save as `bump_c.go`:

```
// bump_c.go
package main

/*
int bump(int n) {
    return n + 1;
}
*/
import "C"
import "fmt"

func main() {
    fmt.Println(int(C.bump(3)))
}
```

Run:

```
CGO_ENABLED=1 go run bump_c.go
```

Output:

```
4
```

`C.bump` returns a C `int`. Convert it with `int(...)` before you print if you want a Go `int`. Types that look the same (`int`, `C.int`) are not the same type.

You cannot `CGO_ENABLED=0 go run bump_c.go`. That command fails. That failure is the tax you pay for Case 2 on every machine and every CI image.

Case 3: Cross-compile as the reason to stay in Go

Save Case 1 as `bump.go` again. Cross-compile without a C toolchain for the target:

```
CGO_ENABLED=0 GOOS=linux GOARCH=arm64 go build -o bump-arm64 bump.go
```

No output on success. You have a Linux arm64 binary. Repeat for `windows/amd64` if you need it.

Now imagine Case 2. You would need a C cross-compiler, headers, and a libc for each target. Teams that "just wrap OpenSSL" discover this on the first Windows build.

Case 4: C strings and manual memory management

When passing strings or byte buffers to C, `C.CString(s)` allocates a null-terminated copy on the C heap using `malloc()`. The Go garbage collector **does not know this memory exists**. You must explicitly free it with `C.free()`.

Save as `c_strings.go`:

```
// c_strings.go
package main

/*
#include <stdlib.h>
#include <string.h>

int c_length(const char* s) {
    return strlen(s);
}
*/
import "C"
import (
    "fmt"
    "unsafe"
)

func main() {
    // Allocate on C heap: Go GC will NEVER free this automatically
    cs := C.CString("desk ticket #101")
```

```
    defer C.free(unsafe.Pointer(cs)) // Must be freed explicitly

    len := int(C.c_length(cs))
    fmt.Println("string length from C:", len)
}
```

Run:

```
CGO_ENABLED=1 go run c_strings.go
```

Output:

```
string length from C: 16
```

If you omit `defer C.free(unsafe.Pointer(cs))`, every call leaks bytes into the process memory space that Go runtime GC cycles will never reclaim. That is the manual memory burden cgo introduces to your codebase.

The trap

Pulling in a C library on day one of a desk CLI because "the C version is the real one" is how a 40-line tool becomes a docker image.

This Go program already does the job. Wrapping C `printf` would not make it more correct:

```
// open.go
package main

import "fmt"

func main() {
    fmt.Println("opened table 3")
}
```

Run:

```
go run open.go
```

Output:

```
opened table 3
```

If you need TLS, use `crypto/tls`. If you need SQLite, prefer a pure-Go driver until you measure a reason not to. If you must call C, isolate `import "C"` in one package, test it hard, and keep it off the public API of the desk.

cgo also means:

1. **Stack switches**: Go goroutines have small growing stacks (starting at 2–4 KB); C functions require fixed POSIX stacks. Crossing the boundary forces stack switches and carries significant call overhead.
2. **Crash propagation**: A segfault in C immediately crashes the entire Go process — it cannot be caught by Go's `recover()`.
3. **Leaked memory**: Memory allocated with C `malloc` is invisible to Go GC and must be manually tracked and freed.

None of that is worth an increment function.

The boring rule

- Write it in Go first. Case 1 is the product.
- Use cgo only for a dependency that does not exist in Go and will not be rewritten.
- When passing Go strings to C with `C.CString`, immediately pair it with `defer C.free(unsafe.Pointer(cs))`.
- Keep `import "C"` in one file. Convert C types at the boundary.
- Default `CGO_ENABLED=0` in release builds so cgo cannot sneak in through a dependency.
- Do not use cgo to "go faster." Measure. The boundary switch overhead often wipes out C speed advantages.

Try this

1. Time `go build` on `bump.go` with `CGO_ENABLED=0` and, if you ran Case 2, on `bump_c.go` with cgo on. The difference is the C compiler.
2. In `c_strings.go`, verify what happens if you remove `#include <stdlib.h>`. Read the compiler error about `C.free`.
3. Run `CGO_ENABLED=0 go run bump_c.go` and read the error. Keep that error in mind when CI is a tiny image.
4. Change `bump` in Case 1 to add 10 instead of 1. Notice you did not edit C, install `gcc`, or worry about `C.int` overflow.
5. If you must practise cgo, add a C function `int tables(void) { return 12; }` and print `C.tables()`. Then delete the file and go back to Case 1.

Performance Tuning

Performance Tuning

The boring default is **measure first**. Write the desk so it is correct and readable. If it is slow, use `testing.B`, then `pprof`, then change the line the profile names. Do not micro-optimise ticket labels because a blog post mentioned allocs.

Mental model

A **benchmark** is a test named `BenchmarkXxx` with argument `*testing.B`. The tooling runs it until the number is stable and reports ns/op. `-benchmem` adds allocations per operation.

pprof is the profiler that ships with Go. You record a CPU or memory profile while a test (or a program) runs, then `go tool pprof` shows where time or bytes went.

Allocation-aware code means you notice when a hot loop builds strings with `+=` and copies the whole buffer every time. It does not mean object pools for a CLI that prints twenty tickets.

`for b.Loop() { ... }` is the Go 1.24+ way to write the benchmark body. The compiler keeps the loop from being erased. Prefer it over a manual `b.N` loop on Go 1.27.

Worked examples

Directory (library in a subfolder so `package main` can import it):

```
desk/
  go.mod
  join/join.go
  join/join_test.go
  main.go
```

Save `go.mod` at `desk/`:

```
module example.com/desk

go 1.27
```

Case 1: Two ways to join ticket ids, one of them copies too much

Save as `join/join.go`:

```go
// join.go
package join

import (
    "fmt"
    "strings"
)

func JoinPlus(ids []int) string {
    s := ""
    for _, id := range ids {
        s += fmt.Sprintf("#%d ", id)
    }
    return s
}

func JoinBuilder(ids []int) string {
    var b strings.Builder
    for _, id := range ids {
        fmt.Fprintf(&b, "#%d ", id)
    }
    return b.String()
}
```

join/join_test.go:

```go
// join_test.go
package join

import "testing"

var ids = []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

func BenchmarkJoinPlus(b *testing.B) {
    b.ReportAllocs()
    for b.Loop() {
        _ = JoinPlus(ids)
    }
}

func BenchmarkJoinBuilder(b *testing.B) {
    b.ReportAllocs()
    for b.Loop() {
        _ = JoinBuilder(ids)
    }
}
```

From the module root:

```
go test -bench=. -benchmem ./join
```

Example output (numbers move with the machine; the **shape** is the lesson):

```
goos: linux
goarch: amd64
pkg: example.com/desk/join
BenchmarkJoinPlus-8            300000          3800 ns/op         496 B/op         22 allocs/op
BenchmarkJoinBuilder-8         500000          2400 ns/op         160 B/op          4 allocs/op
PASS
ok      example.com/desk/join   2.100s
```

`JoinPlus` allocates more because each `+=` may copy the string so far. `JoinBuilder` grows a buffer. You only rewrite the desk to use the builder when this function is on the hot path **and** the numbers bother a real user.

Case 2: The program you actually ship

Save as `main.go` at the module root. The CLI uses the builder because we already measured. It does not use a pool, a cache, or `unsafe`.

```
// main.go
package main

import (
    "fmt"

    "example.com/desk/join"
)

func main() {
    fmt.Println(join.JoinBuilder([]int{7, 8, 9}))
}
```

Run:

```
go run .
```

Output:

```
#7 #8 #9
```

Case 3: `pprof` when a number is not enough

CPU profile of one benchmark:

```
go test -bench=BenchmarkJoinPlus -cpuprofile=cpu.pprof ./join
```

Then a text summary (no interactive UI):

```
go tool pprof -top cpu.pprof
```

Example top lines (names and percents will not match yours exactly):

```
File: join.test
Duration: 2.10s, Total samples = 1.80s (85.71%)
Showing nodes accounting for 1.80s, 100% of 1.80s
      flat  flat%   sum%        cum   cum%
     0.90s 50.00% 50.00%      0.90s 50.00%  runtime.mallocgc
     0.40s 22.22% 72.22%      1.20s 66.67%  example.com/desk/join.JoinPlus
     0.20s 11.11% 83.33%      0.50s 27.78%  fmt.Sprintf
```

If `mallocgc` and `JoinPlus` dominate, the allocation story from `-benchmem` is confirmed. Memory profile:

```
go test -bench=BenchmarkJoinPlus -memprofile=mem.pprof ./join
go tool pprof -top -alloc_space mem.pprof
```

Read the function names. Change **those**. Delete the `*.pprof` files when you are done; they are not source.

Case 4: Zero-cost speedup: preallocating slice capacity

Save as `alloc/alloc_test.go`. The simplest, highest-return optimization in Go services is giving `make` a capacity hint when the item count is known ahead of time. Growing a slice without a capacity hint triggers repeated reallocation and memory copying.

```
// alloc_test.go
package alloc

import "testing"

func BenchmarkAppendNoCap(b *testing.B) {
    b.ReportAllocs()
    for b.Loop() {
        var s []int
        for i := range 1000 {
            s = append(s, i)
        }
        _ = s
    }
}
```

```
func BenchmarkAppendWithCap(b *testing.B) {
    b.ReportAllocs()
    for b.Loop() {
        s := make([]int, 0, 1000)
        for i := range 1000 {
            s = append(s, i)
        }
        _ = s
    }
}
```

Run:

```
go test -bench=. -benchmem ./alloc
```

Typical output:

```
BenchmarkAppendNoCap-4        100320         14379 ns/op       25208 B/op         12 allocs/op
BenchmarkAppendWithCap-4      429908          2633 ns/op           0 B/op          0 allocs/op
```

By providing `make([]int, 0, 1000)`, allocations dropped from 12 per operation to 0 (the slice backing array is sized in one shot), and execution speed improved by over 5x. This requires no complex caching or object pools.

The trap

Rewriting the desk to avoid allocations before a user has waited on anything. This program is already fast enough:

```
// tickets.go
package main

import "fmt"

func main() {
    for _, id := range []int{7, 8, 9} {
        fmt.Printf("ticket %d\n", id)
    }
}
```

Run:

```
go run tickets.go
```

Output:

```
ticket 7
ticket 8
ticket 9
```

A pool of `[]byte`, a custom formatter, and a blog about "zero alloc" would not make this more correct. They would make Monday morning slower for the next reader. Benchmark when someone can feel a delay, or when a profile of a service says so. Not because twelve tickets allocate.

The boring rule

- Make it right. Then measure. Then change the hot function.
- `go test -bench=. -benchmem` is the first instrument.
- `go tool pprof -top` is the second. Do not guess.
- `strings.Builder` (or `bytes.Buffer`) beats `+=` in a loop. That is the usual win. Stop there.
- Do not pool, intern, or `unsafe`-cast the desk. Do not "optimise" `fmt.Printf` of three lines.

Try this

1. Grow `ids` in the test to 1_000 elements. Re-run `-benchmem`. Watch `JoinPlus` get worse faster than `JoinBuilder`.
2. Add `b.ResetTimer()` after building a large `ids` slice inside the benchmark (move setup above the loop). Confirm the ns/op still compares the join, not the slice make.
3. Record `-memprofile` for `BenchmarkJoinBuilder` and `-top -alloc_space`. See that the remaining alloc is mostly the result string.
4. Leave `tickets.go` as it is. Do not convert it to a builder.

Designing for the Long Term

Designing for the Long Term

The boring default is a **small exported API**, **package boundaries that match real jobs**, and **compatibility you treat as a promise**. The rest of this book is technique. This chapter is what still matters when the desk has been in production for a year.

Mental model

A **package** is a name the rest of the program can import. Everything exported (`Open`, `Ticket`, `Label`) is a promise. Everything unexported is yours to change on a Tuesday.

A **boundary** is a reason to split: "tickets" versus "shifts" versus `main`. Not "models" versus "helpers" versus `utils`. If two files always change together and nobody imports them separately, they are one package.

Compatibility means a caller that compiled against last month's module still compiles. You can add a method. You can add an unexported field. You cannot rename `Open`, shuffle its parameters, or change `Ticket` to `*Ticket` without a new major version (`example.com/desk/v2`).

Worked examples

Layout:

```
desk/
  go.mod
  ticket/ticket.go
  main.go
```

Save `go.mod`:

```
module example.com/desk

go 1.27
```

Case 1: A stable ticket API

Unexported fields, a constructor that validates, methods that return values. Callers cannot write struct literals that break when you add a field.

Save as `ticket/ticket.go`:

```
// ticket.go
package ticket

import "fmt"

// Ticket is an opened desk ticket. Use Open. Do not depend on the zero value.
type Ticket struct {
    id    int
    table int
}

// Open returns a ticket for a positive id and table.
func Open(id, table int) (Ticket, error) {
    if id <= 0 {
        return Ticket{}, fmt.Errorf("ticket id must be positive")
    }
    if table <= 0 {
        return Ticket{}, fmt.Errorf("table must be positive")
    }
    return Ticket{id: id, table: table}, nil
}

// ID is the ticket identifier.
func (t Ticket) ID() int { return t.id }

// Table is the table the ticket is for.
func (t Ticket) Table() int { return t.table }

// Label is the one-line stub.
func (t Ticket) Label() string {
    return fmt.Sprintf("ticket %d → table %d", t.id, t.table)
}
```

Save as `main.go`:

```
// main.go
package main

import (
    "fmt"
    "os"

    "example.com/desk/ticket"
)

func main() {
    t, err := ticket.Open(7, 12)
```

```
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    fmt.Println(t.Label())
}
```

Run from the module root:

```
go run .
```

Output:

```
ticket 7 → table 12
```

This is the surface you freeze. Internals can grow.

Case 2: Add behaviour without breaking callers

A year later you need a note on the ticket. Add an unexported field and a method. `Open`'s signature does not change. `main.go` does not change.

Save as `ticket/ticket.go`:

```
// ticket.go
package ticket

import "fmt"

// Ticket is an opened desk ticket. Use Open. Do not depend on the zero value.
type Ticket struct {
    id    int
    table int
    note  string
}

// Open returns a ticket for a positive id and table.
func Open(id, table int) (Ticket, error) {
    if id <= 0 {
        return Ticket{}, fmt.Errorf("ticket id must be positive")
    }
    if table <= 0 {
        return Ticket{}, fmt.Errorf("table must be positive")
    }
    return Ticket{id: id, table: table}, nil
}
```

```
// WithNote returns a copy with a note. The original ticket is unchanged.
func (t Ticket) WithNote(note string) Ticket {
    t.note = note
    return t
}

// ID is the ticket identifier.
func (t Ticket) ID() int { return t.id }

// Table is the table the ticket is for.
func (t Ticket) Table() int { return t.table }

// Label is the one-line stub.
func (t Ticket) Label() string {
    if t.note == "" {
        return fmt.Sprintf("ticket %d → table %d", t.id, t.table)
    }
    return fmt.Sprintf("ticket %d → table %d (%s)", t.id, t.table, t.note)
}
```

The original `main.go` still runs:

```
go run .
```

Output:

```
ticket 7 → table 12
```

A new caller can opt in:

```
// note.go
package main

import (
    "fmt"
    "os"

    "example.com/desk/ticket"
)

func main() {
    t, err := ticket.Open(7, 12)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    t = t.WithNote("window")
```

```
    fmt.Println(t.Label())
}
```

Keep only one `main` in the directory. Replace `main.go` with that file, or run it as the module's `main.go`. Output of `go run .`:

```
ticket 7 → table 12 (window)
```

Old binaries that called `Open` and `Label` still match Case 1. New code asks for a note. That is how a package lasts.

Case 3: A boundary that is a real job

A second package for shifts, imported by `main`, not dumped into `ticket` as "related desk stuff." Save as `shift/shift.go`:

```
// shift.go
package shift

import "fmt"

// Name is a roster label such as "morning".
type Name string

const (
    Morning Name = "morning"
    Evening Name = "evening"
)

// Label prints the roster line for a person.
func Label(person string, n Name) string {
    return fmt.Sprintf("%s · %s", person, n)
}
```

Save as `main.go` (this binary talks to both packages; they do not import each other):

```
// main.go
package main

import (
    "fmt"
    "os"

    "example.com/desk/shift"
    "example.com/desk/ticket"
)
```

```
func main() {
    t, err := ticket.Open(7, 12)
    if err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
    fmt.Println(t.Label())
    fmt.Println(shift.Label("Amina", shift.Morning))
}
```

Run:

```
go run .
```

Output:

```
ticket 7 → table 12
Amina · morning
```

Do not import `shift` from `ticket` unless a ticket literally cannot exist without a shift. Cycles are a smell that the split was fake.

The trap

A `utils` package that every file imports. Six months later it holds string helpers, time parsing, and a half-written HTTP client. Names collide. Tests become a hairball.

This is the wrong home for `Open`:

```
// utils.go
package utils

import "fmt"

func OpenTicket(id, table int) string {
    return fmt.Sprintf("ticket %d → table %d", id, table)
}
```

```
// use_utils.go
package main

import "fmt"

func OpenTicket(id, table int) string {
    return fmt.Sprintf("ticket %d → table %d", id, table)
```

```
}

func main() {
    fmt.Println(OpenTicket(7, 12))
}
```

Run the second file (it does not need `utils`):

```
go run use_utils.go
```

Output:

```
ticket 7 → table 12
```

Put `Open` in `package ticket`. Put `main` in `package main`. If you only have twenty lines, **one** package `main` is fine. Split when a name wants two homes, or when another binary must import the API without importing `main`.

The boring rule

- Export a constructor and methods. Keep fields unexported unless the type is a bag of options you intend to grow with keyed literals only.
- Add methods and unexported fields freely. Do not rename or retag exported signatures without `v2`.
- Split packages along jobs (`ticket`, `shift`), not layers (`models`, `helpers`, `utils`).
- Recap from the rest of the book: `gofmt`, `go vet`, `go test`, `govulncheck`; errors as values; small interfaces; `CGO_ENABLED=0`; no reflect/unsafe/cgo by default; measure before you tune; secrets out of source; `//go:embed` for files; stamp a version with `-ldflags`.
- The code a stranger can read on Monday morning is the code you still want in a year.

Try this

1. Add `func (t Ticket) Empty() bool` that reports `t.id == 0`. Do not export a field to do it. Run `main.go`.
2. Try `ticket.Ticket{id: 7}` from `main.go` and read the compile error. Keep the fields unexported.
3. Add `package shift` as in Case 3 and print `shift.Label("Amina", shift.Morning)` from `main`. Keep `ticket` unaware of `shift`.
4. List the exported names in `ticket` (`go doc ./ticket`). If a name is not for callers, unexport it.

Appendices

Glossary

Glossary

Short definitions for words this book uses with a fixed meaning. If a chapter taught a default, that default is in the definition.

alignment. The address multiple a type prefers. `int64` wants 8-byte alignment on 64-bit. Padding in a struct exists to satisfy it. See `unsafe.Sizeof` in the unsafe chapter — as a ruler, not as a licence.

benchmark. A function `BenchmarkXxx(*testing.B)` run by `go test -bench`. Reports ns/op. `-benchmem` adds bytes and allocs per op.

build constraint (build tag). A `//go:build` line that includes or excludes a file. `//go:build ignore` keeps a `go generate` helper out of the package.

cgo. The bridge that lets a Go file `import "C"` and call C. Needs a C compiler. Breaks easy cross-compiles. Avoid unless you must.

channel. A typed pipe between goroutines. Send `ch <- v`, receive `v := <-ch`. Buffer size is set at `make`. Close only from the sender side.

compatibility promise. For the standard library: exported signatures stay. For your module: behave the same, or bump a major version (`/v2`).

constructor. An exported function (`Open`, `New`) that returns a valid value. Prefer this over exported struct fields that callers fill in.

context. `context.Context` carries cancellation, a deadline, and optional request values down a call chain. The first argument of an RPC-shaped function.

coverage. The fraction of statements `go test -cover` saw run. A number, not a goal of 100%.

embed (`//go:embed`). Compiler feature that bakes a file into a `string`, `[]byte`, or `embed.FS` at build time. Import `"embed"` or `_ "embed"`.

error. A value that implements `error`. Returned, wrapped with `%w`, inspected with `errors.Is` / `errors.As`. Not an exception.

escape analysis. The compiler's decision that a value must live on the heap (it "escapes") because a pointer to it outlives the stack frame. `go build -gcflags=-m` prints the decisions.

example test. `ExampleXxx` in a `_test.go` file. `// Output:` makes `go test` compare stdout. Shows up in `go doc`.

exported. A name that starts with a capital letter. Visible from other packages. Part of your API.

fuzz. `FuzzXxx(*testing.F)` — the testing tool feeds random inputs that still satisfy your seed corpus.

garbage collection (GC). Automatic reclaim of heap values that nothing points to. You still close files and stop goroutines; GC does not.

generate (`go generate`). Runs `//go:generate` commands. Not part of `go build`. Prefer a `go run` helper you own.

go.mod. The file that names a **module** and its Go version and requirements.

goroutine. A function the runtime can run concurrently. Started with `go f()`. Cheap compared to an OS thread. Not magic: you still need to wait, cancel, or deadlock.

govulncheck. Tool that matches your module graph (and call graph) against the Go vulnerability database.

interface. A set of methods. Satisfied implicitly: no `implements` keyword. Keep it small. Define it where it is used, when you have two real implementations.

iter.Seq / iter.Seq2. Type aliases for push-iterator functions (Go 1.23+). `iter.Seq[V]` is `func(yield func(V) bool)`; `iter.Seq2[K, V]` adds a key. `range` can iterate over them. `slices.Collect` and `maps.Collect` drain them into slices and maps.

ldflags `-X`. Linker flag that overwrites a package-level string at build time. Used to stamp `version`.

maps package. `"maps"` (Go 1.21+) provides generic functions over `map` values: `maps.Clone`, `maps.Copy`, `maps.Keys`, `maps.Values`, `maps.Equal`, `maps.DeleteFunc`. Everything you used to write with a `for range`.

method. A function with a receiver. `func (t Ticket) Label() string`. Value receiver copies; pointer receiver can mutate.

module. A versioned unit of source with a `go.mod` and an import path (`example.com/desk`). Contains one or more packages.

mutex. `sync.Mutex` (or `RWMutex`) for shared memory. Lock, copy of a mutex is a vet finding, unlock. Prefer channels when they fit.

package. A directory of Go files with the same `package` name. The import path is module path plus directory.

panic / recover. Abort the goroutine; `recover` only in `defer`. Not for ordinary errors.

pprof. Profiler shipped as `go tool pprof`. CPU and memory profiles from tests or `net/http/pprof`.

pointer. Address of a value. `*T`. Nil is the zero value. Do not share mutable pointers across goroutines without a lock or a single owner.

race detector. `go test -race`. Finds unsynchronised shared memory. Use it in CI.

reflection (`reflect`). Run-time inspection of types and values. Last resort. `fmt` and `encoding/json` already do the usual jobs.

slice. Header (pointer, length, capacity) plus a backing array. `append` may allocate a new array.

slices package. `"slices"` (Go 1.21+) provides generic functions over slices: `slices.Sort`, `slices.SortFunc`, `slices.Contains`, `slices.Index`, `slices.BinarySearch`, `slices.Compact`, `slices.Reverse`, `slices.Clone`, `slices.Collect`. Prefer over hand-written loops.

range-over-function. Since Go 1.23, `range` accepts a function with signature `func(yield func(V) bool)` or `func(yield func(K, V) bool)`. The runtime calls the function; the function calls `yield` per element; `break` causes `yield` to return `false`. See `iter.Seq` / `iter.Seq2`.

staticcheck. Optional extra static analyser. Not part of the `go` binary.

stringer. A common generator for `String()` methods. This book prefers a hand-written `String` or a tiny `go generate` helper.

table test. A slice of cases in a `TestXxx` function, looped with `t.Run`.

toolchain. The `go` command: run, build, test, vet, fmt, doc, mod.

unsafe. Package that can break the type system. This book uses `unsafe.Sizeof` as a ruler. `unsafe.Pointer` is a contract with the compiler; do not take it.

vet (`go vet`). Static checks that ship with Go. Wrong `Printf` verbs, copied locks, and similar. Run with tests.

workspace (`go.work`). A file that lists modules you are editing together. Local replace without publishing.

zero value. The value of a declared variable with no initialiser: `0`, `""`, `nil`, struct of zeros. Useful when it is a valid empty; dangerous when it is not (then use a constructor).

`CGO_ENABLED`. `0` skips the C compiler. Boring default for pure Go. `1` is required for cgo.

`GOOS` / `GOARCH`. Target OS and architecture for `go build`. Cross-compilation for pure Go is these two variables.

Command Cheat Sheet

Command Cheat Sheet

Commands this book actually uses. Run them from a module directory unless a filename is listed. Substitute `example.com/desk` for your module path.

Everyday

Command	What it does
`go run .`	Compile the package in this directory and run it
`go run file.go`	Compile and run listed files (no `go.mod` needed for a toy)
`go build -o desk .`	Write a binary named `desk`
`go test ./...`	Test every package under this directory
`go test -v ./...`	Same, with names of tests and examples
`gofmt -w .`	Rewrite Go files in place
`go fmt ./...`	Same idea, package-oriented
`go vet ./...`	Compiler-adjacent static checks
`go doc ./ticket`	Print package docs
`go doc ./ticket.Open`	Print one symbol
`go env GOOS GOARCH`	Print this machine's target defaults
`go version`	Print the toolchain version (this book: 1.27)

Modules

Command	What it does
`go mod init example.com/desk`	Create `go.mod`
`go mod tidy`	Add missing modules, drop unused ones
`go get example.com/mod@v1.2.3`	Add or bump a requirement
`go list -m all`	Print the module graph

Tests, benches, profiles

Command	What it does
`go test -cover ./...`	Statement coverage
`go test -race ./...`	Race detector

Command	What it does
`go test -bench=. -benchmem ./join`	Benchmarks plus allocs
`go test -bench=BenchmarkJoinPlus -cpuprofile=cpu.pprof ./join`	CPU profile
`go test -bench=BenchmarkJoinPlus -memprofile=mem.pprof ./join`	Memory profile
`go tool pprof -top cpu.pprof`	Text summary of a profile

Ship

Command	What it does
`CGO_ENABLED=0 go build -o desk .`	Pure-Go binary, no cgo
`go build -ldflags="-X main.version=1.2.3" -o desk .`	Stamp `var version` in `package main`
`GOOS=linux GOARCH=amd64 go build -o desk .`	Cross-compile
`GOOS=linux GOARCH=amd64 CGO_ENABLED=0 go build -ldflags="-X main.version=1.2.3" -o desk .`	Boring release line
`go generate .`	Run `//go:generate` lines
`go run golang.org/x/vuln/cmd/govulncheck@latest ./...`	Vulnerability scan
`go tool dist list`	All `GOOS/GOARCH` pairs

Optional extra

Command	What it does
`staticcheck ./...`	Third-party static analysis (install separately)

Standard library packages (quick reference)

Package	Since	Key items
`slices`	1.21	`Sort`, `SortFunc`, `Contains`, `BinarySearch`, `Compact`, `Reverse`, `Clone`, `Collect`
`maps`	1.21	`Clone`, `Copy`, `Keys`, `Values`, `Equal`, `DeleteFunc`, `Collect`
`cmp`	1.21	`Compare`, `Ordered` constraint
`iter`	1.23	`Seq[V]`, `Seq2[K,V]` — push-iterator types for range-over-function
`log/slog`	1.21	`NewTextHandler`, `NewJSONHandler`, `HandlerOptions`, `With`, `SetDefault`
`sync`	—	`Mutex`, `RWMutex`, `WaitGroup` (`.Go` since 1.25), `Once`, `Map`
`sync/atomic`	—	`Int64`, `Bool`, `Pointer[T]` — typed atomics
`context`	—	`Background`, `WithCancel`, `WithTimeout`, `WithCancelCause` (1.20), `Cause` (1.20)
`testing`	—	`T.Run`, `T.Helper`, `T.TempDir`, `T.Setenv`, `T.Context` (1.21), `B.Loop` (1.24)
`net/http`	—	`ServeMux` with method patterns (1.22), `PathValue` (1.22)

Reminder 1: `go run` is a compile

Save as `hello.go`:

```
// hello.go
package main

import "fmt"

func main() {
    fmt.Println("desk is open")
}
```

Run:

```
go run hello.go
```

Output:

```
desk is open
```

There is no interpreter. `go build -o hello hello.go` then `./hello` prints the same line.

Reminder 2: `go test` runs examples

Save as `go.mod`:

```
module example.com/desk

go 1.27
```

Save as `greet.go`:

```go
// greet.go
package desk

func Greet() string {
    return "desk is open"
}
```

Save as `greet_test.go`:

```go
// greet_test.go
package desk

import "fmt"

func ExampleGreet() {
    fmt.Println(Greet())
    // Output:
    // desk is open
}
```

Run:

```
go test -v .
```

Output:

```
=== RUN   ExampleGreet
--- PASS: ExampleGreet (0.00s)
PASS
ok      example.com/desk    0.002s
```

If the printed line drifts from `// Output:`, the test fails. That is documentation you cannot ignore.

The boring rule

- `fmt` / `vet` / `test` on every change.
- `govulncheck` and a stamped `CGO_ENABLED=0` build before you ship.
- When in doubt, `go doc cmd/go` is the long form of this page.

What's New: Go 1.21 – 1.27

What's New: Go 1.21 – 1.27

This appendix is a reference, not a tutorial. Each section lists what changed in that release and shows one complete program that demonstrates two or three of the most useful additions. All programs compile and run with Go 1.27.

The desk domain appears throughout: orders, tickets, prices, shifts.

Go 1.21

Feature	Package / layer	What it does
`min` / `max`	language builtin	Return the smallest or largest of two or more comparable values.
`clear`	language builtin	Zero all elements of a slice or delete all keys of a map.
`slices`	`slices`	Sort, search, compare, and transform slices with generic functions.
`maps`	`maps`	Copy, clone, and compare maps with generic functions.
`cmp`	`cmp`	`cmp.Compare`, `cmp.Or`, and the `cmp.Ordered` constraint.
Structured logging	`log/slog`	`slog.Info`, `slog.Error`, handler configuration, key-value pairs.
One-time helpers	`sync`	`sync.OnceFunc`, `sync.OnceValue`, `sync.OnceValues`.
Test context	`testing`	`t.Context()` returns a context that cancels when the test ends.

```
// desk121.go
package main

import (
```

```
    "cmp"
    "fmt"
    "log/slog"
    "os"
    "slices"
)

type Order struct {
    ID    int
    Item  string
    Price float64
}

func main() {
    orders := []Order{
        {1, "keyboard", 89.99},
        {2, "monitor", 349.00},
        {3, "mouse", 29.50},
        {4, "desk pad", 19.95},
    }

    // slices.SortFunc with cmp.Compare for a numeric field.
    slices.SortFunc(orders, func(a, b Order) int {
        return cmp.Compare(a.Price, b.Price)
    })

    logger := slog.New(slog.NewTextHandler(os.Stdout, nil))

    for _, o := range orders {
        logger.Info("order",
            "id", o.ID,
            "item", o.Item,
            "price", o.Price,
        )
    }

    // min / max builtins.
    cheapest := orders[0].Price
    priciest := orders[len(orders)-1].Price
    fmt.Printf("range: %.2f - %.2f\n", cheapest, priciest)

    // clear resets the slice elements to zero values.
    scratch := []int{10, 20, 30}
    clear(scratch)
    fmt.Println("after clear:", scratch)
}
```

Run:

```
go run desk121.go
```

Output:

```
time=... level=INFO msg=order id=4 item="desk pad" price=19.95
time=... level=INFO msg=order id=3 item=mouse price=29.5
time=... level=INFO msg=order id=1 item=keyboard price=89.99
time=... level=INFO msg=order id=2 item=monitor price=349
range: 19.95 - 349.00
after clear: [0 0 0]
```

Go 1.22

Feature	Package / layer	What it does
`for i := range n`	language	Integer range: iterates `i` from `0` to `n-1`. No `int` variable needed.
Per-iteration loop variable	language	Each loop body gets its own copy of `i` and `v`; goroutines no longer share the same address.
Method + path patterns	`net/http`	`mux.HandleFunc("GET /tickets/{id}", h)` matches method and captures `{id}`.
`r.PathValue`	`net/http`	`r.PathValue("id")` extracts a named segment from the matched route.

```
// desk122.go
package main

import (
    "fmt"
    "net/http"
    "strconv"
)

var tickets = map[int]string{
    1: "Replace keyboard",
    2: "Monitor flicker",
    3: "Desk lamp broken",
```

```
}

func ticketHandler(w http.ResponseWriter, r *http.Request) {
    raw := r.PathValue("id")
    id, err := strconv.Atoi(raw)
    if err != nil {
        http.Error(w, "bad id", http.StatusBadRequest)
        return
    }
    title, ok := tickets[id]
    if !ok {
        http.Error(w, "not found", http.StatusNotFound)
        return
    }
    fmt.Fprintf(w, "ticket %d: %s\n", id, title)
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /tickets/{id}", ticketHandler)

    // for i := range n - print ticket IDs without a separate counter variable.
    fmt.Println("known ticket IDs:")
    for i := range 3 {
        fmt.Println(" ", i+1)
    }

    fmt.Println("listening on :8122")
    if err := http.ListenAndServe(":8122", mux); err != nil {
        fmt.Println(err)
    }
}
```

Run:

```
go run desk122.go &
curl -s http://localhost:8122/tickets/2
```

Output:

```
known ticket IDs:
  1
  2
  3
listening on :8122
ticket 2: Monitor flicker
```

Go 1.23

Feature	Package / layer	What it does
`iter` package	`iter`	Defines `iter.Seq[V]` and `iter.Seq2[K,V]` — the function types a `range` loop can consume.
Range-over-function	language	`for v := range f` works when `f` is `func(yield func(V) bool)`.
`strings.SplitSeq`	`strings`	Like `strings.Split` but yields substrings one at a time via an iterator.
`strings.FieldsSeq`	`strings`	Lazy word-splitting iterator; no intermediate slice allocated.
`maps.Keys` / `maps.Values`	`maps`	Return `iter.Seq` iterators over map keys or values.
`time.Tick` GC fix	`time`	The ticker created by `time.Tick` is now garbage-collected when unreachable.

```
// desk123.go
package main

import (
    "fmt"
    "maps"
    "strings"
)

func main() {
    // strings.FieldsSeq: iterate words without building a []string.
    shift := "Monday Tuesday Wednesday Thursday Friday"
    fmt.Print("days: ")
    for day := range strings.FieldsSeq(shift) {
        fmt.Printf("%s ", day)
    }
    fmt.Println()

    // maps.Keys iterator - consume keys without an intermediate slice.
    prices := map[string]float64{
        "keyboard": 89.99,
        "monitor":  349.00,
        "mouse":    29.50,
```

```
    }

    fmt.Println("items in stock:")
    for item := range maps.Keys(prices) {
        fmt.Printf("  %s\n", item)
    }

    // strings.SplitSeq: lazy split on a delimiter.
    csvRow := "order_id,item,qty,price"
    fmt.Print("columns: ")
    for col := range strings.SplitSeq(csvRow, ",") {
        fmt.Printf("[%s]", col)
    }
    fmt.Println()
}
```

Run:

```
go run desk123.go
```

Output:

```
days: Monday Tuesday Wednesday Thursday Friday
items in stock:
  keyboard
  monitor
  mouse
columns: [order_id][item][qty][price]
```

Go 1.24

Feature	Package / layer	What it does
`b.Loop()`	`testing`	Replaces `for i := 0; i < b.N; i++`; handles timer reset automatically.
`testing/synctest` (experimental)	`testing/synctest`	Runs goroutines in a fake-time bubble to test concurrent code deterministically.
Weak pointers	`weak`	`weak.Pointer[T]` — a reference that does not keep an object alive; useful for caches.

Feature	Package / layer	What it does
`runtime.AddCleanup`	`runtime`	Attach a cleanup function to an object, called on collection (replaces `SetFinalizer`).
`sync/atomic And/Or`	`sync/atomic`	Bitwise AND and OR operations added to integer atomic types.

```
// desk124_test.go
package main

import (
    "testing"
)

// priceAfterDiscount applies a percentage discount.
func priceAfterDiscount(price, pct float64) float64 {
    return price * (1 - pct/100)
}

// BenchmarkDiscount uses b.Loop() - no manual b.N loop needed.
func BenchmarkDiscount(b *testing.B) {
    price := 349.00
    pct := 15.0
    var result float64
    for b.Loop() {
        result = priceAfterDiscount(price, pct)
    }
    _ = result
}

func TestDiscount(t *testing.T) {
    ctx := t.Context() // cancelled when the test ends
    _ = ctx

    got := priceAfterDiscount(100.0, 10.0)
    if got != 90.0 {
        t.Fatalf("want 90.0, got %f", got)
    }
}
```

Run:

```
go test -bench=. -benchmem ./desk124_test.go
```

Output:

```
goos: linux
goarch: amd64
BenchmarkDiscount-8   1000000000   0.2990 ns/op   0 B/op   0 allocs/op
PASS
```

Go 1.25

Feature	Package / layer	What it does
`wg.Go(f)`	`sync`	Adds a goroutine to a `sync.WaitGroup` and calls `f()` in it; no separate `Add`/`Done` pair needed.
`testing/synctest` stable	`testing/synctest`	Fake-time goroutine bubble graduates from experimental to stable.
`os.Root`	`os`	A handle to a directory; all path operations are constrained to it, preventing traversal attacks.

```go
// desk125.go
package main

import (
    "fmt"
    "log/slog"
    "os"
    "sync"
)

type Ticket struct {
    ID    int
    Title string
}

func process(t Ticket) error {
    slog.Info("processing", "id", t.ID, "title", t.Title)
    return nil
}

func main() {
    tickets := []Ticket{
```

```
        {1, "Replace keyboard"},
        {2, "Monitor flicker"},
        {3, "Desk lamp broken"},
        {4, "Chair squeaks"},
    }

    var wg sync.WaitGroup

    // wg.Go launches a goroutine and tracks it - no Add/Done boilerplate.
    for _, tk := range tickets {
        wg.Go(func() {
            if err := process(tk); err != nil {
                slog.Error("failed", "id", tk.ID, "err", err)
            }
        })
    }

    wg.Wait()
    fmt.Fprintln(os.Stdout, "all tickets processed")
}
```

Run:

```
go run desk125.go
```

Output:

```
time=... level=INFO msg=processing id=1 title="Replace keyboard"
time=... level=INFO msg=processing id=2 title="Monitor flicker"
time=... level=INFO msg=processing id=3 title="Desk lamp broken"
time=... level=INFO msg=processing id=4 title="Chair squeaks"
all tickets processed
```

Go 1.26

Feature	Package / layer	What it does
`errors.AsType[T]`	`errors`	Generic form of `errors.As`; returns `(T, bool)` without declaring a pointer-to-target.
`slices.Chunk`	`slices`	Yields successive sub-slices of length at most n via an iterator.

Feature	Package / layer	What it does
`url.Clone`	`net/url`	Deep-copies a `*url.URL` including its query `Values` map.
`context.AfterFunc` clarification	`context`	Behaviour when the stop function is called after cancellation is now defined by the spec.

```
// desk126.go
package main

import (
    "errors"
    "fmt"
    "slices"
)

// DeskError carries a numeric code for routing purposes.
type DeskError struct {
    Code    int
    Message string
}

func (e *DeskError) Error() string {
    return fmt.Sprintf("desk error %d: %s", e.Code, e.Message)
}

func processOrder(id int) error {
    if id <= 0 {
        return fmt.Errorf("processOrder: %w", &DeskError{Code: 400, Message: "invalid order
    }
    return nil
}

func main() {
    err := processOrder(-1)

    // errors.AsType[T] - no &target variable needed.
    if de, ok := errors.AsType[*DeskError](err); ok {
        fmt.Printf("caught DeskError code=%d msg=%s\n", de.Code, de.Message)
    }

    // slices.Chunk - batch 5 order IDs into groups of 2.
    orders := []int{101, 102, 103, 104, 105}
    fmt.Println("batches:")
```

```
    for batch := range slices.Chunk(orders, 2) {
        fmt.Println(" ", batch)
    }
}
```

Run:

```
go run desk126.go
```

Output:

```
caught DeskError code=400 msg=invalid order id
batches:
  [101 102]
  [103 104]
  [105]
```

Go 1.27

Feature	Package / layer	What it does
`encoding/json/v2`	`encoding/json/v2`	Redesigned JSON package: opt-in, cleaner API, better error messages, strict by default.
`json:",omitzero"`	`encoding/json/v2`	Omits a field when it holds its zero value (works for structs, not just pointers).
`stdlib/uuid`	`uuid`	First-party UUID generation and parsing; no third-party dependency needed.
`strings.CutLast`	`strings`	Like `strings.Cut` but finds the *last* occurrence of the separator.
`bytes.CutLast`	`bytes`	Same as `strings.CutLast` but operates on byte slices.
Promoted field literals	language	Embedded-field names may be omitted in composite literals when unambiguous.
`synctest.Sleep`	`testing/synctest`	Advances fake time inside a synctest bubble, waking goroutines blocked on timers.

```
// desk127.go
package main

import (
    "fmt"
    "strings"

    jsonv2 "encoding/json/v2"
)

// ShiftEntry records one desk shift.
// HourlyRate is omitted from JSON output when it is zero.
type ShiftEntry struct {
    StaffID    string  `json:"staff_id"`
    Day        string  `json:"day"`
    HourlyRate float64 `json:"hourly_rate,omitzero"`
}

func main() {
    shifts := []ShiftEntry{
        {"emp-001", "Monday", 22.50},
        {"emp-002", "Tuesday", 0},       // HourlyRate omitted in output.
        {"emp-003", "Wednesday", 18.75},
    }

    // encoding/json/v2: Marshal respects omitzero.
    data, err := jsonv2.Marshal(shifts)
    if err != nil {
        panic(err)
    }
    fmt.Println(string(data))

    // strings.CutLast: extract the extension from a versioned filename.
    filename := "report.2025.csv"
    before, after, found := strings.CutLast(filename, ".")
    if found {
        fmt.Printf("base=%q ext=%q\n", before, after)
    }

    // strings.CutLast on a path: find the last path segment.
    path := "/desks/floor2/station7"
    _, segment, _ := strings.CutLast(path, "/")
    fmt.Printf("last segment: %s\n", segment)
}
```

Run:

```
go run desk127.go
```

Output:

```
[{"staff_id":"emp-001","day":"Monday","hourly_rate":22.5},{"staff_id":"emp-002","day":"Tuesday"
base="report.2025" ext="csv"
last segment: station7
```

Quick index

Feature	Version
`min`, `max`, `clear` builtins	1.21
`slices`, `maps`, `cmp` packages	1.21
`log/slog`	1.21
`sync.OnceFunc` / `OnceValue`	1.21
`t.Context()`	1.21
`for i := range n`	1.22
Per-iteration loop variable capture fix	1.22
`net/http` method+path patterns, `PathValue`	1.22
`iter` package, range-over-function	1.23
`strings.SplitSeq`, `FieldsSeq`	1.23
`maps.Keys`, `maps.Values` iterators	1.23
`b.Loop()` in benchmarks	1.24
`testing/synctest` (experimental)	1.24
`weak.Pointer`, `runtime.AddCleanup`	1.24
`wg.Go()`	1.25
`testing/synctest` stable	1.25
`os.Root`	1.25
`errors.AsType[T]`	1.26
`slices.Chunk`	1.26
`url.Clone`	1.26
`encoding/json/v2`, `omitzero`	1.27
`stdlib/uuid`	1.27
`strings.CutLast`, `bytes.CutLast`	1.27
Promoted field literals	1.27
`synctest.Sleep`	1.27