Zero Series

# The World

Volume 1: A World That Ticks

Build a living world and walk into it

Early Access edition. Volumes 1–10, 105 chapters. Built 2026-09-19.
Free to read online at kryolabs.duckdns.org/zero. Buying this file supports the writing of the remaining volumes.

VOLUME 1

# A World That Ticks

Go and podman from zero, ending in a container that never stops

*A simulation server that advances on a fixed tick from a seed you chose, and a log you can prove replays byte-identical.*

VOLUME 1 · CHAPTER 1

# The Workstation

Installing Go and podman before writing a line of simulation code

## 1. Verifying the workstation

The first machine in The World is the workstation: the compiler, the container engine, and the directory where the server starts. **Never trust an installed tool until it has run something and you have read the output.**

An installer finishing without complaint proves the installer ran. It does not prove the compiler compiles or the container engine starts containers. Each tool you install today earns its place by doing its actual job once, in front of you, before anything depends on it.

The world is the reason for the strictness. It will run on a machine you control: ground with real soil chemistry, water that flows where the terrain says it must, plants that compete for light, creatures that eat the plants and each other, and a village of characters who remember what happened to them.

The client program is also yours. It lets you walk in, stand in that village, and be remembered. The world ticks while you sleep. It keeps its own history. It never resets.

None of that exists yet, and the book does not hand it over whole. You build the server that owns all of it, the simulation that advances it, the wire protocol that carries it, and the renderer that shows it.

The language is Go. The runtime home is a container. By the last page of this volume, the first server lives in a box that can keep counting time whether or not you are logged in.

Create the `theworld` module and its `worldd` command below. Run the formatter, vet and tests before adding simulation behavior.

## 2. Go and the first program

⚙ TOOL — INSTALLING GO 1.26

Download the archive for your operating system from `go.dev/dl` and follow the three-line install on that page (on Linux: unpack to `/usr/local`, add `/usr/local/go/bin` to your `PATH`, open a new terminal). This volume targets Go 1.26; the reference output below records Go 1.26.3 on Linux/amd64.

A newer compatible compiler can build the examples, but version strings and diagnostics can differ. Exact transcript replay needs the reference toolchain, not merely a compatible language version. Go's install page is the reference: `go.dev/doc/install`.

▣ BUILD · STAGE 1 — MAKE THE COMPILER INTRODUCE ITSELF

```
go version
```

```
$ go version

go version go1.26.3 linux/amd64
```

Your line will differ in the last digit and probably in the tail. The patch number climbs over time. `linux/amd64` names this machine's operating system and processor, so a Mac prints `darwin/arm64` and Windows prints `windows/amd64`.

Use Go 1.26 or a compatible newer release for learning. Keep 1.26.3 when checking this quoted version line.

If the command is not found at all, the `PATH` step above did not take. Open a fresh terminal before debugging anything else, because the old one still has the old `PATH`.

The version line proves the toolchain exists. Now make it do its job. Create a scratch directory anywhere (it will not be part of the project; `~/scratch` is fine) and put one file in it.

▣ BUILD · STAGE 2 — THE FIRST PROGRAM

```
// hello.go
package main

import "fmt"

func main() {
        fmt.Println("the world is not running yet")
}
```

```
$ go run hello.go

the world is not running yet
```

Seven lines, and five of them carry the program. `package main` declares that this file belongs to a program instead of a library. Go programs start in a package with exactly that name.

`import "fmt"` pulls in the standard library's formatting package, the one that knows how to print. `func main()` is the entry point: when the program starts, this function runs, and when it returns, the program ends.

`go run` did two jobs in one command: it compiled the file into a real machine-code binary in a temporary location, then executed it. There is no interpreter here. Every Go program in this book is compiled first, every time, and the compile is fast enough that you rarely see it happening.

◆ NOTE — THE EDITOR

Any text editor works. If you want one with Go support wired in, VS Code plus its Go extension gives you error underlines and formatting on save, free.

The book never depends on editor features. Everything is checked from the command line, so what your editor thinks is never the authority on whether the code is right.

## 3. The module and worldd

A scratch file proves the compiler works, but a project of this size needs a structure the tools recognize. Go's unit of project is the *module*: a directory tree with a `go.mod` file at its root.

That file names the module and records which Go version and which outside libraries it depends on. Everything you build across all twenty-five volumes lives in one module. Create its home and initialize it:

▣ BUILD · STAGE 3 — INITIALIZE THE THEWORLD MODULE

```
mkdir theworld
cd theworld
go mod init theworld
go mod edit -go=1.26
```

```
$ go mod init theworld

go: creating new go.mod: module theworld
go: to add module requirements and sums:
        go mod tidy
```

```
$ cat go.mod

module theworld

go 1.26
```

> The second suggested command, `go mod tidy`, matters later, when the module gains dependencies. Today there are none to tidy.
>
> The `go mod edit` line does one small thing, and it matters. `go mod init` stamps the file with the exact toolchain that ran it, `1.26.3` on this machine and some other patch on yours.
>
> The `go` directive declares a minimum Go version and selects the language version. It can include a patch-level minimum, but it never locks the compiler to that exact patch.
>
> Setting it to `go 1.26` permits Go 1.26 or newer, and it means your `go.mod` now matches the one every later chapter quotes, byte for byte. The file is two facts: this module is named `theworld`, and its code is Go 1.26.
>
> The name carries real weight. It becomes the root of every import path in the project, and that matters as soon as packages begin to refer to one another.

Inside the module, layout is convention. Go projects put each runnable program in its own directory under `cmd/`, named after the binary it produces.

This one is `cmd/worldd`: the world daemon, the server that will own every rock, river, creature, and villager. The trailing `d` is an old Unix habit marking a program built to run continuously in the background. By the end of this volume it has work to justify the letter.

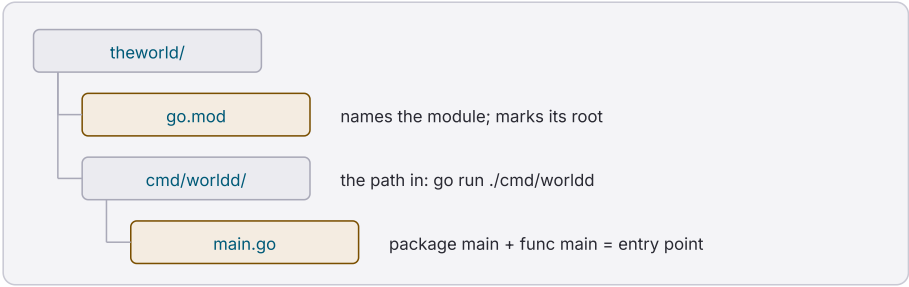



**Figure 1.1** — The whole skeleton. `go.mod` tells the toolchain where the module starts and what it is called; `./cmd/worldd` is how you name the package you want run; `main.go` is where execution begins. Every command in this book that starts with `go` is run from the directory holding `go.mod`.

**▣ BUILD · STAGE 4 — WORLDD DRAWS ITS FIRST BREATH**

```
// cmd/worldd/main.go
package main

import (
        "fmt"
        "runtime"
)

const version = "0.0.1"

func main() {
        fmt.Println("worldd", version, "starting")
        fmt.Println("go runtime:", runtime.Version())
        fmt.Println("nothing exists yet: no ground, no water, no
time")
}
```

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

```
$ go run ./cmd/worldd

worldd 0.0.1 starting
go runtime: go1.26.3
nothing exists yet: no ground, no water, no time
```

Two new pieces of Go appeared. The `import` block now lists two packages in parentheses, one per line. `const version = "0.0.1"` declares a named constant: a value fixed at compile time, so nothing can change it while the program runs.

The second output line comes from `runtime.Version()`, the standard library reporting which Go built the running binary. Your line shows your own patch release.

The command that printed nothing matters too. `go vet ./...` checks every package in the module (`./...` means "here and everything below") for mistakes the compiler technically permits. Silence is a pass. Run it before every run in this book.

**⚠ WORKED FAILURE — THE IMPORT THAT REFUSED TO RIDE ALONG**

While writing stage 4 you might reasonably think ahead: the server will surely need `os`, the package for talking to the operating system, so import it now and save a trip. Add `"os"` to the import block without using it, and:

```
$ go run ./cmd/worldd

# theworld/cmd/worldd
cmd/worldd/main.go:5:2: "os" imported and not used
```

The program did not run. This is a compile error, not a warning: Go refuses to build a file that imports a package and never touches it.

Read the message the way you will read hundreds like it. The # line names the package that failed to compile, and the next line gives file, line 5, column 2: the exact position of the offending import.

The reasoning from symptom to cause is one step here. The policy behind it is the lesson: in a codebase that will grow for twenty-five volumes, an import is a claim that a dependency is

real, and Go keeps every claim honest by force. Delete the line (or never add it) and the build comes back. Import when you use, not when you predict.

## 4. Podman containers

Podman runs the world's supporting services and reference checks in containers. Each container has a separate user-space view, but shares the host kernel on Linux or a helper VM's kernel on Windows and macOS.

Rootless Podman avoids a privileged daemon. Processor architecture still matters for replay.

**⚙ TOOL — INSTALLING PODMAN**

On most Linux distributions it is one package: `sudo apt install podman`, `sudo dnf install podman`, or your distribution's equivalent. On macOS and Windows, install Podman Desktop and run `podman machine init` then `podman machine start` once, which creates the small Linux virtual machine containers need there. Full instructions: `podman.io/docs/installation`.

**▣ BUILD · STAGE 5 — A CONTAINER EARNS ITS KEEP**

```
podman --version
podman run --rm docker.io/library/alpine:3.22 echo "a container said this"
```

```
$ podman --version

podman version 5.7.0
```

```
$ podman run --rm docker.io/library/alpine:3.22 echo "a
container said this"

a container said this
```

On the first run, Podman may print image-download progress. `run` starts the container. `--rm` removes it after exit. The arguments after the image name select its command.

A later run can reuse the cached image. That speedup is useful, but the cached image is also one more reason exact replay has to record what image was used.

## 5. Replay conditions

A replay claim names its conditions: the same source and inputs, deterministic draw and update order, an exact compiler and dependencies, and a reference platform. A seed alone does not fix all of those.

Later simulation code uses floating-point arithmetic. Its final bits can depend on the architecture and compiler even when every operation appears in a fixed order. The book's byte comparisons test the recorded reference environment; they do not prove identical results on any machine.

Today's `go.mod` records a compatibility requirement. Likewise, `alpine:3.22` and `golang:1.26` are mutable tags: a publisher can move either to new content while your local cache still holds the old image.

Exact replay requires an image reference ending in `@sha256:…` and an explicit platform such as `--platform linux/amd64`, plus the recorded Go version. Inspect a pulled image with `podman image inspect` and record its `RepoDigests` and platform before using the digest reference.

The early volume scripts retain tags. They do not record the original image digest, so resolving a tag today cannot recover that missing provenance. The contract compatibility note records this limit and the digest-pinned references the later volumes actually use.

## 6. Checkpoint

✓ CHECKPOINT — WHAT YOU CAN NOW DO

- Install a Go toolchain and prove it works with two commands: one that prints its version and one that compiles and runs a real program.
- Explain what `package main`, `import`, and `func main()` each do in the seven-line program from this chapter.
- Create a Go module, read its `go.mod`, and say why the module name is the root of every import path to come.
- Run `go vet ./...` and interpret both of its answers: silence, and a file-line-column report.
- Start a container from a version-tagged image, explain each part of the `podman run` command, and say where the image came from and why the second run was faster.
- Explain why a tag is weaker than a digest.
- Handed the error `"os" imported and not used`, name the policy behind it and fix it in one line.

⚡ EXERCISES — TRY FIRST, THEN REVEAL

▼ **Exercise 1 — a second binary.** The module can hold a second program: `worldc`, the client path. Create `cmd/worldc/main.go` with a `main` that prints `worldc: no world to connect to yet`, then run it. What did you have to change compared to stage 4, and what does `go vet ./...` now cover?

Almost nothing changes: same `package main`, same entry point, different directory, and `go run ./cmd/worldc` picks it by path. Both programs can be package `main` because packages are scoped by directory, one package per directory. `go vet ./...` now checks both binaries in one command; the `./...` pattern grows with the module for free, so the book standardizes on it.

▼ **Exercise 2 — break it on purpose, twice. From your home directory (not the module), run `go run ./cmd/worldd` and read the error. Then, back inside the module, change `func main()` to `func Main()` and run it again. Which of Go's rules did each error enforce?**

The first prints `go: go.mod file not found in current directory or any parent directory`: package paths like `./cmd/worldd` are resolved relative to a module, and Go searches upward from where you stand until it finds a `go.mod`. The second fails because the entry point must be exactly `main`, lowercase; `Main` is just an ordinary function nobody calls, so the linker reports that `main` is undeclared. Both errors name their rule in the message; Go usually does.

▼ **Exercise 3 — Go inside a container. Without installing anything new, run `podman run --rm docker.io/library/golang:1.26 go version`. Predict the output before you run it, including how it can differ from your own `go version`.**

After a pull (this image is far larger than Alpine; give it a minute) it prints a `go version go1.26.x linux/…` line. The patch number can differ from your host's because the image carries its own toolchain, and a later pull of the same tag can update it. The platform reads `linux` even on a Mac or Windows machine, because the container runs Linux; the

architecture follows the selected image platform. Record both before comparing reference output.

VOLUME 1 · CHAPTER 2

# A Place on the Grid

Structs and methods for the coordinate a cell lives at

## 1. The Coord rule

`worldd` announces itself and exits because it has no way to say where anything is. The first server rule is small and permanent: **a place is one value: the column and the row travel together, or not at all.**

The world is a grid: columns running east, rows running south, and every future inhabitant of The Hollow standing on exactly one square of it. A place on that grid is a column number and a row number.

Keeping the numbers loose makes every grid function grow a pair of parameters: `x int, y int`. A function that moves something takes four integers, two for where it is and two for where it goes.

Nothing stops you from passing the row where the column belongs, and the compiler cannot object, because an `int` is an `int`. The call `move(7, 12, 12, 7)` compiles as happily backwards as forwards.

In a simulation that runs for days unattended, a swapped pair does not crash anything. It quietly puts a creature on the wrong square and lets the world carry on being subtly wrong.

Go's tool for binding several values into one is the *struct*: a type you define by listing named fields, each with its own type. Once a place is a single value, a function that moves something takes two arguments, a from and a to, and passing them backwards is a mistake you can see.

◆ NOTE — THE TYPES THE FIELDS ARE MADE OF

Struct fields are built from Go's basic types, and this book leans on four of them. `int` is a whole number, positive or negative; grid columns and rows are `int`s. `float64` is a number with a fractional part, like a height of 2.5 meters.

`bool` is `true` or `false`. `string` is text.

Every type also has a *zero value*, the value a variable holds before you assign anything: `0` for numbers, `false` for `bool`, `""` for strings. Go never leaves a variable undefined, a guarantee that matters in a program meant to run unattended.

The type also needs a home. The layout from last chapter reserved `internal/sim` for the simulation itself, and this is the moment it earns its keep.

`cmd/worldd` is the command: it starts things and prints things. The world's own vocabulary, starting today with what a place is, belongs to the `sim` package, where the command can use it but does not own it.

## 2. The Coord struct

▣ BUILD · STAGE 1 — DEFINE COORD AND HOLD ONE IN YOUR HAND

```
// internal/sim/coord.go
// Package sim holds the simulation's own types. Nothing in here
// prints, listens, or knows that a command called worldd
exists.
package sim

// Coord is a place on the world grid: column X, row Y.
type Coord struct {
        X int
        Y int
}
```

```
// cmd/worldd/main.go
package main

import (
        "fmt"

        "theworld/internal/sim"
)

func main() {
        var origin sim.Coord
        spring := sim.Coord{X: 12, Y: 7}
        fmt.Println("origin:", origin)
        fmt.Println("spring:", spring)
        fmt.Println("spring column:", spring.X)
}
```

```
$ go run ./cmd/worldd

origin: {0 0}
spring: {12 7}
spring column: 12
```

`type Coord struct { ... }` teaches Go a new noun. From that line on, `Coord` is as real to the compiler as `int`: you can declare variables of it, pass it, return it.

The import path `theworld/internal/sim` is the module name from last chapter plus the directory. Inside `main`, the package's names arrive with a `sim.` prefix.

Capitalized names like `Coord` and `X` are the ones a package makes visible outside itself.

`var origin sim.Coord` got the zero value with no assignment anywhere. A struct's zero value is every field at its own zero, so an unset place is `{0 0}`, the top-left corner, never garbage.

The *struct literal* `sim.Coord{X: 12, Y: 7}` builds a value with the fields named, so the reader of the call site knows which 12 is which. The dot in `spring.X` reaches into the value for one field.

A `Coord` that only sits there is still a pair of boxes. The server will constantly step from one place to a neighbor and measure how far apart two places are.

Those questions should live *on the type*, so that every part of the program asks the same code the same question. That is what a method is: a function with a home.

▣ BUILD · STAGE 2 — METHODS: BEHAVIOR BESIDE THE DATA

```
// internal/sim/coord.go — add below the type

// Offset returns the coordinate dx columns and dy rows away.
func (c Coord) Offset(dx, dy int) Coord {
        return Coord{X: c.X + dx, Y: c.Y + dy}
}

// Dist returns the number of grid steps between two
coordinates,
// counting only moves along a row or a column.
func (c Coord) Dist(o Coord) int {
        dx := c.X - o.X
        if dx < 0 {
                dx = -dx
        }
        dy := c.Y - o.Y
        if dy < 0 {
                dy = -dy
        }
        return dx + dy
}


// cmd/worldd/main.go — the new body of main
        var origin sim.Coord
        spring := sim.Coord{X: 12, Y: 7}
        east := spring.Offset(1, 0)
        fmt.Println("east of spring:", east)
        fmt.Println("origin to spring:", origin.Dist(spring),
"steps")
        fmt.Println("spring to origin:", spring.Dist(origin),
"steps")
```

```
$ go run ./cmd/worldd

east of spring: {13 7}
```

```
origin to spring: 19 steps
spring to origin: 19 steps
```

The part in parentheses before the name, `(c Coord)`, is the *receiver*. It declares that `Offset` belongs to `Coord`, and inside the method `c` is the coordinate the call happened on.

You call it with the same dot that reads a field. `spring.Offset(1, 0)` asks the spring for the square one column east.

`Dist` counts steps the way something walking the grid would, no diagonals: from the origin to the spring is 12 columns plus 7 rows, 19 steps. The two `if` blocks flip negative differences positive, so the answer comes out the same from either end, as the run confirms.

`Offset` never touches `c`. It builds and returns a new `Coord`, and `spring` is still `{12 7}` afterwards. That choice turns into this chapter's worked failure.

## 3. The Cell struct

A coordinate names a square. A `Cell` is what the square holds.

For now, before terrain proper arrives, a cell needs three things: where it is, how high its ground sits, and how deep the water standing on it runs. One of those fields is a struct, and Go does not blink: a field's type can be any type, including one you defined a page ago.

**▣ BUILD · STAGE 3 — A STRUCT INSIDE A STRUCT, AND TWO QUESTIONS IT CAN ANSWER**

```
// internal/sim/cell.go
package sim

// Cell is one square of ground: where it sits, how high the
ground
// is, and how deep the water standing on it runs. Heights and
```

```
depths
// are in meters.
type Cell struct {
        At        Coord
        Elevation float64
        Water     float64
}

// Wet reports whether any water stands on the cell.
func (c Cell) Wet() bool {
        return c.Water > 0
}

// Surface returns the height of the cell's top: ground plus
water.
func (c Cell) Surface() float64 {
        return c.Elevation + c.Water
}


// cmd/worldd/main.go — the new body of main
        spring := sim.Coord{X: 12, Y: 7}
        pool := sim.Cell{At: spring, Elevation: 1.0, Water: 0.4}
        bank := sim.Cell{At: spring.Offset(1, 0), Elevation:
2.5}
        fmt.Println("pool:", pool.At, "wet:", pool.Wet(),
"surface:", pool.Surface())
        fmt.Println("bank:", bank.At, "wet:", bank.Wet(),
"surface:", bank.Surface())
```

```
$ go run ./cmd/worldd

pool: {12 7} wet: true surface: 1.4
bank: {13 7} wet: false surface: 2.5
```

The `bank` literal names only two of the three fields, and `Water` quietly takes its zero: dry, exactly as a field left unsaid should be.

Both methods answer questions any part of the server might ask, and both answer them in one place. If the definition of wet ever grows a threshold, the code that changes is one line in `sim`, not a comparison copied through the codebase.

That is the case for methods in one sentence: the type carries its own vocabulary, so the rest of the program asks instead of assuming.

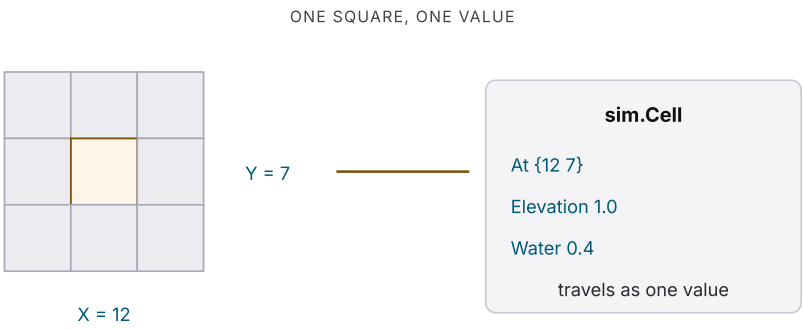



**Figure 2.1** — the pool at column 12, row 7, and the single struct value that carries everything the server knows about that square.

## 4. Pointer receivers

`Wet` and `Surface` read a cell. A server also needs to change one: rain falls, and the water on a square deepens.

Following the pattern of every method so far gives an obvious next method. It is wrong in a way no error message reports.

⚠ WORKED FAILURE — THE FLOOD THAT EVAPORATED ON RETURN

```
// internal/sim/cell.go — the obvious attempt
// Flood adds depth meters of standing water to the cell.
func (c Cell) Flood(depth float64) {
        c.Water += depth
}


// cmd/worldd/main.go — the new body of main
        spring := sim.Coord{X: 12, Y: 7}
        bank := sim.Cell{At: spring.Offset(1, 0), Elevation:
2.5}
        fmt.Println("before:", "wet:", bank.Wet(), "water:",
bank.Water)
        bank.Flood(0.3)
        fmt.Println("after: ", "wet:", bank.Wet(), "water:",
bank.Water)
```

```
$ go run ./cmd/worldd

before: wet: false water: 0
after:  wet: false water: 0
```

It compiles. `go vet ./...` passes. The line `c.Water += depth` runs, and adds 0.3 to something.

Reason from the symptom: the addition definitely happened, so there must have been a *second* `Cell` for it to happen to. There was.

Go passes copies. Calling a function with a struct hands the function a duplicate of it, and a receiver written `(c Cell)` is a parameter like any other.

`c` was a copy of `bank`, the flood soaked the copy, and the copy was thrown away when the method returned.

Stage 2's `Offset` had the same receiver and never noticed, because it never tried to change `c`. It returned a new value instead. A reading method on a copy is correct. A writing method on a copy is a no-op with good intentions.

**▣ BUILD · STAGE 4 — A POINTER RECEIVER, SO THE CHANGE LANDS**

```
// internal/sim/cell.go — the fix: *Cell, not Cell
// Flood adds depth meters of standing water to the cell.
func (c *Cell) Flood(depth float64) {
        c.Water += depth
}
```

```
$ go run ./cmd/worldd

before: wet: false water: 0
after:  wet: true water: 0.3
```

One character changed. `*Cell` means the method receives not a copy of the cell but its *address*: a pointer, a value that says where the original lives in memory.

> Through the pointer, `c.Water += depth` reaches the actual `bank`, and the run shows the water arriving and staying.
>
> The call site did not change at all. `bank.Flood(0.3)` still reads the same, because Go sees a pointer receiver and passes the address for you.
>
> The receiver declaration is where the choice lives, and it is a one-word answer to one question: does this method change the thing it belongs to? No: value receiver. Yes: pointer receiver.

## 5. Why copies are the default

Everything in this chapter rests on one property of Go: assignment and argument passing copy the value, whatever the value is. An `int` copies, and so does a struct of two `int`s, and so does a struct holding another struct.

Copies make code easy to reason about. No function can change your variables behind your back unless you handed it their address on purpose.

The cost is the trap you walked out of: a method that wants to mutate must say so, with a pointer receiver. The `*` in the declaration makes that statement in public. When you read `func (c *Cell) Flood` six volumes from now, it will still tell you at a glance that flooding changes the cell.

The same property explains why `Coord` can stay copy-friendly forever. A coordinate is a fact, not a thing with a life: moving something should produce the new place, not edit the old one.

`Offset` returning a fresh value means the spring's coordinates cannot be corrupted by whoever borrowed them.

Small immutable facts get value receivers. Stateful residents of the world, of which `Cell` is the first and creatures will be louder, get

pointer receivers on anything that changes them. The whole server stays readable through that split.

That is what behavior beside data buys in practice. The two types you wrote are not passive records with helper functions scattered around them; they are the beginnings of a vocabulary.

Code elsewhere in the server will say `cell.Wet()` and `place.Dist(target)`, sentences with the noun and the verb attached. The definitions of those verbs sit in one file, next to the fields they read, where a change is one edit.

## 6. Checkpoint

✓ CHECKPOINT — WHAT YOU CAN NOW DO

- Define a struct type, build one with a named-field literal, and predict its zero value field by field, including `{{0 0} 0 0}` for a struct inside a struct.
- Explain what the two loose `int`s design costs, and what binding them into `Coord` makes impossible.
- Write a method, name its receiver, and say precisely what `c` is inside `Offset` when `spring.Offset(1, 0)` runs.
- Compute `Dist` by hand for any two coordinates: 12 columns plus 7 rows made the run print 19 steps.
- Handed a method that compiles, runs, and changes nothing, check the receiver before anything else, and narrate where the lost mutation went.
- Choose between `(c Cell)` and `(c *Cell)` for a new method and defend the choice in one sentence.

⚡ EXERCISES — TRY FIRST, THEN REVEAL

▼ **Exercise 1 — a coordinate that knows its limits. Give `Coord` a method `In(w, h int) bool` that reports whether**

**the coordinate lies on a grid `w` columns wide and `h` rows tall. Print `spring.In(64, 64)` and `sim.Coord{X: 64, Y: 7}.In(64, 64)`, and decide before running which is `false`.**

Four comparisons joined with `&&`: `c.X ⩾ 0 && c.X < w && c.Y ⩾ 0 && c.Y < h`. The prints come out `true` then `false`: on a 64-wide grid the columns run 0 through 63, so `X: 64` is the first column that does not exist. Off-by-one boundaries like this are exactly where a method earns its keep, because the `<`-versus-`⩽` decision gets made once.

▼ **Exercise 2 — drain, but never below dry. Write `Drain(depth float64)` on `Cell`: it removes up to `depth` meters of standing water and refuses to go negative. Start a cell at `Water: 0.4`, drain 1.0, and print the result. Which receiver does it need?**

Pointer receiver, because draining changes the cell: subtract, then clamp with `if c.Water < 0 { c.Water = 0 }`. The print reads `pool water: 0`, not `-0.6`. Write it with `(c Cell)` first if you want the failure box again on your own terms: the print stays 0.4, and this time you can name the missing character.

▼ **Exercise 3 — predict the zero world. Declare `var blank sim.Cell` and write down, before running, exactly what `fmt.Println(blank)` prints, field by field. Then run it.**

`{{0 0} 0 0}`: the inner braces are the zero `Coord` inside `At`, then `Elevation` and `Water` at 0. A blank cell is a dry square at the origin on the valley floor: every field defined, none of them meaningful yet. Distinguishing "zero" from "never set" is a real problem this server will meet again, and the first defense is knowing the zero values cold.

VOLUME 1 · CHAPTER 3

# Rock, Water, Soil

A terrain grid of rock, water and soil in one typed slice

## 1. The y·W + x slice

You have a coordinate and a cell, and nowhere to stand. The terrain rule is the first storage rule of the world: **terrain is a typed constant, and the ground is one flat slice of them, addressed by `y·W + x`.**

Everything asks the ground what sits underneath it. Water settles into low soil, plants root where rock does not stop them, animals drink where land meets water, and a village rises where all three meet.

That question has to be cheap, small, and hard to misread. A terrain kind is not a string and not a bare number; it is a named constant of its own type, so the compiler refuses nonsense before the world runs.

In memory, the grid of those kinds is a single flat slice. A little arithmetic turns any coordinate into a position in it.

`worldd` prints a real map by the end of the chapter: a rock rim, a soil valley, a pond. You place every one of those cells by hand, and that is deliberate.

Randomness is not needed to prove the container the terrain lives in. Hand-placing the cells means every number on the screen is one you can predict before the run.

## 2. The Terrain type

The obvious first move is a string per cell: `"rock"`, `"water"`, `"soil"`. It works, and it is wrong twice over.

A string costs sixteen bytes of header plus the text itself, for information that has three possible values. Worse, `"watre"` compiles cleanly and fails at some distant runtime moment, in whatever code first compares against it.

Plain integers with a comment saying `0=rock 1=water 2=soil` fix the size and keep the danger. Any `int` in the program can wander into a terrain's slot, including a count or a coordinate.

Go's answer is a defined type with named constants. Add this to `internal/sim`, next to the coordinate type from last chapter.

▣ BUILD · STAGE 1 — A TYPE WHOSE VALUES HAVE NAMES

```
// internal/sim/terrain.go
package sim

// Terrain is what the ground is at one cell.
type Terrain uint8

const (
        Rock Terrain = iota // the zero value: a new world is
solid rock
        Water
        Soil
)

// String makes a Terrain print as a word instead of a number.
func (t Terrain) String() string {
        switch t {
        case Rock:
                return "rock"
        case Water:
                return "water"
        case Soil:
                return "soil"
        }
        return "unknown"
}
```

```
// Glyph is the one-byte map symbol for a terrain kind.
func (t Terrain) Glyph() byte {
        switch t {
        case Water:
                return '~'
        case Soil:
                return '.'
        }
        return '#'
}


// cmd/worldd/main.go — a throwaway main to poke the new type
package main

import (
        "fmt"

        "theworld/internal/sim"
)

func main() {
        fmt.Println(sim.Rock, sim.Water, sim.Soil)
        fmt.Println(uint8(sim.Rock), uint8(sim.Water),
uint8(sim.Soil))
        var t sim.Terrain
        fmt.Println("a fresh Terrain is", t)
}
```

```
$ go run ./cmd/worldd

rock water soil
0 1 2
a fresh Terrain is rock
```

`type Terrain uint8` declares a new type whose underlying storage is one unsigned byte. It is not an alias: a `uint8` holding a page count will not pass where a `Terrain` is expected unless you convert it on purpose.

The compiler now polices the difference between ground and arithmetic.

`iota` is Go's counter for constant blocks: it starts at 0 on the first line and climbs by one per line. `Rock`, `Water`, and `Soil` receive 0, 1, 2 without anyone typing the numbers, and adding a kind later is one new line, not a renumbering.

The `String` method is a quiet contract with the standard library: `fmt` checks whether a value knows how to describe itself and calls this method if so. The first printed line says `rock water soil`, while the converted second line shows the bytes underneath.

The third line is a design decision wearing a demo's clothes. Go initializes every variable; a `Terrain` nobody has assigned is 0, and 0 is `Rock` because `Rock` came first in the block.

That ordering is chosen. A freshly allocated world is solid bedrock, which is geologically sensible and obvious at a glance. A bug that forgets to assign terrain shows up as rock where you expected anything else, not as a plausible-looking valley.

◆ NOTE — WHY ONE BYTE MATTERS AT WORLD SCALE

This chapter's grid is 12 by 8: 96 cells, 96 bytes, nothing. The type is being chosen for the world this book builds.

At 1,024 by 1,024 cells, one byte per cell is a single megabyte. Strings at that scale cost tens of megabytes and scatter the heap with tiny allocations. Picking the small representation now costs nothing and never has to be unpicked.

## 3. The flat slice

Now the storage. This is your first Go slice, and every later system reads this one; several rewrite it live.

A slice is a small header: a pointer to a run of elements, plus a length and a capacity. `make([]Terrain, 96)` builds one in a single step: allocate 96 contiguous bytes, zero them all, and return a header pointing at the run.

Zero terrain is `Rock`, so one call makes one block of memory with every cell already holding a legal value.

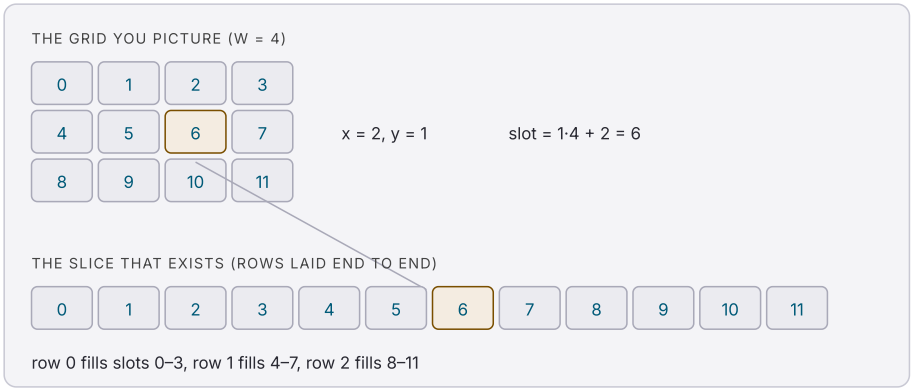
A slice is one-dimensional, and the world is not. Arithmetic is the bridge between them, and it is easier to see with numbers than with symbols.

Lay a 12-wide grid row after row into a slice: row 0 occupies slots 0 through 11, row 1 occupies 12 through 23, row 2 starts at 24. The cell at `x=3, y=2` sits three slots into that third row: 24 + 3 = 27. Skip `y` full rows, then walk `x` slots into the current one.

**∑ MATH INTERLUDE — THE FLATTENING FORMULA**

For a grid `W` cells wide, the cell at column `x`, row `y` lives at slot `i = y·W + x`. Check it against the numbers above: y=2, W=12, x=3 gives 2·12 + 3 = 27. The formula runs backwards too, using integer division and remainder: slot 27 in a 12-wide grid is row 27 ÷ 12 = 2 (integer division drops the fraction) and column 27 mod 12 = 3. Two multiplications a lookup, and no table anywhere.

**W** the grid's width: how many cells make one row
**x** column, counted from 0 at the left edge
**y** row, counted from 0 at the top edge
**i** the slot in the flat slice: y·W + x
**mod** the remainder after integer division: 27 mod 12 = 3, written 27 % 12 in Go




**▣ BUILD · STAGE 2 — THE GRID, WITH THE FORMULA IN ONE PRIVATE PLACE**

```
// internal/sim/terrain.go — continued

// Grid is the ground itself: W by H cells of terrain in one
flat slice.
type Grid struct {
        W, H  int
        cells []Terrain
}

// NewGrid allocates a w-by-h grid in a single allocation.
// Every cell starts as the zero value, Rock.
func NewGrid(w, h int) *Grid {
        return &Grid{W: w, H: h, cells: make([]Terrain, w*h)}
}

// index turns a 2D coordinate into a position in the flat
slice.
func (g *Grid) index(c Coord) int {
        return c.Y*g.W + c.X
}

// In reports whether a coordinate is on the grid at all.
func (g *Grid) In(c Coord) bool {
        return c.X ⩾ 0 && c.X < g.W && c.Y ⩾ 0 && c.Y < g.H
}

// At returns the terrain at a coordinate.
func (g *Grid) At(c Coord) Terrain {
        return g.cells[g.index(c)]
}

// Set overwrites the terrain at a coordinate.
func (g *Grid) Set(c Coord, t Terrain) {
```

```
        g.cells[g.index(c)] = t
}


// cmd/worldd/main.go — replace the throwaway main
func main() {
        g := sim.NewGrid(12, 8)
        c := sim.Coord{X: 3, Y: 2}
        fmt.Println("slot for (3,2):", c.Y*g.W+c.X)
        fmt.Println("before:", g.At(c))
        g.Set(c, sim.Water)
        fmt.Println("after: ", g.At(c))
        fmt.Println("neighbor untouched:", g.At(sim.Coord{X: 4,
Y: 2}))
}
```

```
$ go run ./cmd/worldd

slot for (3,2): 27
before: rock
after:  water
neighbor untouched: rock
```

`cells` and `index` start with small letters, so nothing outside `internal/sim` can touch them. The rest of the program goes through `At` and `Set` or it does not get in.

That is last chapter's method discipline paying its first real rent: the flattening formula exists in exactly one function, three lines long. If the layout ever changes, one function changes with it, and no caller anywhere is holding a copy of the arithmetic.

The run confirms the formula against the interlude, 27 on the nose, and confirms that writing one cell disturbed nothing beside it.

## 4. Rendering the map

`Render` walks the grid row by row and emits one glyph per cell: `#` for rock, `~` for water, `.` for soil.

Then `main` lays soil across the interior, leaves the untouched rim as rock, and places fourteen water cells one by one to make a pond.

**▣ BUILD · STAGE 3 — RENDER IT, AND DRAW THE FIRST MAP**

```
// internal/sim/terrain.go — add the import and the renderer
import "strings"

// Render draws the whole grid as one glyph per cell, one row
per line.
func (g *Grid) Render() string {
        var b strings.Builder
        for y := 0; y < g.H; y++ {
                for x := 0; x < g.W; x++ {
                        b.WriteByte(g.At(Coord{X: x, Y:
y}).Glyph())
                }
                b.WriteByte('\n')
        }
        return b.String()
}


// cmd/worldd/main.go — the whole file
package main

import (
        "fmt"

        "theworld/internal/sim"
)

const version = "0.0.1"

func main() {
        fmt.Println("worldd", version, "starting")

        g := sim.NewGrid(12, 8)

        // Soil everywhere except a one-cell rock rim.
        for y := 1; y < g.H-1; y++ {
                for x := 1; x < g.W-1; x++ {
                        g.Set(sim.Coord{X: x, Y: y}, sim.Soil)
                }
        }

        // A pond, placed by hand. Real generation needs
randomness
        // the sim does not have yet.
        pond := []sim.Coord{
                {X: 4, Y: 2}, {X: 5, Y: 2}, {X: 6, Y: 2}, {X: 7,
```

```
Y: 2},
                {X: 3, Y: 3}, {X: 4, Y: 3}, {X: 5, Y: 3}, {X: 6,
Y: 3},
                {X: 7, Y: 3}, {X: 8, Y: 3},
                {X: 4, Y: 4}, {X: 5, Y: 4}, {X: 6, Y: 4}, {X: 7,
Y: 4},
        }
        for _, c := range pond {
                g.Set(c, sim.Water)
        }

        fmt.Print(g.Render())
        fmt.Println("ground:", g.W*g.H, "cells,", len(pond), "of
them water")
}
```

```
$ go run ./cmd/worldd

worldd 0.0.1 starting
############
#..........#
#...~~~~...#
#..~~~~~~..#
#...~~~~...#
#..........#
#..........#
############
ground: 96 cells, 14 of them water
```

Read it against the code. The top and bottom rows and both edge columns were never assigned, so they render as rock: the zero value drew the valley walls for free.

The pond's middle row runs from x=3 to x=8, one cell wider than the rows above and below it, and the map shows exactly that bulge.

Nothing here is generated, so nothing here can surprise you. That property becomes the baseline every run is judged against.

`strings.Builder` deserves its one sentence: appending to a string with `+` copies the whole string every time, while a builder accumulates bytes in a growing buffer and produces the final string once, at the end.

> For 96 cells it is a habit. For the grids this world grows into, it is the difference between a render and a stall.

There is also a slice hiding in plain sight in `main`: `pond` is a `[]sim.Coord` built from a literal. `for _, c := range pond` visits each element in order, with the underscore discarding the slot number that `range` also offers.

Slices of structs, iterated and consumed, become one of this book's steady patterns.

## 5. Why the grid is flat

The tempting representation was `[][]Terrain`, a slice of row slices, which lets you write `grid[y][x]` and feel done. The flat slice earns its keep in memory, not in syntax.

A slice of slices for a 12 by 8 grid is nine separate allocations, one for the outer spine and one per row. The rows land wherever the allocator has room; every `grid[y][x]` is two hops, first to the spine, then through a pointer to wherever that row lives.

The flat slice is one allocation. A lookup is one multiply, one add, one read.

The deeper reason is how the machine reads memory. A CPU never fetches one byte; it pulls 64-byte cache lines, so touching any cell drags its 63 nearest slice-neighbors into fast memory with it.

In the flat layout those neighbors are real terrain, most of the current row and a piece of the next. A row-by-row pass like `Render` finds nearly everything it wants already loaded.

Scattered rows squander that. The hardware keeps prefetching, but what arrives is whatever happens to sit next to each stray row. When a tick means sweeping the whole grid, the flat layout means the

sweep runs at the speed memory streams instead of the speed pointers chase.

The formula also refuses to let the grid go ragged. A slice of slices can hold rows of different lengths, and nothing but discipline keeps a bug from appending to one row and not another.

In a flat slice, the dimensions live in `W` and `H`, and every cell exists precisely once. What the formula does demand is respect for the order of its two terms, and that is where this chapter's mistake lives.

⚠ WORKED FAILURE — TWO MULTIPLICATIONS, ONE OF THEM WRONG

The formula is easy to invert without noticing. Suppose `index` had come out of your fingers as columns-first:

```
func (g *Grid) index(c Coord) int {
        return c.X*g.W + c.Y // BUG: skips c.X rows' worth of
cells per column
}
```

```
$ go run ./cmd/worldd

worldd 0.0.1 starting
panic: runtime error: index out of range [97] with length 96

goroutine 1 [running]:
theworld/internal/sim.(*Grid).Set(...)
        /home/you/theworld/internal/sim/terrain.go:67
main.main()
        /home/you/theworld/cmd/worldd/main.go:19 +0x40a
exit status 2
```

Reason from the numbers, because they name the culprit. The slice holds 96 cells, so legal slots run 0 through 95, and something asked for 97.

`x·12 + y = 97` means x=8, y=1: a legal cell, eight columns in, one row down, that the bad arithmetic mapped past the end of the slice.

Go's bounds check caught the first out-of-range write and stopped the program with the address of the crime. The stack trace (your path will be your checkout's) points into `Set`, called from `main`'s soil loop. The fix is one swapped pair of letters.

The panic is the friendly version of this bug. On a square grid, x·W+y never leaves the slice, so nothing panics; the world comes out transposed, every map mirrored across its diagonal, and you get to notice that visually or not at all.

This is a reason to test on grids where width and height differ. A rectangle turns a silent scrambling into a loud, addressed, first-offense panic.

## 6. Checkpoint

✓ CHECKPOINT — WHAT YOU CAN NOW DO

- Flatten any coordinate by hand, y·W + x, and run it backwards with integer division and remainder to recover the cell a slot belongs to.
- Explain what `type Terrain uint8` buys over a bare number or a string, and what `iota` is doing in the constant block.
- Say what `make([]Terrain, w*h)` allocates, what every element holds the instant it exists, and why `Rock` was placed at zero on purpose.
- Argue the flat slice against `[][]Terrain` on allocations, cache lines, and ragged rows, not on taste.
- When a transposed map or an index-out-of-range panic comes from grid code, check the order of terms in the flattening formula and explain why a square grid hides the same bug.

⚡ EXERCISES — TRY FIRST, THEN REVEAL

▼ **Exercise 1 — census. Write a method `Count(t Terrain) int` on `*Grid` that reports how many cells hold a given kind, and have `main` print the count for all three. Do the three numbers sum to 96?**

A single loop over the flat slice, no coordinates needed: this is the first job where the flat layout is strictly easier than nested slices. The hand-drawn map counts 36 rock, 14 water, 46 soil, and 36 + 14 + 46 = 96. If your rock count is 40, your rim loops overlap at the corners; if the sum is not 96, some cell is being counted twice or missed, which the flat loop makes impossible, so check that you looped over `cells` and not over coordinates.

▼ **Exercise 2 — a fourth kind of ground. Add `Sand` to the terrain type, give it a glyph, and ring the pond with a one-cell beach. Where in the constant block does the new name go, and why does the position matter?**

Append it after `Soil`, taking value 3, and extend both switches; a new case in `String` and one in `Glyph`. Appending matters because inserting `Sand` mid-block would renumber every constant below it, and the moment terrain values are written to disk (this volume's event log is coming), a renumbering silently relabels the saved world. The beach is fourteen to twenty `Set` calls or a small loop over the pond's neighbors; rerun and the map shows a pale ring where soil met water.

▼ **Exercise 3 — falling off the world. Ask the grid for `Coord{X: -1, Y: 0}` and read the panic. Then use the `In` method to build `AtOK(c Coord) (Terrain, bool)` that refuses instead of crashing, and prove both behaviors with two runs.**

The direct read panics with `index out of range [-1]`: the formula computes 0·12 + (-1) and the bounds check rejects it. `AtOK` is three lines: if `!g.In(c)`, return `Rock, false`; otherwise `g.At(c), true`. The two-value return is a Go idiom you will meet everywhere from map lookups to type checks. Returning rock for the void is a real design choice, not a shrug: to every wanderer in later volumes, the edge of the world behaves like an unbreakable cliff.

VOLUME 1 · CHAPTER 4

# One Seed, One World

A seeded generator that draws the same terrain every time

## 1. Seeded randomness

The hand-drawn pond from last chapter cannot scale past the author's patience. The rule that replaces it is the world's first randomness rule: **every random draw in this world comes from a generator built from one explicit seed, so the seed names the world completely.**

The map needs variation, some ingredient that decides where the water goes without you deciding for it. The obvious source is the operating system or the clock: something unpredictable.

The simulation has a stronger demand. When a creature drowns at tick 40,000 of a three-day run, you need to run those three days again, watch the same creature approach the same pond, and catch the bug in the act.

A world that comes out different every time cannot be debugged, cannot be tested, and cannot keep this volume's promise: a run of the server can be reproduced exactly.

The terrain must surprise you on the first run, then repeat itself, cell for cell, on every run after. That works because random does not have to mean unrepeatable. It can mean patternless to any consumer while completely determined by a starting number.

That starting number is called a seed. Tell someone the seed and you have told them where every rock, every pond, and every later seeded event came from. The proof is a tiny generator you can run in your

head, the real one wired into the terrain grid, and the same world generated twice.

## 2. The toy generator

A random number generator is smaller than the name suggests. It holds one number, the state, and it has one rule for turning the current state into the next one.

Ask it for a number and it applies the rule, remembers the result, and hands the result to you. That is the whole machine.

Here is one small enough to run by hand: the state is a number from 0 to 15, and the rule is *multiply by 5, add 3, keep the remainder after dividing by 16*.

Start the state at 7 and turn the crank. Five sevens are 35, plus 3 is 38, and 38 divided by 16 leaves remainder 6: the first output is 6. From 6, 30 plus 3 is 33, remainder 1. From 1, the result is 8. From 8, 43 leaves remainder 11.

The machine seeded with 7 begins 6, 1, 8, 11, and there is no arithmetic in it you could not do in a checkout line. A program can crank it farther.

▣ BUILD · STAGE 1 — A GENERATOR SMALL ENOUGH TO DISTRUST

```
// cmd/worldd/main.go — a throwaway main; the real one returns
in stage 2
package main

import "fmt"

// next is a toy random number generator: sixteen possible
states,
// one rule for stepping between them.
func next(state int) int {
        return (5*state + 3) % 16
}

func run(seed, n int) {
```

```
        fmt.Printf("seed %2d:", seed)
        state := seed
        for i := 0; i < n; i++ {
                state = next(state)
                fmt.Printf(" %d", state)
        }
        fmt.Println()
}

func main() {
        run(7, 18)
        run(7, 18)
        run(8, 18)
}
```

```
$ go run ./cmd/worldd

seed  7: 6 1 8 11 10 5 12 15 14 9 0 3 2 13 4 7 6 1
seed  7: 6 1 8 11 10 5 12 15 14 9 0 3 2 13 4 7 6 1
seed  8: 11 10 5 12 15 14 9 0 3 2 13 4 7 6 1 8 11 10
```

The two seed-7 lines are identical, and not by luck. The machine's next output depends only on its state, so the same starting state must produce the same sequence forever.

That is the property this whole book stands on, already present in a four-line function.

The sequence looks respectably scrambled: 6, 1, 8, 11, 10 hops around with no stride a glance can catch. Before any value repeats, all sixteen possible values have appeared exactly once.

Read seed 8's line against seed 7's. It is the same sequence, entered at a different point. With only sixteen states there is only one loop to walk, and every seed picks a doorway into it.

**∑ MATH INTERLUDE — THE WHOLE GENERATOR IN ONE LINE**

The toy is a *linear congruential generator*, and its rule in symbols is $s' = (a \cdot s + c) \bmod m$, with $a=5$, $c=3$, $m=16$ here. Check it against the hand crank: $s=7$ gives $(5 \cdot 7 + 3) \bmod 16 = 38 \bmod 16 = 6$, the first output above.

> The state can never leave 0..m−1, because a remainder after dividing by m cannot reach m. The sequence must eventually revisit a state, and the moment it does, it repeats forever.
>
> The count of outputs before that happens is the generator's *period*. The run shows this one achieving the best possible: all 16, a full lap.
>
> `s` the state: the one number the generator remembers
> `s'` the next state, which is also the next output
> `a, c` the multiplier and the increment: 5 and 3 in the toy
> `m` the modulus: how many states exist (16 in the toy), written % in Go
> `period` how many outputs appear before the sequence repeats

Everything wrong with the toy is quantity, not kind. Sixteen states means the terrain of a 96-cell map would visibly repeat before one pond finished.

The generator in Go's standard library, `math/rand/v2`, keeps 128 bits of state instead of 4, so the loop it walks has about $3.4\times10^{38}$ states. It scrambles each state before showing it to you, so consecutive outputs share no visible family resemblance.

The algorithm is called PCG, a permuted congruential generator: the same multiply-and-add heart you hand-cranked, with better output. What it does not change, because no PRNG changes it, is the contract. Seed it, and the entire sequence is fixed before the first draw.

## 3. Terrain from a seed

Now wire that contract into the ground. The generation strategy is deliberately humble: soil the interior as before, then drop a spring at a random interior cell and let the water wander.

Thirty times, the walker floods the cell under it and stumbles one cell in a random direction, refusing steps onto the rim. A path that doubles back floods cells it already flooded, so the pond comes out dense where the walk lingered: crude hydrology, but recognizably a pond, and grown instead of placed.

▣ BUILD · STAGE 2 — TERRAIN FROM A SEED

```
// internal/sim/gen.go
package sim

import "math/rand/v2"

// Generate builds a w-by-h world from one seed: a rock rim, a
soil
// interior, and a pond carved by a random walk. The same seed
always
// builds the same world.
func Generate(w, h int, seed uint64) *Grid {
        rng := rand.New(rand.NewPCG(seed, 0))

        g := NewGrid(w, h)
        for y := 1; y < h-1; y++ {
                for x := 1; x < w-1; x++ {
                        g.Set(Coord{X: x, Y: y}, Soil)
                }
        }

        // Drop a spring somewhere in the interior, then let the
water
        // wander: at each step it floods the cell it stands on
and
        // stumbles one cell in a random direction, staying off
the rim.
        c := Coord{X: 1 + rng.IntN(w-2), Y: 1 + rng.IntN(h-2)}
        steps := []Coord{{X: 1, Y: 0}, {X: -1, Y: 0}, {X: 0, Y:
1}, {X: 0, Y: -1}}
        for i := 0; i < 30; i++ {
                g.Set(c, Water)
                d := steps[rng.IntN(4)]
                n := c.Offset(d.X, d.Y)
                if n.X ⩾ 1 && n.X ⩽ w-2 && n.Y ⩾ 1 && n.Y ⩽
h-2 {
                        c = n
                }
        }
        return g
}
```

```
// cmd/worldd/main.go — the real main again
package main

import (
        "fmt"

        "theworld/internal/sim"
)

const version = "0.0.1"

func main() {
        const seed = 5
        fmt.Println("worldd", version, "seed", seed)
        g := sim.Generate(12, 8, seed)
        fmt.Print(g.Render())
}
```

```
$ go run ./cmd/worldd

worldd 0.0.1 seed 5
############
#.........#
#.........#
#.~~~~~....#
#.~~~~~~...#
#...~~~~...#
#...~......#
############
```

`rand.NewPCG(seed, 0)` constructs the generator's 128-bit starting state from two 64-bit words, and the two words do different jobs. The first is the seed, the number that names the world.

The second picks a *stream*. Build two generators from one seed and different second words and you get two unrelated sequences, so a system can have draws of its own without sharing a position with somebody else's.

This book pins the convention here: terrain generation draws from stream 0, and the world's laws draw from stream 1. The stream numbers are constants in the code, never choices anyone makes at run time, so one number you choose still names everything.

`rand.New` wraps that raw engine in the type with the convenient methods. `rng.IntN(n)` is the one this chapter leans on: a draw scaled into the range 0 to n−1, fair across it.

Count the draws and the whole run is accounted for: two to place the spring, which lands at column 5, row 3, and one per step of the walk. Thirty-two draws in all, each one pulled from the fixed sequence that seed 5 unrolls.

Read the map the way you read last chapter's, but notice who drew it. Thirty flooded steps produced only sixteen water cells, so the walk crossed its own path fourteen times; the dense southwest bulge is where it lingered.

Nobody chose those cells. Nobody can choose them: change the seed and a different pond grows in a different corner. What you chose is the number 5, and that is the only authorship left.

◆ NOTE — ONE GENERATOR PER PURPOSE, MADE WHERE IT IS USED

`Generate` builds its own `rng`, uses it, and lets it die. That is a discipline, not an accident: a generator is a consumable stream, and any code that shares one with somebody else no longer controls what it will draw. The worked failure below is what happens when this rule bends.

## 4. The same seed twice

The claim on the table is strong. Same seed, same world: not similar, not statistically alike, but equal in every one of 96 cells.

Strong claims get tested by machine, not by squinting. `Render` already flattens a whole grid into one string, and Go compares strings with `==`, byte by byte. Build the world twice, build a rival world from seed 6, and let three comparisons speak.

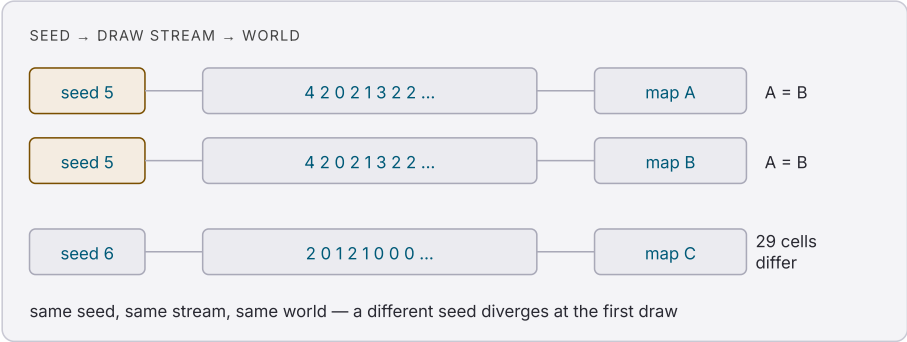
▣ BUILD · STAGE 3 — THE WORLD TESTIFIES

```
// cmd/worldd/main.go — the body of main grows a proof
func main() {
        const seed = 5
        fmt.Println("worldd", version, "seed", seed)

        first := sim.Generate(12, 8, seed)
        second := sim.Generate(12, 8, seed)
        other := sim.Generate(12, 8, seed+1)

        fmt.Print(first.Render())
        fmt.Println("same seed, same world: ", first.Render() ==
second.Render())
        fmt.Println("seed 6, same world:    ", first.Render() ==
other.Render())

        differ := 0
        for y := 0; y < first.H; y++ {
                for x := 0; x < first.W; x++ {
                        c := sim.Coord{X: x, Y: y}
                        if first.At(c) ≠ other.At(c) {
                                differ++
                        }
                }
        }
        fmt.Println("cells where seed 6 disagrees:", differ)
}
```

```
$ go run ./cmd/worldd

worldd 0.0.1 seed 5
############
#..........#
#..........#
#.~~~~~....#
#.~~~~~~...#
#...~~~~...#
#...~......#
############
same seed, same world:  true
seed 6, same world:     false
cells where seed 6 disagrees: 29
```

The first `true` is the sentence this volume is built on, printed by the program itself: two separate calls, two separate grids, two separate ponds grown step by step, and not one byte of difference between them.

> The seed-6 lines calibrate how much that means. One seed away, 29 cells disagree, because seed 6's first draws differ and every step of its walk compounds the divergence.
>
> Nearby seeds are not nearby worlds. Each seed is a doorway into a stretch of the sequence unrelated, for any purpose you care about, to its neighbor's.




## 5. Why deterministic generation works

The guarantee is airtight because nothing in the chain has anywhere to hide state you did not supply. A PRNG's next output is computed from its current state alone; the starting state is computed from the seed alone; and `Generate` consumes draws in an order fixed by its own code, two for the spring and then one per step.

Same seed, same first draw. Same first draw, same spring. Same spring and same second draw, same first step. Each equality forces the next, all the way down the walk, and the identical maps are that induction made visible.

Determinism belongs to the generator itself; the craft in PCG's design went into making a fully determined sequence *look* lawless.

This is also why the seed deserves to be called the world's name, the one metaphor this chapter needs. The 96-cell map takes 96 cells to

write down, but "seed 5" reproduces it from eight characters. Anything grown from draws is carried by the number the draws came from.

Report a bug as "seed 5, tick 300" and anyone can stand exactly where you stood. Keep a seed in a config file and a server can be rebuilt from bare metal into the same valley.

The same reasoning convicts the tempting alternatives. Seeding from the clock produces a world named by the nanosecond you happened to press enter, a name nobody can use twice.

Go's own package-level functions, `rand.IntN` and family called without a generator, are worse in a quieter way: `math/rand/v2` seeds them unpredictably at startup, on purpose, and offers no way to re-seed.

They exist for programs that want dice, and a simulation is not allowed to want dice. It wants a ledger of decisions it can replay. Every stochastic system in this book gets draws from a generator whose seed is written down.

**⚠ WORKED FAILURE — ONE STREAM, TWO DRINKERS**

Here is the version of this chapter that almost got written. It seemed tidier to build the generator once in `main` and pass it in, so that callers could share one source of randomness:

```
// The signature that seemed cleaner:
func Generate(w, h int, rng *rand.Rand) *Grid { ... }

// main, seeding once and generating twice:
rng := rand.New(rand.NewPCG(5, 0))
first := sim.Generate(12, 8, rng)
second := sim.Generate(12, 8, rng)
fmt.Println("same seed, same world: ", first.Render() ==
second.Render())
fmt.Print(second.Render())
```

```
$ go run ./cmd/worldd
```

```
worldd 0.0.1 seed 5
same seed, same world:  false
############
#.......~~.#
#........~~#
#........~~#
#........~~#
#.....~~~~~#
#...~~~~...#
############
```

The proof line reads `false`, and the printed second map is a pond seed 5 never grew. Reason from the symptom.

Both calls saw the same generator. The worlds differ because a generator is not a value; it is a stream with a position.

The first `Generate` consumed draws one through thirty-two. The second began at draw thirty-three, mid-sequence, and grew its pond from numbers that belong to no seed anyone will ever type.

The world it built is an orphan: real, valid, and unreproducible, exactly the kind of world this chapter exists to forbid.

The fix is the code you already have. A world's generator is born from the seed inside `Generate` and dies there, making draw one of world A and draw one of world B the same draw.

When a stream is shared across systems, the sharing is an explicit design with the draw order pinned. It is never an accident of a convenient signature.

## 6. Checkpoint

✓ CHECKPOINT — WHAT YOU CAN NOW DO

- Hand-crank a linear congruential generator, (5s + 3) mod 16 from any seed, and predict its output before a program confirms it.

- Explain why a PRNG's sequence is fixed the moment it is seeded, and why that makes "random" and "repeatable" compatible instead of contradictory.
- Say what `rand.NewPCG(seed, 0)` and `rng.IntN(n)` each contribute, and account for every draw `Generate` consumes, in order.
- Prove same-seed determinism the mechanical way: two generated grids compared as strings with `==`, no squinting involved.
- Given a "same seed, different world" symptom, check whether two consumers are drinking from one shared stream before suspecting the generator itself.
- Say why a simulation must not touch the auto-seeded package-level `rand` functions, even though they compile fine.

⚡ EXERCISES — TRY FIRST, THEN REVEAL

▼ **Exercise 1 — the seed safari. Loop seeds 1 through 12, generating and rendering each. Every one of those worlds already existed before you ran the loop; pick the one you would want to live in, and note its number.**

A three-line loop in `main`: `for seed := uint64(1); seed ⩽ 12; seed++`, print the seed, print the render. You will find real variety: seed 3 grows a two-lobed pond with a channel, seed 4's water hugs the western wall like a river, seed 7 puddles in the northwest corner. Run the loop twice and the safari itself repeats exactly, which is the chapter's proof at gallery scale. Whatever number you picked, write it down; a favorite seed is a world you can come back to.

▼ **Exercise 2 — break the toy. In the stage 1 generator, change the increment from 3 to 4 and run seeds 7 and 8 again. The period was 16; what is it now, and what does the**

**answer say about how carefully real generator constants are chosen?**

Seed 7 prints `7 7 7 7 …`: since 5·7 + 4 = 39 and 39 mod 16 is 7, the state is a fixed point and the period collapses to 1. Seed 8 fares barely better, cycling `12 0 4 8` forever, period 4. One nudged constant turned a full-lap generator into a stopped clock, and which disaster you get depends on the seed. The constants in a real PRNG are not adjustable knobs; they are load-bearing numbers with proofs attached, and the library's job is to have chosen them so you never think about it.

▼ **Exercise 3 — the second word. `NewPCG` takes two 64-bit words and `Generate` passes 0 as the second. Change that 0 to 1, keep seed 5, and run the proof again. Is it still the same world?**

The two grids still match each other, so the proof line stays `true`; what changed is which world seed 5 means. Compare the map against the one printed above and it is a different pond: a lobe stretched across the north with one dry cell trapped inside it, and a second patch down the western side. Both words are part of the starting state, so the world's true name is really the pair (5, 0), and terrain's word is a constant 0 precisely so that one number stays a complete name. Changing it does not hand you another seed's world; it hands you another stream of the same seed, which is the mechanism the stream convention rests on. Put the 0 back before moving on, and the pond returns to the one this chapter grew.

VOLUME 1 · CHAPTER 5

# Something in the World

Entities as data, identified by an ID that outlives them

## 1. The grid says ground

The map has a rock rim, a soil floor, and a pond, but it cannot answer where a shrub stands. The split is permanent: **the grid says what the ground is; entities say what stands on it, and no fact lives in both places.**

A terrain grid can only say what the ground *is*. Everything built so far is *of* the world: rock, water, soil, properties of squares. Nothing yet is *in* it.

A world becomes a world when the first thing stands on the ground instead of being the ground.

Extending the terrain type looks tempting: add `ShrubOnSoil` and `WalkerOnSoil` constants next to `Rock` and `Soil`, and let the grid carry both facts.

That road collapses quickly. Every thing-on-ground combination needs a constant, and when something moves, the grid must remember what ground to restore behind it.

The grid also cannot say *which* shrub is which. Two shrubs are two identical bytes in a slice. Eat one, and no record anywhere can say whether it was the one by the pond or the one by the rim.

The ground is dense, one value per cell, because every cell has ground. Entities are sparse: a handful now, thousands later, most cells

holding none. Dense facts live in the flat slice you already built. Sparse facts need a different container.

Things in a world also change. A creature moves, grows, gets hurt, heals, until almost everything about it differs from the day it appeared.

For the world to say "this is still the same one," something about it must stay permanent. The fix is old and everywhere: issue each entity a number when it enters the world, never change it, and never give it to anything else, even after the entity dies. The number is the identity. Everything else is current condition.

## 2. The Entity struct

The entity itself is small, and it should be. What is it, where is it, and which one is it: three fields.

Kinds get the treatment terrain kinds got, a defined type with `iota` constants. The volume starts with two: a shrub that roots in soil, and a walker that goes where it likes. Neither behaves yet. They exist, which today is the whole point.

▣ BUILD · STAGE 1: THE ENTITY TYPE, AND A SLICE THAT ALMOST WORKS

```
// internal/sim/entity.go
package sim

// EntityID names one entity for its whole life. IDs are handed
// out once, in order, and never reused, even after a removal.
type EntityID uint64

// EntityKind is what sort of thing an entity is.
type EntityKind uint8

const (
        Shrub EntityKind = iota
        Walker
)

// String makes an EntityKind print as a word.
func (k EntityKind) String() string {
```

```
        switch k {
        case Shrub:
                return "shrub"
        case Walker:
                return "walker"
        }
        return "unknown"
}

// Glyph is the one-byte map symbol for an entity kind.
func (k EntityKind) Glyph() byte {
        if k == Walker {
                return '@'
        }
        return 'o'
}

// Entity is one thing standing in the world: what it is, where
it
// is, and the ID that stays the same while both of those
change.
type Entity struct {
        ID   EntityID
        Kind EntityKind
        At   Coord
}


// cmd/worldd/main.go — a throwaway main: store entities the way
// you already know how, and find one
func main() {
        ents := []sim.Entity{
                {ID: 1, Kind: sim.Shrub, At: sim.Coord{X: 2, Y:
5}},
                {ID: 2, Kind: sim.Shrub, At: sim.Coord{X: 9, Y:
2}},
                {ID: 3, Kind: sim.Walker, At: sim.Coord{X: 10,
Y: 6}},
        }
        want := sim.EntityID(3)
        for _, e := range ents {
                if e.ID == want {
                        fmt.Println("found:", e.Kind, "at",
e.At)
                }
        }
        fmt.Println("looked at", len(ents), "entities to find
one")
}
```

```
$ go run ./cmd/worldd

found: walker at {10 6}
```

```
looked at 3 entities to find one
```

`EntityID` is a defined type over `uint64` for the same reason `Terrain` sat over `uint8`: an ID is not a count, and the compiler should refuse to let one impersonate the other.

Sixty-four bits is deliberate too. IDs are never reused, and at even a million spawns a second a `uint64` outlasts any machine you will ever own.

The storage is the honest problem in this stage. A slice answers "find entity 3" by walking every element and comparing. Three entities, three looks, fine.

The valley this book is heading toward holds thousands, and the systems that run it ask "where is entity so-and-so" constantly: every attack names a target, every memory names who it happened with, every log line names its actor.

Answering each by scanning the whole population turns the sim into a search engine for its own contents.

Be precise about what the slice is bad at. It holds entities beautifully, packed and iterable; it cannot *jump straight to one* when all you hold is its ID.

The grid solved this for cells with arithmetic, a coordinate computing its own slot. IDs climb forever and most are eventually dead, so "slot = ID" would mean a slice as long as every entity that ever lived.

The container you want acts as if that arithmetic existed: hand it a key, get the value, no scan. Go ships one.

## 3. The World map

A *map* is Go's built-in key-to-value store. `map[EntityID]*Entity` reads as "a map from entity IDs to entity pointers". Hand it an ID

between square brackets and it hands back the pointer stored under that ID, in effectively constant time, whether it holds three entries or three hundred thousand.

You create one with `make`, like a slice; store with `m[id] = e`; read with `m[id]`; remove with the built-in `delete(m, id)`; count with `len(m)`.

Reading a key that was never stored does not fail. It returns the value type's zero value, `nil` for a pointer. Go offers a second form, `e, ok := m[id]`, where `ok` answers whether this key was actually present.

> ◆ NOTE: A MAP MUST BE MADE BEFORE IT IS WRITTEN
>
> `var m map[EntityID]*Entity` declares a map holding its zero value, `nil`: reads on a nil map politely return zero values, but a write panics with `assignment to entry in nil map`. Constructors like the `NewWorld` below exist for this: the one place a `World` is born is the one place its map is guaranteed `make`d.

The map needs an owner. Placement is a negotiation between an entity and the ground under it, so the type holding the entities must also see the grid.

Call it `World`, and give it the spawning rules, the lookup, and the ID counter that makes identity real.

> ▣ BUILD · STAGE 2: WORLD, THE GRID, THE MAP, AND THE RULES OF STANDING

```
// internal/sim/entity.go — add the placement rule to the kind

// CanStand reports whether this kind of entity may occupy
ground of
// kind t. Shrubs root only in soil; a walker stands anywhere
dry.
func (k EntityKind) CanStand(t Terrain) bool {
        switch k {
        case Shrub:
```

```
                return t == Soil
        case Walker:
                return t != Water
        }
        return false
}


// internal/sim/world.go
package sim

// World owns the ground and everything standing on it.
type World struct {
        Ground *Grid
        ents   map[EntityID]*Entity
        nextID EntityID
}

// NewWorld wraps a terrain grid in a world with nothing
standing on it.
func NewWorld(g *Grid) *World {
        return &World{
                Ground: g,
                ents:   make(map[EntityID]*Entity),
                nextID: 1,
        }
}

// Spawn creates an entity of kind k at c and returns its new
ID.
// It refuses, returning 0 and false, if c is off the grid or
the
// ground there cannot hold this kind of entity.
func (w *World) Spawn(k EntityKind, c Coord) (EntityID, bool) {
        if !w.Ground.In(c) || !k.CanStand(w.Ground.At(c)) {
                return 0, false
        }
        id := w.nextID
        w.nextID++
        w.ents[id] = &Entity{ID: id, Kind: k, At: c}
        return id, true
}

// Get returns the entity with this ID, and whether it exists at
all.
func (w *World) Get(id EntityID) (*Entity, bool) {
        e, ok := w.ents[id]
        return e, ok
}

// Remove takes an entity out of the world. Its ID retires with
it.
func (w *World) Remove(id EntityID) {
        delete(w.ents, id)
```

```
}

// Population is how many entities the world holds right now.
func (w *World) Population() int {
        return len(w.ents)
}


// cmd/worldd/main.go — rebuild chapter 3's hand-drawn valley
// (the soil loop and the pond literal, unchanged), then
populate it
        w := sim.NewWorld(g)
        id1, ok := w.Spawn(sim.Shrub, sim.Coord{X: 2, Y: 5})
        fmt.Println("shrub:", id1, ok)
        id2, ok := w.Spawn(sim.Shrub, sim.Coord{X: 9, Y: 2})
        fmt.Println("shrub:", id2, ok)
        id3, ok := w.Spawn(sim.Walker, sim.Coord{X: 10, Y: 6})
        fmt.Println("walker:", id3, ok)
        id4, ok := w.Spawn(sim.Walker, sim.Coord{X: 5, Y: 3})
        fmt.Println("walker on the pond:", id4, ok)
        fmt.Println("population:", w.Population())
```

```
$ go run ./cmd/worldd

shrub: 1 true
shrub: 2 true
walker: 3 true
walker on the pond: 0 false
population: 3
```

The fourth spawn is the placement rule earning its keep: {5, 3} is open water, a walker cannot stand there, and the world said no.

Nothing panicked and nothing was half-created. `Spawn` checks before it allocates, so a refused spawn leaves no trace, not even a burned ID.

The rule lives on `EntityKind`, not buried inside `Spawn`, because "can this kind occupy that ground" is a question other systems ask about squares nobody is spawning into.

The lowercase fields of `World` matter too: the map and the counter are private, so no caller can hand out an ID or plant something in the pond by reaching around the methods.

▣ BUILD · STAGE 3: IDENTITY SURVIVES WHAT HAPPENS TO ITS OWNER

```
// cmd/worldd/main.go — continue after the population line
        if e, ok := w.Get(id3); ok {
                fmt.Println("id", e.ID, "is a", e.Kind, "at",
e.At)
                e.At = e.At.Offset(0, -1)
        }
        if e, ok := w.Get(id3); ok {
                fmt.Println("id", e.ID, "is a", e.Kind, "at",
e.At)
        }

        w.Remove(id1)
        _, alive := w.Get(id1)
        fmt.Println("id 1 still in the world:", alive)
        id5, _ := w.Spawn(sim.Shrub, sim.Coord{X: 2, Y: 5})
        fmt.Println("replacement shrub's id:", id5)
```

```
$ go run ./cmd/worldd

shrub: 1 true
shrub: 2 true
walker: 3 true
walker on the pond: 0 false
population: 3
id 3 is a walker at {10 6}
id 3 is a walker at {10 5}
id 1 still in the world: false
replacement shrub's id: 4
```

The walker moved one row north and its ID did not flicker: position changed, identity held.

The mutation landed because the map stores `*Entity`, pointers. What `Get` returns is the entity in the world, not a souvenir copy, so `e.At = ...` writes through: the chapter 2 lesson about addresses, applied to a container.

The shrub that replaced the removed one got ID 4, not a recycled 1. Recycling would save nothing and cost everything: the moment the world keeps records, a reused ID makes old records ambiguous about who they meant. Retired numbers stay retired.

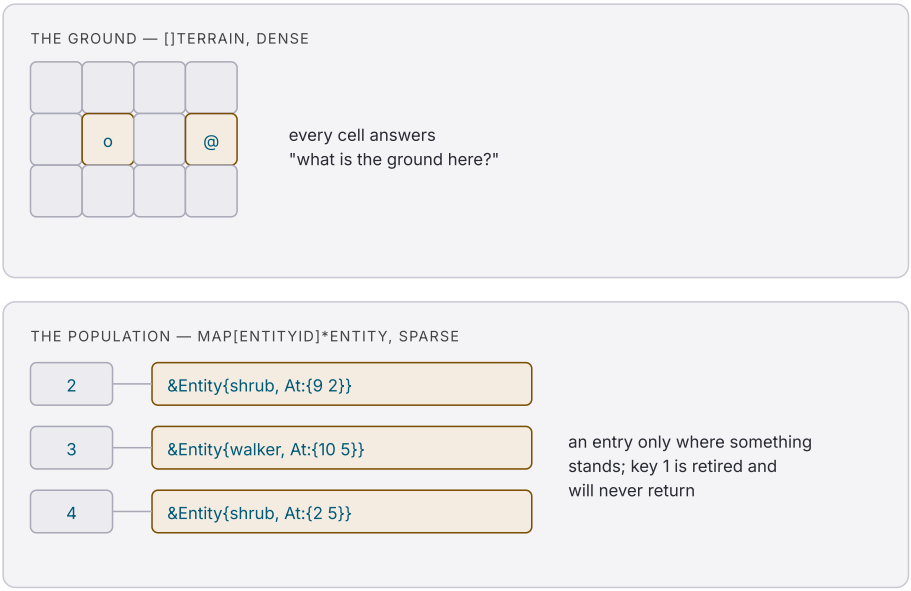



**Figure 5.1** — the world's two stores after stage 3. Dense facts (ground) live in the slice, addressed by arithmetic; sparse facts (things) live in the map, addressed by identity. The glyphs on the left are the same entities the keys on the right own.

## 4. Rendering entities

The map view should show the newcomers: render the ground as before, then stamp each entity's glyph over its cell.

▣ BUILD · STAGE 4: ENTITIES ON THE MAP

```
// internal/sim/world.go — add

// Render draws the ground with every entity drawn over its
cell.
func (w *World) Render() string {
        rows := []byte(w.Ground.Render())
        for _, e := range w.ents {
                rows[e.At.Y*(w.Ground.W+1)+e.At.X] =
e.Kind.Glyph()
        }
        return string(rows)
}
```

```
$ go run ./cmd/worldd

(map portion of the output)

############
#..........#
#...~~~~.o.#
#..~~~~~~..#
#...~~~~...#
#.o.......@#
#..........#
############
```

The `W+1` is not a typo: the rendered ground is text, each row `W` glyphs plus a newline, so a row of the byte slice is one wider than a row of the grid.

The loop is your first *iteration over a map*: `range` visits every key-value pair. Here the visiting order cannot matter, since each entity stamps only its own cell.

Where order can matter, Go has an opinion, and it is about to become this chapter's mistake.

**⚠ WORKED FAILURE: THE ROSTER THAT RESHUFFLED ITSELF**

A world should list its population on demand. The obvious method writes itself: walk the map, collect the entities, print them.

```
// internal/sim/world.go — the obvious attempt
// Roster returns every entity in the world.
func (w *World) Roster() []*Entity {
        out := make([]*Entity, 0, len(w.ents))
        for _, e := range w.ents {
                out = append(out, e)
        }
        return out
}
```

```
$ go run ./cmd/worldd

(roster portion)
```

```
roster:
  4 shrub at {2 5}
  2 shrub at {9 2}
  3 walker at {10 5}
```

```
$ go run ./cmd/worldd

(again, nothing changed)

roster:
  2 shrub at {9 2}
  3 walker at {10 5}
  4 shrub at {2 5}
```

Same code, same seed, same three entities, different order. Your own runs land in their own orders, possibly agreeing for several runs before they betray you.

Reason from the symptom: the entities are identical across runs, so what varies must be the order `range` visits the map. That is exactly right.

Go deliberately randomizes map iteration order, starting each walk at a random bucket, so no program can accidentally depend on an order the language never promised.

For most programs that is a lint. For this one it is a fire alarm wired to the book's central promise: chapter 4 swore the same seed produces the same world, byte for byte, and here are two byte-different outputs from one seed.

Any map iteration whose results reach the world's output, its log, or its state must be forced into an order you choose. The fix costs four lines: collect the keys, sort them, walk them.

▣ BUILD · STAGE 5: THE ROSTER, IN AN ORDER YOU CHOSE

```
// internal/sim/world.go — the fix; add "slices" to the imports

// Roster returns every entity in the world, ordered by ID.
```

```
func (w *World) Roster() []*Entity {
        ids := make([]EntityID, 0, len(w.ents))
        for id := range w.ents {
                ids = append(ids, id)
        }
        slices.Sort(ids)
        out := make([]*Entity, 0, len(ids))
        for _, id := range ids {
                out = append(out, w.ents[id])
        }
        return out
}
```

```
$ go run ./cmd/worldd

(roster portion — now identical on every run)

roster:
  2 shrub at {9 2}
  3 walker at {10 5}
  4 shrub at {2 5}
```

`range` with one variable over a map yields keys only. `slices.Sort`, from the standard library's `slices` package, sorts them in place, ascending.

Twelve runs in a row now produce twelve identical rosters, and ID order means something: entities list in the order they entered the world, oldest first, because IDs were issued in that order and never shuffled.

Identity, handed out for one purpose, quietly pays for another.

## 5. Why maps fit entities

A map feels like magic from the outside: any key, any of a million entries, one step. Under the hood it is the grid's own trick in disguise.

The flat slice turned a coordinate into a slot with `y·W + x`. A map turns a key into a slot with a *hash function*, a computation that scrambles the key's bytes into a number.

That number, reduced to the size of the map's internal array, picks the bucket where the value lives. Lookup is: hash the key, go to the bucket, check the few entries there. No scan, at any size.

The costs are the ones the grid never paid. Buckets need slack to keep collisions rare, so a map spends more memory per entry than a slice. Hashing costs more than one multiply. Entries land wherever their hash sends them, so neighbors in the map are strangers in memory.

That is why iterating a map will never match sweeping a slice. Read it as a fee schedule: dense, ordered, swept-every-tick data belongs in slices; sparse, keyed, jumped-to data belongs in maps. The ground and the population landed on opposite sides of that line, so the world now holds one of each.

The randomized iteration order follows from the same mechanics. Where an entry lands depends on its hash and the map's current internal size, so even without deliberate randomization, order would shift as the map grew.

Go adds the random start to make the lesson unavoidable in testing instead of catastrophic in production. The standing rule for this project: a map is a bag, not a queue, and any time the world speaks, logs, or acts on its population, the order comes from a sort, never from `range`.

The ID scheme outlives every container decision. The entities on the map today are frozen scenery: the walker moved because `main` moved it, and nothing in the world moves anything on its own, because nothing yet says *when*.

The population map is the thing a tick loop can sweep. Entity 3 can move later, and the only thread connecting "the walker that drank at the pond" to "the walker that died on the rim" days later is the number that never changed.

Identity is what makes a history possible. The world remembers *who*, not a square that happened to hold something once.

## 6. Checkpoint

✓ CHECKPOINT: WHAT YOU CAN NOW DO

- Say which facts belong in the terrain grid and which belong in the entity map, and defend the dense-versus-sparse line that separates them.
- Create a map with `make`, store, read, `delete`, and `len` it, and use `e, ok := m[id]` to tell a missing key from a stored zero value.
- Explain why `Spawn` returned `0 false` for the pond walker, and where the rule that refused it lives.
- Trace why `e.At = e.At.Offset(0, -1)` changed the entity in the world, naming the pointer in `map[EntityID]*Entity` that made it possible.
- State the ID policy: issued in order, never changed, never reused, with the record-keeping reason recycling ID 1 would poison.
- When a program's output order changes between identical runs, suspect an unsorted map iteration first, and write the collect-sort-walk fix from memory.

⚡ EXERCISES: TRY FIRST, THEN REVEAL

▼ **Exercise 1: who is standing here? Write `AtCell(c Coord) []*Entity` on `*World`: every entity whose position is `c`, in a deterministic order. Prove it by asking about the replacement shrub's square and an empty one. What did this lookup cost, compared to `Get`?**

The map is keyed by ID, not by place, so this question scans: loop over `Roster()` (already sorted, so the order is settled) and keep entities where `e.At == c`; struct equality works because `Coord` is two comparable ints. The shrub's square

prints one entity, the empty square none, and the cost is the full population per call, the exact price the slice charged for `Get`. A store is fast only for the key it is organized under; when position lookups become hot, the sim needs a second structure organized by place. This scan is the cost that justifies it.

▼ **Exercise 2: census by kind. Build a `map[EntityKind]int` that counts the population by kind, then print shrubs and walkers. What does `counts[k]++` do the first time a kind appears, before any entry for it exists?**

Loop over the roster and increment `counts[e.Kind]++`. The first increment for a kind reads a missing key, and a missing key reads as the value type's zero: 0. So `++` turns "absent" into 1 with no existence check; that behavior makes maps natural counters in Go. The print shows 2 shrubs, 1 walker. If you printed by ranging over `counts` itself, you know which trap you just re-armed.

▼ **Exercise 3: refused for two different reasons. `Spawn` a shrub at {0, 0} and a walker at {-1, 3}. Both return `0 false`. Which of `Spawn`'s two checks refused each, and why does the order of those checks matter?**

The shrub failed `CanStand`: {0, 0} is on the grid, but it is rim rock and shrubs root only in soil. The walker failed `In`: {-1, 3} is off the grid entirely. The order matters because `CanStand` needs `At`, and `At` on an off-grid coordinate panics; `In` standing guard first means the ground is only consulted about squares that exist. Swap the two conditions and the walker spawn crashes instead of returning `false`. Try it; reading that panic on purpose is cheap insurance.

VOLUME 1 · CHAPTER 6

# Errors That Don't Lie

Returned errors and the bounds check a bad coordinate can't get past

## 1. Bad coordinates

The world has ground, a seed, and its first inhabitants, and it still trusts every request that reaches it. The boundary changes here: **a function that can fail returns an error as an ordinary value, and the caller reads it before touching the result.**

`Grid.At` believes every coordinate it is handed. `Grid.Set` writes wherever it is told. The entity map hands back whatever lives under an ID, including nothing.

Chapter 3's worked failure showed what that trust costs: one wrong index and the program died mid-run with `index out of range`. Back then the crash was almost a favor, a loud arrow pointing at broken arithmetic.

That crashed program was a demo you restarted with one keystroke. A server running The Hollow for months does not get to treat a bad coordinate as a full-process event; one bad request cannot take the valley with it.

A request into the simulation can already be wrong in several ways. A coordinate can name a cell that does not exist, off any of four edges. A placement can ask for ground that refuses the occupant: a walker cannot stand in the pond. A lookup can ask for an entity ID the map has never held, or held once and no longer does.

None of these are broken arithmetic. They are requests the world should decline, made by code that deserves an answer. Go's error

values let the world say exactly why and keep ticking.

## 2. Walking off the map

Start with the damage as it stands. The pieces on the bench are the hand-set valley from chapter 3 and the entity layer exactly as the last chapter left it.

Predictable terrain makes error demos honest, so the pond returns for one more chapter. The entity layer has no method that moves anything.

Chapter 5 gave the world `Spawn`, `Get`, and the placement rule `CanStand`. Walking is something `main` does for itself, by taking the pointer `Get` hands back, asking the ground what is there, and writing a new coordinate into the entity if the kind can stand on it.

**⚠ WORKED FAILURE: TWO WALKERS MARCHED WEST, NOBODY CHECKING**

```
// cmd/worldd/main.go — walking, the only way chapter 5 left
open

// marchWest steps one entity one cell west, n times, refusing
only
// ground its kind cannot stand on.
func marchWest(w *sim.World, id sim.EntityID, n int) {
        for i := 0; i < n; i++ {
                e, _ := w.Get(id)
                to := e.At.Offset(-1, 0)
                if e.Kind.CanStand(w.Ground.At(to)) {
                        e.At = to
                }
                fmt.Println("walker", id, "at", e.At)
        }
}


// cmd/worldd/main.go — after building the chapter 3 valley
        w := sim.NewWorld(g)

        scout, _ := w.Spawn(sim.Walker, sim.Coord{X: 2, Y: 5})
        marchWest(w, scout, 7)
```

```
        watch, _ := w.Spawn(sim.Walker, sim.Coord{X: 1, Y: 0})
        marchWest(w, watch, 2)
```

```
$ go run ./cmd/worldd

walker 1 at {1 5}
walker 1 at {0 5}
walker 1 at {-1 5}
walker 1 at {-2 5}
walker 1 at {-3 5}
walker 1 at {-4 5}
walker 1 at {-4 5}
walker 2 at {0 0}
panic: runtime error: index out of range [-1]

goroutine 1 [running]:
theworld/internal/sim.(*Grid).At(...)
        /home/you/theworld/internal/sim/terrain.go:62
main.marchWest(...)
        /home/you/theworld/cmd/worldd/main.go:17
main.main()
        /home/you/theworld/cmd/worldd/main.go:46 +0x33b
exit status 2
```

Two walkers, two different disasters, one cause. Walker 1 leaves the map at `{-1 5}` and nothing objects, because nothing checked.

Trace the arithmetic: `At({-1, 5})` computes slot 5·12 + (−1) = 59, a perfectly legal slot that belongs to `{11 4}`, the east rim one row up.

The flattening formula, fed a coordinate that names no cell, quietly hands back some *other* cell, and the walker keeps marching through negative territory reading phantom ground.

It finally halts at `{-4 5}`: slot 5·12 − 5 = 55 is `{7 4}`, the pond's south-east corner, so the walker, four squares off the map, was stopped by water it never stood next to. Live state now contains an entity at a coordinate the grid cannot answer for.

Walker 2 shows the other face. On row 0, stepping to `{-1 0}` computes slot −1, and the runtime kills the process, taking walker 1, the terrain, and everything else with it.

> The panic is the lucky symptom of the bug, and only row 0 gets the luck. Everywhere else the same mistake corrupts silently, and chapter 3 warned this exact species survives on any coordinate whose bad index still lands in range.
>
> A bounds check fixes both faces at once, and the interesting question is what the check should *do* when it fires. Crashing on purpose makes every row behave like row 0. The world needs a way to say no and survive.

## 3. Grid errors

In Go, `error` is a type like any other: an interface satisfied by anything with an `Error() string` method. That makes an error an ordinary value you can return, store, compare, and print.

A function that can fail returns it as an extra result, last by convention. `nil`, the value meaning "no error here", is the success signal.

The simplest way to mint an error is `errors.New`. The more useful way, once one exists, is `fmt.Errorf`, which builds a message and can wrap another error inside it. Both grid methods get the treatment.

▣ BUILD · STAGE 1: AT AND SET WITH A BOUNCER AT THE DOOR

```
// internal/sim/terrain.go — the imports grow, and one sentinel
appears
import (
        "errors"
        "fmt"
        "strings"
)

// ErrOutOfBounds reports a coordinate that names no cell on
this grid.
var ErrOutOfBounds = errors.New("coordinate is off the grid")

// At returns the terrain at a coordinate, or an error if the
// coordinate names no cell.
func (g *Grid) At(c Coord) (Terrain, error) {
```

```
        if !g.In(c) {
                return Rock, fmt.Errorf("terrain at %v: %w", c,
ErrOutOfBounds)
        }
        return g.cells[g.index(c)], nil
}

// Set overwrites the terrain at a coordinate, or refuses if the
// coordinate names no cell.
func (g *Grid) Set(c Coord, t Terrain) error {
        if !g.In(c) {
                return fmt.Errorf("set terrain at %v: %w", c,
ErrOutOfBounds)
        }
        g.cells[g.index(c)] = t
        return nil
}


// cmd/worldd/main.go — poke the new contract
        t, err := g.At(sim.Coord{X: 5, Y: 3})
        fmt.Println("at {5 3}:", t, err)
        t, err = g.At(sim.Coord{X: -1, Y: 5})
        fmt.Println("at {-1 5}:", t, err)
        err = g.Set(sim.Coord{X: 12, Y: 0}, sim.Soil)
        fmt.Println("set {12 0}:", err)
```

```
$ go run ./cmd/worldd

at {5 3}: water <nil>
at {-1 5}: rock terrain at {-1 5}: coordinate is off the grid
set {12 0}: set terrain at {12 0}: coordinate is off the grid
```

Read the signature first: `At` now returns two values, and Go makes the caller catch both. That is the idiom's teeth.

You cannot take the terrain and ignore the fact that there might not be any. The compiler rejects a call that binds one result of a two-result function.

The second poke shows a refusal: `{-1 5}`, the exact coordinate the scout corrupted itself with, now comes back with an error naming the operation, the coordinate, and the objection.

It also comes back with `rock`, and that first result is filler. When `err` is non-nil, the other return exists only because the signature

> demands something, and a caller that touches it before checking `err` has already lost. Error first, result second, every time.
>
> Two details carry weight. `ErrOutOfBounds` is a package-level variable, a *sentinel*: one shared error value that means this one condition, exported so callers can recognize it later.
>
> The `%w` verb in `fmt.Errorf` wraps the sentinel inside the longer message instead of pasting its text. The returned error carries the readable sentence on the outside and the identity of `ErrOutOfBounds` intact on the inside. Stage 3 collects that debt.
>
> One consequence stays inside the package. `Render` called the old `At` in its loop, and its coordinates are generated by the loop itself, provably on the grid. It now reads `g.cells[g.index(...)]` directly.
>
> `Generate` is the same case. Its soil loops run between the rim's bounds and its water walk refuses any step that would leave the interior, so every coordinate it writes is one it just proved, and it too assigns through `g.cells[g.index(...)]`.
>
> The boundary checks. Interior code that manufactures its own valid coordinates does not pay the toll twice.

## 4. Entity errors

The entity layer has more ways to disappoint a caller, so it gets two sentinels of its own: one for an ID the map does not hold, one for ground that refuses an occupant.

The placement rule itself moves into a small unexported method, `allows`, that both `Spawn` and `Move` consult. The definition of "may stand here" lives in exactly one place, the same discipline that kept the flattening formula in one function.

**▣ BUILD · STAGE 2: SPAWN, GET AND MOVE THAT ANSWER FOR THEMSELVES**

```
// internal/sim/entity.go — sentinels and the one placement rule
var (
        // ErrNoEntity reports an ID the world has no entity
for.
        ErrNoEntity = errors.New("no entity with that ID")
        // ErrBlocked reports terrain that refuses to hold an
entity.
        ErrBlocked = errors.New("terrain refuses the entity")
)

// allows reports whether kind k may stand at a coordinate:
// nil for yes, an error naming the objection for no.
func (w *World) allows(k EntityKind, at Coord) error {
        t, err := w.Ground.At(at)
        if err ≠ nil {
                return err
        }
        if k == Walker && t == Water {
                return fmt.Errorf("%v at %v is %v: %w", k, at,
t, ErrBlocked)
        }
        return nil
}

// Spawn places a new entity and returns its ID, or refuses.
func (w *World) Spawn(k EntityKind, at Coord) (EntityID, error)
{
        if err := w.allows(k, at); err ≠ nil {
                return 0, fmt.Errorf("spawn: %w", err)
        }
        id := w.nextID
        w.nextID++
        w.ents[id] = &Entity{ID: id, At: at, Kind: k}
        return id, nil
}

// Get returns the entity with the given ID, or refuses.
func (w *World) Get(id EntityID) (*Entity, error) {
        e, ok := w.ents[id]
        if !ok {
                return nil, fmt.Errorf("entity %d: %w", id,
ErrNoEntity)
        }
        return e, nil
}

// Move steps an entity to a new coordinate, or refuses and
// leaves the entity where it was.
func (w *World) Move(id EntityID, to Coord) error {
```

```
        e, ok := w.ents[id]
        if !ok {
                return fmt.Errorf("move entity %d: %w", id,
ErrNoEntity)
        }
        if err := w.allows(e.Kind, to); err ≠ nil {
                return fmt.Errorf("move entity %d: %w", id, err)
        }
        e.At = to
        return nil
}


// cmd/worldd/main.go — three requests, two of them doomed
        w := sim.NewWorld(g)

        scout, err := w.Spawn(sim.Walker, sim.Coord{X: 2, Y: 5})
        fmt.Println("spawned walker:", scout, err)

        drowned, err := w.Spawn(sim.Walker, sim.Coord{X: 5, Y:
3})
        fmt.Println("spawned walker:", drowned, err)

        _, err = w.Get(99)
        fmt.Println("lookup 99:", err)
```

```
$ go run ./cmd/worldd

spawned walker: 1 <nil>
spawned walker: 0 spawn: walker at {5 3} is water: terrain
refuses the entity
lookup 99: entity 99: no entity with that ID
```

Follow one refusal end to end, because the layering is the lesson. `Spawn` asks `allows`; `allows` asks the grid, gets `water` back cleanly, applies its own rule, and returns `ErrBlocked` wrapped in a sentence naming the kind, the coordinate, and the ground.

`Spawn` wraps that once more with the operation that failed. Each layer adds the context only it knows, and the final message reads as a complete diagnosis a log file can stand on its own.

The comma-ok form `e, ok := w.ents[id]` distinguishes "absent" from "present". The absent case becomes a real answer with the missing ID in it, not a `nil` pointer waiting to detonate in whoever asked.

> Every mutating path now refuses *before* touching state; a declined `Move` leaves the walker exactly where it was.

A caller still has to react to *which* error it got. A creature bumping `ErrBlocked` can try another direction; code seeing `ErrOutOfBounds` from its own computed coordinate has found a bug.

The messages differ, but string-matching messages is a trap. The wrapping that made them readable also made them unstable as identifiers.

This is what the sentinels and `%w` were for: `errors.Is` walks the wrapped chain and reports whether a given sentinel is anywhere inside it.

**▣ BUILD · STAGE 3: THE MARCH REPLAYED, AND ERRORS.IS NAMING THE CAUSE**

```
// cmd/worldd/main.go — the failure's march, against the
hardened API
        w := sim.NewWorld(g)
        scout, err := w.Spawn(sim.Walker, sim.Coord{X: 2, Y: 5})
        must(err)

        at := sim.Coord{X: 2, Y: 5}
        for {
                next := at.Offset(-1, 0)
                if err := w.Move(scout, next); err ≠ nil {
                        fmt.Println("stopped:", err)
                        fmt.Println("off the grid?",
errors.Is(err, sim.ErrOutOfBounds))
                        fmt.Println("blocked by terrain?",
errors.Is(err, sim.ErrBlocked))
                        break
                }
                at = next
                fmt.Println("walker", scout, "at", at)
        }

        e, err := w.Get(scout)
        must(err)
        fmt.Println("the world is still running; the walker
stands at", e.At)
```

```
$ go run ./cmd/worldd

walker 1 at {1 5}
walker 1 at {0 5}
stopped: move entity 1: terrain at {-1 5}: coordinate is off
the grid
off the grid? true
blocked by terrain? false
the world is still running; the walker stands at {0 5}
```

Set this run beside the worked failure, same walker, same westward march. Before: six moves deep into negative coordinates, halted by phantom water, state corrupted, and a sibling walker's identical mistake killing the process.

After: two legal steps, one refusal at the exact edge, and a walker standing at `{0 5}` in a world still running.

`errors.Is` answers `true` for `ErrOutOfBounds` through two layers of wrapping, because `%w` preserved the chain: the move wrapper, the terrain message, the sentinel at the core.

A plain `==` against the sentinel would answer `false` here, since the outermost error is the wrapper, not the sentinel itself.

That is the pattern this book needs: sentinels for the conditions callers decide on, `%w` to add context without destroying identity, `errors.Is` to ask. Deeper machinery can wait until the world has a use for it.

The `must(err)` helper deserves its two lines, because it looks like a cheat and is a policy. `main` builds its valley from loop counters that cannot leave the grid, so if `Set` refuses one of them, the request was not bad; *the program* is.

`must` panics on any error it is handed. A returned error and a panic answer different questions: the error says "this request cannot be honored", the panic says "this code does not do what its author believed".

> The first is Tuesday for a server. The second should be loud, immediate, and fatal, and now it is chosen instead of ambient.

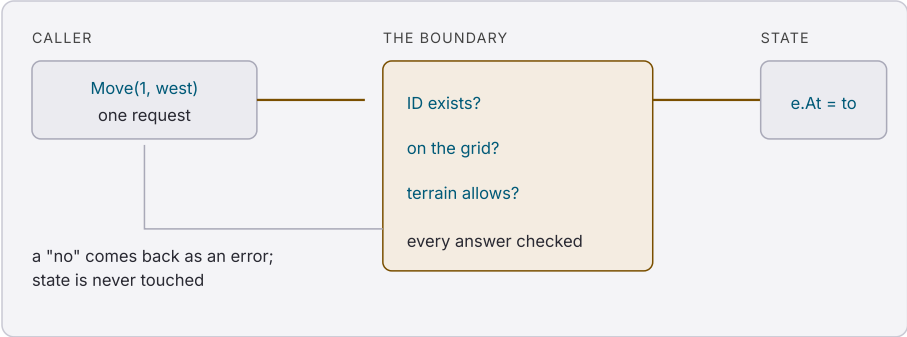



**Figure 6.1**: the sim's methods are a wall around state: a request either passes every check or bounces back as an error, and nothing half-happens.

## 5. Boundary checks

The design principle underneath the syntax: in a long-running simulation, a panic and an error have different blast radii. A panic unwinds the whole process; whatever else the server was doing dies with the one call that went wrong.

A returned error is scoped to the request that earned it. The walker that tried to leave the map lost its move; the pond kept being a pond. Servers that live for months are built out of operations that fail small.

The idiom's plain surface is exactly what makes it carry. Because an error is a value, failure handling is ordinary code: you can wrap it, log it, count it, decide on it with `errors.Is`, or hand it up to a caller with more context, all with the same tools you use on any other value.

Because the failure path is spelled out at every call site, reading a Go function shows you the unhappy path at the same time as the happy one. The `if err ≠ nil` blocks that look repetitive at chapter scale

become, at server scale, the audit trail: every place a request can die is visible, searchable, and decided on purpose.

The checks live inside `sim`, at the boundary, not sprinkled through callers. Nothing outside the package can reach `cells` or `ents` directly, so every route to state passes the same few methods, and those methods now refuse everything illegal.

That means the bounds check is law, not advice. Weather, creatures, villagers, and player packets all have to pass methods that have stopped trusting anyone.

## 6. Checkpoint

✓ CHECKPOINT: WHAT YOU CAN NOW DO

- Explain both faces of the unchecked bounds bug from the failure run: why row 0 panicked at slot −1, and why row 5 silently read `{11 4}` when asked for `{-1 5}`.
- Write a function returning `(T, error)`, and say what the non-error result is worth when the error is non-nil: nothing.
- Define a sentinel with `errors.New`, wrap it with `fmt.Errorf` and `%w`, and predict what `errors.Is` and plain `==` each answer against the wrapped result.
- Trace one refusal through three layers, grid to `allows` to `Move`, and say which piece of the final message each layer contributed.
- Defend the split `must` enforces in one sentence each: errors are for requests that may be declined, panics are for code that is wrong.

⚡ EXERCISES: TRY FIRST, THEN REVEAL

▼ **Exercise 1: remove, twice.** Write `Remove(id EntityID) error` on `*World`: delete the entity from the map, or refuse an ID that is not there. Spawn a walker, remove it, then

**remove it again and print the second error. Which sentinel should `errors.Is` find in it?**

The comma-ok lookup from `Get`, then the built-in `delete(w.ents, id)`. The second call prints something like `remove entity 1: no entity with that ID`, and `errors.Is(err, sim.ErrNoEntity)` reports `true`. The double-remove test matters more than it looks: once entities can die, two systems will occasionally try to reap the same one, and "already gone" must be an answer, not a crash.

▼ **Exercise 2: nothing roots in bedrock. Extend `allows` with a second rule: a `Shrub` may stand only on soil. Spawn one shrub on the valley floor and one on the rim, and print both results. How many functions did the new rule touch?**

One: `allows`. Add `if k == Shrub && t != Soil { return fmt.Errorf(..., ErrBlocked) }` and both `Spawn` and `Move` enforce it without changing, because they already ask `allows` for every placement. The rim attempt prints a refusal naming rock; the floor attempt returns a real ID. That is the payoff of routing every placement through one rule: the day terrain kinds multiply, legality still has one address.

▼ **Exercise 3: the comparison that lies. Capture the error from `w.Move(scout, sim.Coord{X: -1, Y: 5})` and print both `err == sim.ErrOutOfBounds` and `errors.Is(err, sim.ErrOutOfBounds)`. Predict the two booleans before running, then explain the difference in one sentence.**

`false`, then `true`. The value `Move` returned is the outermost wrapper, a different error value from the sentinel, so `==` compares two distinct things and says no; `errors.Is` unwraps layer by layer, finds the sentinel at the core, and

says yes. The one-sentence version: `==` asks "is this exactly that value?", while `errors.Is` asks "is that value anywhere in this chain?", and wrapped errors only ever answer the second honestly.

VOLUME 1 · CHAPTER 7

# The Fixed Tick

Why the world advances in equal steps instead of following the clock

## 1. Fixed ticks

Stop typing and the world does nothing. The rule that changes that is the volume's spine: **the world advances in whole, equal ticks and counts them; wall-clock time is only the pace at which those ticks are delivered, and no law inside the world may read it.**

The grid holds its terrain until something calls `Set`. Entities sit where they were spawned until something calls `Move`. Last chapter's hardened methods refuse bad requests, but every one of them waits for a caller, and so far that caller has been `main`, poking the world by hand.

A server does not get poked. It runs for months with nobody watching, and the world inside it has to keep going: water finds a low place, a creature burns energy, somebody's crop finishes at four in the morning.

The two loops most people write first are both wrong here. The first runs the world as fast as the processor allows, so a faster machine gets a faster world and a plant that took a minute to grow on your laptop takes eleven seconds on the server you deploy to.

The second measures how long the last pass around the loop took and scales every rate by that number. Game engines call it variable timestep, and it computes the world's motion from a measurement of your hardware's mood.

Both loops let the machine's speed leak into the world's physics. Sim time is a number the world owns; wall time is a fact about your machine. Keep them apart and one seed produces one world on a laptop, on a server, and at four hundred times speed.

## 2. Frame time and tick time

Before any of this touches `sim`, watch the leak in arithmetic small enough to check by hand. One walker moves east at three cells per second.

Written the variable-timestep way, each pass around the loop adds three cells times however many seconds that pass took: a machine drawing 144 frames a second adds 3 × 1/144, one drawing 48 adds 3 × 1/48.

Written the fixed way, every tick adds the same 0.3 cells, because the tick rate is a constant the program chose and not a property of the hardware.

▣ BUILD · STAGE 1: ONE WALKER, TWO WAYS OF COUNTING TIME

```
// cmd/worldd/main.go — a throwaway main: one walker, three
machines
package main

import "fmt"

// byFrame moves a walker east at 3 cells per second, once per
frame,
// on a machine that produces a frame every dt seconds.
func byFrame(name string, dt float64, frames int) {
        pos := 0.0
        for i := 0; i < frames; i++ {
                pos += 3.0 * dt
        }
        fmt.Printf("%-13s %3d frames of %.4fs → x = %.4f\n",
name, frames, dt, pos)
}

// byTick moves the same walker 0.3 cells per tick, ticks being
the
```

```
// only unit it knows about.
func byTick(name string, ticks int) {
        pos := 0.0
        for i := 0; i < ticks; i++ {
                pos += 0.3
        }
        fmt.Printf("%-13s %3d ticks            → x = %.4f\n",
name, ticks, pos)
}

func main() {
        fmt.Println("motion measured in frames:")
        byFrame("fast machine", 1.0/144, 100)
        byFrame("normal", 1.0/60, 100)
        byFrame("busy laptop", 1.0/48, 100)

        fmt.Println("motion measured in ticks:")
        byTick("fast machine", 100)
        byTick("normal", 100)
        byTick("busy laptop", 100)
}
```

```
$ go run ./cmd/worldd

motion measured in frames:
fast machine  100 frames of 0.0069s → x = 2.0833
normal        100 frames of 0.0167s → x = 5.0000
busy laptop   100 frames of 0.0208s → x = 6.2500
motion measured in ticks:
fast machine  100 ticks            → x = 30.0000
normal        100 ticks            → x = 30.0000
busy laptop   100 ticks            → x = 30.0000
```

The top three lines are three different worlds. Same code, same walker, same hundred passes around the loop, and the walker lands on column 2.08, column 5, or column 6.25 depending on nothing but who ran it.

Nobody wrote a bug; the loop was asked the wrong question. The damage also compounds: put a wall at column 6 and the busy laptop's walker is through it while the fast machine's walker is nowhere near, so the runs disagree about events and not merely positions.

> The bottom three lines are one world described three times. Ticks do not know what a second is, so there is nothing in them for hardware to change.

The frame-based version was *trying* to be honest, and smooth motion at a fixed real-world speed is the right trade for drawing a picture. It is the wrong trade for deciding what happened.

This book keeps both in their places: the simulation counts ticks, and a client can interpolate between them for the eye.

> **∑ INTERLUDE: CONVERTING BETWEEN TICKS AND SECONDS**
>
> Pick the rate first; everything else is division. This book runs at ten ticks per second of sim time, so thirty ticks is three seconds of world, a minute is 600 ticks, and a day is 60 × 60 × 24 × 10 = 864,000 ticks. Going the other way, a creature that should take two seconds to cross a cell takes 20 ticks.
>
> Written as a formula, with $r$ the tick rate, $t$ a count of ticks, and $s$ a duration in seconds of sim time:
>
> `s = t / r` and `t = s × r`
>
> Rates convert the same way: a speed of $v$ cells per second is $v / r$ cells per tick, so three cells a second at ten ticks a second is stage 1's 0.3. Do the conversion once, when you write the law, and store the per-tick number; a law that divides by the tick rate at run time is one edit away from dividing by a measured duration instead.
>
> Sim time is a `uint64`, holding about $1.8 \times 10^{19}$ ticks: at ten a second, roughly 58 billion years of world before it wraps. The counter will not be what fails.
>
> `r`

tick rate: ticks of sim time per second of sim time's own clock (10 in this book)
`t`
a count of ticks; the world's only measure of when
`s`
a duration in seconds of sim time, never of wall time
`v`
a speed written in cells per second, before conversion

## 3. Soil touching water

A counter that increments and changes nothing is impossible to debug, so the world gets its first law in the same breath as its first heartbeat.

Water spreads: any soil cell touching water becomes water next tick, the crudest model of a pond overflowing and enough to see.

The second law is smaller. Walkers drift one cell west each tick and stay put when the ground refuses them, which is last chapter's `ErrBlocked` being asked a question by something other than `main`.

▣ BUILD · STAGE 2: THE TICK RATE, THE COUNTER, AND STEP

```
// internal/sim/world.go — World gains one field and one reader
type World struct {
        Ground *Grid

        tick   uint64 // sim time: how many ticks this world has
taken
        ents   map[EntityID]*Entity
        nextID EntityID
}

// Tick is how many ticks of sim time this world has taken.
func (w *World) Tick() uint64 { return w.tick }


// internal/sim/tick.go
package sim
```

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

// TickRate is how many ticks of sim time one second of wall
time is
// meant to hold.
const TickRate = 10

// TickDuration is the wall-clock budget for one tick.
const TickDuration = time.Second / TickRate

// Step advances the world by exactly one tick.
func (w *World) Step() error {
        w.tick++
        if err := w.spreadWater(); err ≠ nil {
                return fmt.Errorf("tick %d: spread water: %w",
w.tick, err)
        }
        if err := w.driftWalkers(); err ≠ nil {
                return fmt.Errorf("tick %d: drift walkers: %w",
w.tick, err)
        }
        return nil
}

// driftWalkers steps every walker one cell west, in ID order. A
// walker the ground refuses stays where it is.
func (w *World) driftWalkers() error {
        for _, e := range w.Roster() {
                if e.Kind ≠ Walker {
                        continue
                }
                err := w.Move(e.ID, e.At.Offset(-1, 0))
                switch {
                case err == nil:
                case errors.Is(err, ErrBlocked), errors.Is(err,
ErrOutOfBounds):
                        // nowhere to go this tick; the walker
stays
                default:
                        return err
                }
        }
        return nil
}
```

Three decisions are packed into that small file. `TickRate` is a constant declared once, in the package that owns the world's

laws, so no caller can hand the simulation a different one and get a different history.

`TickDuration` is derived from it instead of typed out as `100 * time.Millisecond`. It is the only `time` value in the file, existing for the loop's benefit and not the world's.

`Step` increments the counter *first*, before any law runs, so a law reading `w.tick` sees the number of the tick it is computing. The world's first step is tick 1, and nothing that happens in it gets stamped with the 0 the world sat at before it began.

The field is lowercase and the reader is the method `Tick()`, for the reason chapter 5 made the entity map private: a counter anyone outside the package can assign to is a counter that can be made to disagree with the history it labels.

`driftWalkers` walks `Roster`, not the entity map, which cashes in chapter 5's lesson: map order is randomized, roster order is by ID.

With walkers that only drift west, the order looks harmless. When two entities want the same cell, order decides who gets it, and a world whose outcomes follow map iteration order disagrees with itself between runs.

The two tolerated error kinds are a decision too. An entity that cannot move is ordinary, so those are absorbed here; anything else is a bug and goes up to whoever runs the loop.

That leaves `spreadWater`, which is four lines of obvious code and one of the sharpest mistakes in simulation work. Write the obvious version first.

⚠ WORKED FAILURE: THE FLOOD THAT OUTRAN ITS OWN TICK

```
// internal/sim/tick.go — the obvious spread: scan, and flood as
you go
var neighbours = []Coord{{X: 1, Y: 0}, {X: -1, Y: 0}, {X: 0, Y:
1}, {X: 0, Y: -1}}

func (w *World) spreadWater() error {
        g := w.Ground
        for y := 0; y < g.H; y++ {
                for x := 0; x < g.W; x++ {
                        c := Coord{X: x, Y: y}
                        t, err := g.At(c)
                        if err ≠ nil {
                                return err
                        }
                        if t ≠ Soil {
                                continue
                        }
                        for _, d := range neighbours {
                                n, err := g.At(c.Offset(d.X,
d.Y))
                                if err ≠ nil {
                                        continue // off the grid
holds no water
                                }
                                if n == Water {
                                        if err := g.Set(c,
Water); err ≠ nil {
                                                return err
                                        }
                                        break
                                }
                        }
                }
        }
        return nil
}


// cmd/worldd/main.go — seed 5's valley, and exactly one tick
        w := sim.NewWorld(sim.Generate(12, 8, 5))
        fmt.Printf("tick %d\n%s", w.Tick(), w.Render())
        w.Step()
        fmt.Printf("tick %d\n%s", w.Tick(), w.Render())
```

```
$ go run ./cmd/worldd

tick 0
############
#..........#
#..........#
#.~~~~~....#
#.~~~~~~...#
```

```
#...~~~~...#
#...~......#
############
tick 1
############
#..........#
#.~~~~~~~~~#
#~~~~~~~~~~#
#~~~~~~~~~~#
#~~~~~~~~~~#
#~~~~~~~~~~#
############
```

One tick, and the pond ate the valley: not one cell of spread but everything the water could eventually reach, all at once.

The clue is the row that survived. Row 1, along the north rim, is still dry while row 2 and everything south and east of it is water, so the flood went south and east and refused to go north, and no line of code mentions a direction.

Directional bias with no direction in the code means the bias came from visiting order, and the only ordering here is the scan: row 0 first, then row 1, each row west to east.

Reaching row 2, the scan checked the cell below, found the pond in row 3, and flooded row 2 immediately. Continuing east, each cell it had just flooded was already water when its eastern neighbour was examined, so the flood raced ahead of the scan inside one pass.

Row 1 was scanned before any of that, when its southern neighbours were still soil, so it kept its ground.

Name the bug precisely, because the name is the fix: the tick had no width. Reading and writing one grid in a single pass lets a cell's new value become another cell's input in the same tick, so "next tick" and "later in this loop" collapse into one moment.

A tick has to be an instant every law sees identically, which means deciding from the state as it stood at the start and applying the changes afterward.

**▣ BUILD · STAGE 3: DECIDE FIRST, THEN FLOOD**

```
// internal/sim/tick.go — the fix: collect, then apply

// spreadWater floods every soil cell that touched water at the
start
// of this tick, and no cell that only touches water because of
it.
func (w *World) spreadWater() error {
        g := w.Ground
        flooding := make([]Coord, 0, 16)
        for y := 0; y < g.H; y++ {
                for x := 0; x < g.W; x++ {
                        c := Coord{X: x, Y: y}
                        t, err := g.At(c)
                        if err ≠ nil {
                                return err
                        }
                        if t ≠ Soil {
                                continue
                        }
                        for _, d := range neighbours {
                                n, err := g.At(c.Offset(d.X,
d.Y))
                                if err ≠ nil {
                                        continue // off the grid
holds no water
                                }
                                if n == Water {
                                        flooding =
append(flooding, c)
                                        break
                                }
                        }
                }
        }
        for _, c := range flooding {
                if err := g.Set(c, Water); err ≠ nil {
                        return err
                }
        }
        return nil
}


// cmd/worldd/main.go — one shrub, one walker, six ticks by hand
        w := sim.NewWorld(sim.Generate(12, 8, 5))
        w.Spawn(sim.Shrub, sim.Coord{X: 9, Y: 1})
        w.Spawn(sim.Walker, sim.Coord{X: 10, Y: 6})
        for i := 0; i < 6; i++ {
                fmt.Printf("tick %d\n%s", w.Tick(), w.Render())
                if err := w.Step(); err ≠ nil {
                        fmt.Println("step:", err)
```

```
                        return
                }
        }
        fmt.Printf("tick %d\n%s", w.Tick(), w.Render())
```

```
$ go run ./cmd/worldd

(ticks 0 through 3 of 6)

tick 0
#############
#........o.#
#..........#
#.~~~~~....#
#.~~~~~~...#
#...~~~~...#
#...~.....@#
#############
tick 1
#############
#........o.#
#.~~~~~....#
#~~~~~~~...#
#~~~~~~~~..#
#.~~~~~~~..#
#..~~~~~.@.#
#############
tick 2
#############
#.~~~~~..o.#
#~~~~~~~...#
#~~~~~~~~..#
#~~~~~~~~~.#
#~~~~~~~~~.#
#.~~~~~~~@.#
#############
tick 3
#############
#~~~~~~~.o.#
#~~~~~~~~..#
#~~~~~~~~~.#
#~~~~~~~~~~#
#~~~~~~~~~~#
#~~~~~~~~@.#
#############
```

Now the water moves like water: one cell in every direction, once per tick, the front advancing evenly out of the pond chapter 4's seed drew, north rim on the same schedule as south.

Two costs bought that. The scan visits every cell whether or not anything near it is wet, fine at 96 cells and ruinous at a million. The slice of pending changes is a second copy of part of the world, the standard price of simultaneity.

Watch the walker too, the first thing in this book stopped by an event and not by an argument. It starts at column 10 and drifts to column 9 on tick 1.

On tick 2 its step west would enter water that arrived this tick, `Move` returns `ErrBlocked`, `driftWalkers` absorbs it, and it stands at column 9 while the flood closes around it. Nothing coordinated the two. They met because they share a tick.

The shrub is the honest wart. It stands at tick 6 with water on every side, because nothing re-checks an entity's ground after that ground changes: `allows` runs when a request is made, and the shrub never made one.

That is a real gap, and it is the sort a world can only see once it has a tick to see things in.

## 4. The scheduled loop

`Step` is the whole simulation. What is left is delivery: calling it ten times a second, without `main` counting to six. The first attempt writes itself, and it is nearly right.

▣ BUILD · STAGE 4: SLEEP BETWEEN TICKS, AND WATCH THE DRIFT

```
// cmd/worldd/main.go — the sleeping loop, timed
// load stands in for the work later volumes will do inside a
tick.
const load = 30 * time.Millisecond

func main() {
        w := sim.NewWorld(sim.Generate(40, 20, 5))
        start := time.Now()
```

```
        for w.Tick() < 30 {
                if err := w.Step(); err ≠ nil {
                        fmt.Println("step:", err)
                        return
                }
                time.Sleep(load)
                time.Sleep(sim.TickDuration)
        }
        fmt.Printf("%d ticks at %d/s should take %.2fs\n",
w.Tick(), sim.TickRate,
                float64(w.Tick())/sim.TickRate)
        fmt.Printf("wall time actually spent: %.2fs\n",
time.Since(start).Seconds())
}
```

```
$ go run ./cmd/worldd

(timings measured on the author's machine; yours will differ)

30 ticks at 10/s should take 3.00s
wall time actually spent: 3.91s
```

Three seconds of sim time took 3.91 seconds to deliver. The loop sleeps a full tick *after* doing the tick's work, so every iteration costs the budget plus the work, and the error accumulates: thirty ticks, thirty helpings of 30 ms, nine tenths of a second behind.

That 30 ms is padding standing in for ecosystem, creature, and villager work inside `Step`. Run this for a day and the world is hours behind the wall clock, perfectly correct about its own history and useless as a place anyone can visit.

**⚙ TOOL · TIME.TICKER**

`time.NewTicker(d)` returns a ticker that delivers the current time on its `C` field every `d`, on a schedule fixed from the moment it was created, not from when you last looked. The expression `<-ticker.C` takes the next delivery, waiting if none is ready and returning immediately if one is already waiting, and `defer ticker.Stop()` releases it. That is every part of it this chapter

needs; the machinery underneath is chapter 9's subject. Full reference: `pkg.go.dev/time#Ticker`.

**▣ BUILD · STAGE 5: THE SAME THIRTY TICKS, ON A FIXED SCHEDULE**

```
// cmd/worldd/main.go — the loop on a ticker

// step does one tick's work: the world, then the padding.
func step(w *sim.World) {
        if err := w.Step(); err ≠ nil {
                panic(err)
        }
        time.Sleep(load)
}

func main() {
        w := sim.NewWorld(sim.Generate(40, 20, 5))

        ticker := time.NewTicker(sim.TickDuration)
        defer ticker.Stop()

        start := time.Now()
        for w.Tick() < 30 {
                ←ticker.C
                step(w)
        }
        fmt.Printf("%d ticks at %d/s should take %.2fs\n",
w.Tick(), sim.TickRate,
                float64(w.Tick())/sim.TickRate)
        fmt.Printf("wall time actually spent: %.2fs\n",
time.Since(start).Seconds())
}
```

```
$ go run ./cmd/worldd

(timings measured on the author's machine; yours will differ)

30 ticks at 10/s should take 3.00s
wall time actually spent: 3.03s
```

Same work, same padding, and the drift is gone: 3.03 seconds instead of 3.91. The difference is what each loop counts from.

The sleeping loop measures forward from whenever it happened to finish, so every delay it suffers is added to the next deadline permanently. The ticker's deadlines were laid down in advance,

one every 100 ms from the moment it was made, so early work waits and long work eats slack that was already allocated.

Nothing about the world changed, only its delivery.

One property matters more than the timing. If a tick's work badly overruns its budget, the ticker does not queue the missed beats and fire them in a burst; it drops them and delivers only the most recent.

The loop then runs slower in wall time and the world falls behind real people, a capacity problem you can measure and fix. Catching up instead, by running extra ticks, would call `Step` at a rate set by how overloaded the machine is: the leak this chapter closes.

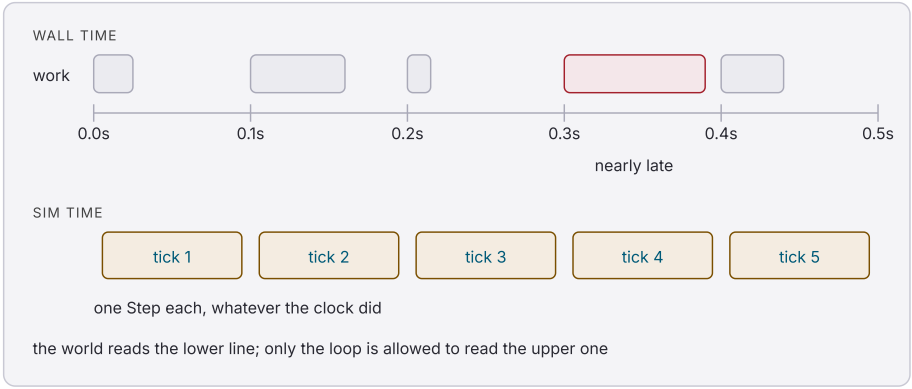



**Figure 7.1**: the loop absorbs every irregularity of real time so that the world above it sees a row of identical steps.

## 5. The tick number

Run one seeded world for a hundred ticks twice: paced at ten ticks a second with the 30 ms load, then unpaced, as fast as the processor will go. Record the water count after every tick and compare the two hundred-number traces.

▣ BUILD · STAGE 6: ONE HUNDRED TICKS AT TWO SPEEDS

```
// internal/sim/terrain.go — a number to compare runs by

// Count is how many cells of the grid hold terrain t.
func (g *Grid) Count(t Terrain) int {
        n := 0
        for _, c := range g.cells {
                if c == t {
                        n++
                }
        }
        return n
}


// cmd/worldd/main.go — one hundred ticks, twice, at two speeds
const (
        seed  = 5
        ticks = 100
        load  = 30 * time.Millisecond
)

// run advances a fresh world for n ticks, recording the water
count
// after every one. If paced, it runs at TickRate; otherwise it
runs
// as fast as the machine allows.
func run(n int, paced bool) ([]int, time.Duration) {
        w := sim.NewWorld(sim.Generate(40, 20, seed))
        trace := make([]int, 0, n)

        var ticker *time.Ticker
        if paced {
                ticker = time.NewTicker(sim.TickDuration)
                defer ticker.Stop()
        }

        start := time.Now()
        for w.Tick() < uint64(n) {
                if paced {
                        <-ticker.C
                        time.Sleep(load)
                }
                if err := w.Step(); err != nil {
                        panic(err)
                }
                trace = append(trace, w.Ground.Count(sim.Water))
        }
        return trace, time.Since(start)
}

func main() {
```

```
        fast, fastWall := run(ticks, false)
        slow, slowWall := run(ticks, true)

        fmt.Printf("unpaced: %d ticks in %.1fms of wall time\n",
len(fast),
                float64(fastWall.Microseconds())/1000)
        fmt.Printf("paced:   %d ticks in %.2fs of wall time\n",
len(slow),
                slowWall.Seconds())
        fmt.Println("water counts identical, all", ticks,
"ticks:", slices.Equal(fast, slow))
        fmt.Println("water at ticks 1, 5, 20, 100:",
                fast[0], fast[4], fast[19], fast[99])
}
```

```
$ go run ./cmd/worldd

(wall times measured on the author's machine; yours will
differ)

unpaced: 100 ticks in 0.4ms of wall time
paced:   100 ticks in 10.03s of wall time
water counts identical, all 100 ticks: true
water at ticks 1, 5, 20, 100: 33 137 625 684
```

Twenty-five thousand times the wall time, and not one number out of place. Both runs were 33 water cells into the flood at tick 1, both had filled all 684 cells of the valley floor by tick 100, and both passed through the same counts in the same order between.

The wall-clock figures are the only lines that change on your machine.

Fixing the timestep turns the tick into an *address*: every state the world has been in is reachable by a number, and that number means the same thing on every machine, at every speed, in every run from that seed.

It is what makes "tick 4,312" a phrase with a referent. A log line, a replay test, or a saved record only works if sim time is a count instead of a measurement.

The unpaced run is also the whole trick behind accelerated time: deliver ticks as fast as the machine can while nobody is looking, slow to real time when a player arrives, identical history either way. A world that ran a thousand years while you slept and one you watched for an hour differ only in how many ticks they took.

The tick rate never appears in `spreadWater`, which floods once per call and has no idea how often it is called. It does not appear in `driftWalkers` or in `Step`.

Every law is written per tick, and only the loop in `main` knows what a second is. Hold that line and the rate stays tunable: raise it to 20 for a finer-grained world, drop it to 5 to carry twice as much world per core, no law edited either way.

Cross it once and finding your way back means auditing every law you have written.

## 6. Checkpoint

**✓ CHECKPOINT: WHAT YOU CAN NOW DO**

- State what `Tick()` reports: steps the world has taken, never seconds anyone lived through.
- Say why stage 1's 144-frame and 48-frame runs put the walker in two different places, and why the tick-based runs cannot.
- Convert between the two units in both directions at a rate of 10: 30 ticks is 3 seconds, a two-second action is 20 ticks, a day is 864,000 ticks.
- Explain why writing into the grid during the spread scan flooded the valley in one tick, and why deciding first and applying afterward advances the front one cell.
- Say what a `time.Ticker` does that a `time.Sleep` loop does not, and why dropping a missed beat is the right behaviour for this program.

- Defend the boundary that keeps `time` out of `internal/sim` except as the loop's budget constant.

⚡ EXERCISES: TRY FIRST, THEN REVEAL

▼ **Exercise 1: the tick that dries out. Add a second law to `Step`: on every tenth tick, one water cell chosen by the scan's first hit reverts to soil. Print the water count each tick for 20 ticks and find the ticks where the count moves the wrong way.**

Guard it with `if w.tick%10 == 0`, which is true on ticks 10 and 20 because `Step` counted before it called any law, and pick the victim in the same decide-then-apply two passes `spreadWater` uses. The count still climbs, since one drying cell cannot outpace an advancing front, but ticks 10 and 20 land one cell short of where they would otherwise be: on the 40-by-20 valley, 322 and 624 instead of stage 6's 323 and 625. Run it twice and both traces lose their cell in the same place, which is only true because "every tenth tick" is counted, not timed.

▼ **Exercise 2: change the rate, keep the world. Set `TickRate` to 40 and run the paced loop for 30 ticks again. Predict both the wall time and the final water count before you run it, then explain which one you got wrong and why.**

The water count is unchanged, because no law reads the rate: 30 ticks of flooding is 30 ticks of flooding. The wall time is the interesting half. At 40 ticks a second the budget is 25 ms, and 30 ms of padding does not fit inside it, so the loop cannot keep the schedule and the run lands near 0.9 seconds instead of the 0.75 the arithmetic promises. That is stage 5's

capacity signal: the world stayed correct and the machine told you it is out of room.

▼ **Exercise 3: catch the leak in review. Somebody proposes `if time.Since(spawnedAt) > 2*time.Second { grow() }` inside a plant law. Write the two-sentence review comment that rejects it, and the replacement line.**

The replacement is a stored tick: record `plantedTick` when the plant appears, then `if w.tick-p.plantedTick ⩾ 20 { grow() }` at a rate of 10. The comment writes itself from this chapter: a law calling `time.Since` grows the plant after two seconds of the *operator's* time, so one seed gives different worlds on a loaded server, under accelerated time, and in chapter 10's replay test. Run both under stage 6's unpaced loop and only one still works, because a hundred unpaced ticks take under a millisecond and the wall-clock plant never grows.

VOLUME 1 · CHAPTER 8

# What the World Remembers

An append-only log of every event the tick loop produces

## 1. Append-only history

The loop from last chapter works, and that is the problem. The rule for memory is severe on purpose: **every event is written once, as one line, at the moment it happens, to a file that is only ever appended to.**

Ten times a second `Step` runs, walkers move and the ground changes under them; ten times a second the world overwrites what it just was. Print the map at the end of a run and you learn where everything stands right now.

Where a walker went, when the pond swallowed a square of soil, which of the two walkers reached the water first: gone, every one of them, overwritten by the tick that came after.

That is fine for a program you watch. It is fatal for a server meant to run for months. Ask anything about the past and the running process has no answer: when the pond last grew, which walker was standing where at tick 80, whether something changed the ground at 3 a.m. or it has always looked like this.

Debugging is the smallest of those needs. A world that keeps no record cannot be replayed, cannot be audited, cannot be told about, and cannot teach anything to anyone later, because teaching runs on what was written down.

No line is edited. No line is deleted. History grows at one end and is frozen everywhere else. The evidence is a twelve-second run that

accumulates two hundred lines of testimony, then yields to shell tools older than the format they are reading.

## 2. Spring and walkers

Start with what there is to remember, because a log is only as interesting as the world underneath it. The two laws written last chapter were placeholders chosen to exercise the tick, not to be lived in. Run them on the twelve-by-eight valley for 120 ticks and watch what they leave behind.

▣ BUILD · STAGE 1: LAWS WITH SOMETHING TO SAY

```
// cmd/worldd/main.go — last chapter's laws, unpaced, 120 ticks
        for w.Tick() < ticks {
                must(w.Step())
        }
        fmt.Print(w.Render())
        fmt.Println("ticks:", w.Tick(), " water cells:",
w.Ground.Count(sim.Water))
```

```
$ go run ./cmd/worldd

############
#~~~~~~~~~~#
@~~~~~~~~~~#
#~~~~~~~~~~#
#~~~~~~~~~~#
#~o~~~~~~~~#
#~~~~~~~~@~#
############
ticks: 120  water cells: 60
```

Sixty water cells is every square of the interior. The flood that copies itself into each neighbouring soil cell every tick reaches the far wall in six ticks, six tenths of a second, and the remaining 114 ticks have nothing left to flood. The walkers fare no better: one pressed west until the rim stopped it and has stood at the left-hand wall ever since, the other took a single step and hit water. A log of that run would be a burst of lines followed by two

minutes of silence. Replace both laws with ones that keep producing news.

```
// internal/sim/tick.go — the valley's two real laws

// SpringPeriod is how many ticks pass between one seep of the
spring
// and the next: every SpringPeriod ticks, one soil cell
touching
// water becomes water.
const SpringPeriod = 40

// steps are the four cells reachable from any cell in one move.
// The order is fixed, because a draw into it decides where a
walker
// goes and the world's history has to be reproducible.
var steps = []Coord{{X: 1, Y: 0}, {X: -1, Y: 0}, {X: 0, Y: 1},
{X: 0, Y: -1}}

// Step advances the world by exactly one tick.
func (w *World) Step() error {
        w.tick++
        if err := w.spring(); err != nil {
                return err
        }
        if err := w.wander(); err != nil {
                return err
        }
        return nil
}

// spring seeps once every SpringPeriod ticks: one soil cell
that
// touches water, chosen from the seeded stream, becomes water.
The
// candidates are collected in a fixed scan order so the draw
always
// picks from the same list in the same arrangement.
func (w *World) spring() error {
        if w.tick%SpringPeriod != 0 {
                return nil
        }
        g := w.Ground
        candidates := make([]Coord, 0, 16)
        for y := 0; y < g.H; y++ {
                for x := 0; x < g.W; x++ {
                        c := Coord{X: x, Y: y}
                        if t, _ := g.At(c); t != Soil {
                                continue
                        }
                        for _, d := range steps {
                                if n, err := g.At(c.Offset(d.X,
```

```
d.Y)); err == nil && n == Water {
                                        candidates =
append(candidates, c)
                                        break
                                }
                        }
                }
        }
        if len(candidates) == 0 {
                return nil
        }
        return w.Flood(candidates[w.rng.IntN(len(candidates))])
}

// wander steps every walker one cell in a direction drawn from
the
// world's stream, in ID order. A walker the ground refuses
stays
// where it is.
func (w *World) wander() error {
        for _, e := range w.Roster() {
                if e.Kind ≠ Walker {
                        continue
                }
                d := steps[w.rng.IntN(len(steps))]
                err := w.Move(e.ID, e.At.Offset(d.X, d.Y))
                switch {
                case err == nil:
                case errors.Is(err, ErrBlocked), errors.Is(err,
ErrOutOfBounds):
                        // nowhere to go this tick; the walker
stays
                default:
                        return err
                }
        }
        return nil
}


// internal/sim/world.go — the world takes a seed of its own
type World struct {
        Ground *Grid

        tick   uint64 // sim time: how many ticks this world has
taken
        rng    *rand.Rand
        ents   map[EntityID]*Entity
        nextID EntityID
}

// NewWorld wraps a terrain grid in a world with nothing
standing on
// it. The seed is the same number the grid was generated from;
```

```
the
// world takes its own draw stream from it, so terrain
generation and
// the laws never share a position in one sequence.
func NewWorld(g *Grid, seed uint64) *World {
        return &World{
                Ground: g,
                rng:    rand.New(rand.NewPCG(seed, 1)),
                ents:   make(map[EntityID]*Entity),
                nextID: 1,
        }
}
```

```
$ go run ./cmd/worldd

############
#..........#
#...~....@.#
#.~~~~~....#
#.~~~~~~...#
#.o~~~~~...#
#...~.~....#
#######@####
ticks: 120  water cells: 19
```

Sixteen cells of pond at the start, nineteen after twelve seconds: three seeps, one every forty ticks, each of them a soil square that happened to be touching water when the spring drew.

The walkers are somewhere unpredictable instead of pinned against a wall, and both laws can keep producing events for an hour. That is the property a log needs.

A world that finishes changing in six tenths of a second has no history to keep, and a valley whose creatures always go west has one you could write down from memory.

The important line is the signature. `NewWorld(g)` in chapter 5 took a grid and nothing else, because a world that only held things had no decisions to make. Motion does, so it is now `NewWorld(grid, seed)`, and the world keeps a generator built from that seed.

The stream number matters: the same seed the terrain came from, but fed to `rand.NewPCG(seed, 1)` where `Generate` used

`rand.NewPCG(seed, 0)`. PCG's second argument selects a stream, a different sequence of numbers from the same seed, so carving the pond can never consume a draw that a walker was going to make.

Both are decided by the seed. Neither can disturb the other.

Two details in `spring` exist for the same reason. The candidate list is built by scanning rows top to bottom and columns left to right, so the same set of eligible cells always arrives in the same arrangement, and drawing index 3 means the same square on every machine.

`wander` iterates `Roster()`, which sorts by ID, so walker 2 always draws before walker 3. Iterate the entity map instead, whose order Go randomizes on purpose, and the two walkers would swap draws between runs.

Determinism survives only where every list the code walks has a defined order. The seed does not grant it by itself.

## 3. A 70-byte move line

Before adopting a format, price it. A log is a file that grows every tick forever, so the cost per line multiplied by a year is a number you want to meet early instead of discovering it on a full disk. Pricing means writing one real line, so start with what an event is: a tick number, a word for what happened, and whichever details that kind of happening has. A spawn has an entity and a destination. A move has an entity, a departure, and an arrival. A terrain change has a square and a new kind of ground.

▣ BUILD · STAGE 2: ONE EVENT, MEASURED IN BYTES

```
// internal/sim/log.go
package sim
```

```
// Event is one thing that happened in the world, stamped with
the
// tick it happened on. Fields a given kind of event has nothing
to
// say about are left out of the line entirely.
type Event struct {
        Tick uint64   `json:"t"`
        Kind string   `json:"ev"`
        ID   EntityID `json:"id,omitempty"`
        From *Coord   `json:"from,omitempty"`
        To   *Coord   `json:"to,omitempty"`
        What string   `json:"what,omitempty"`
}

// ptr gives an event a coordinate it is also allowed to not
have.
func (c Coord) ptr() *Coord { return &c }


// internal/sim/coord.go — Coord gains two tags and no new
behaviour
type Coord struct {
        X int `json:"x"`
        Y int `json:"y"`
}


// cmd/worldd/main.go — a throwaway main, back for one stage
func main() {
        spawn := sim.Event{Tick: 0, Kind: "spawn", ID: 3,
                To: &sim.Coord{X: 10, Y: 6}, What: "walker"}
        move := sim.Event{Tick: 17, Kind: "move", ID: 3,
                From: &sim.Coord{X: 10, Y: 6}, To: &sim.Coord{X:
10, Y: 5}}
        flood := sim.Event{Tick: 40, Kind: "terrain",
                To: &sim.Coord{X: 6, Y: 4}, What: "water"}

        for _, e := range []sim.Event{spawn, move, flood} {
                line, err := json.Marshal(e)
                if err ≠ nil {
                        panic(err)
                }
                fmt.Printf("%s   (%d bytes)\n", line,
len(line)+1)
        }
}
```

```
$ go run ./cmd/worldd

{"t":0,"ev":"spawn","id":3,"to":{"x":10,"y":6},"what":"walker"}
(64 bytes)
{"t":17,"ev":"move","id":3,"from":{"x":10,"y":6},"to":
{"x":10,"y":5}}   (70 bytes)
```

```
{"t":40,"ev":"terrain","to":{"x":6,"y":4},"what":"water"}   (58
bytes)
```

`json.Marshal` takes any value and returns the bytes of its JSON encoding, reaching struct fields by reflection, so only exported fields appear. A lowercase field is invisible to the encoder exactly as it is invisible to other packages.

The backtick strings after each field are *struct tags*, metadata the compiler stores and hands to whoever asks. `encoding/json` asks, and obeys two instructions here: the name (`t`, `ev`) replaces the Go field name in the output, and `omitempty` drops a field whose value is the zero one.

That is why the terrain line carries no `"id"` and the spawn line carries no `"from"`: an event only pays for the facts it has.

`From` and `To` are `*Coord`, pointers, and the reason is `omitempty` itself. It considers a struct value never empty, so a plain `Coord` field would print `{"x":0,"y":0}` on every event that has no such coordinate.

That would put the top-left corner of the map in lines that mean nothing of the kind. A pointer has a genuine empty value, `nil`, so the field vanishes when there is nothing to say.

The `+1` in the byte count is the newline each line ends with. Call it 70 bytes for a move, the event kind that dominates the event log.

**∑ MATH INTERLUDE: A YEAR OF THE VALLEY, IN BYTES**

The tick rate is 10 per second. Suppose The Hollow eventually holds 200 walking creatures, each moving on about half of all ticks. Events per second is then 200 × 10 × 0.5 = 1,000, and at 70 bytes each that is 70,000 bytes a second. A day holds 86,400 seconds, so a day of history costs 70,000 × 86,400 = 6,048,000,000 bytes, near enough 6.05 GB, and a year of it about 2.2 TB. Large, but affordable on one disk, and that number

> is the reason the log records *events* and not snapshots. Writing the whole map every tick instead, for a 512-by-512 world at one byte per cell, costs 262,144 × 10 = 2.6 MB a second: 226 GB a day, thirty-seven times more, to store mostly cells that did not change. The event log's economy is that it says only what changed.
>
> `r` tick rate: how many ticks the world takes per second (10)
> `n` how many entities are acting
> `p` the fraction of ticks on which one entity does something (0.5 above)
> `b` bytes in one line, newline included (70 for a move)
> `r·n·p·b` bytes per second of history; multiply by 86,400 for a day

JSON is not the cheapest way to hold those facts. A move packed as binary fields, two coordinates and an ID, fits in about 20 bytes, so this format costs roughly three and a half times the minimum. What the extra bytes buy is that every tool on the machine can already read the file, including tools written decades before the format existed and tools nobody has written yet. For a log whose whole purpose is to be read later, by programs and people you cannot name today, that trade is the right way round.

The obvious container for many events is one JSON array holding all of them: `[{...},{...},{...}]`. It is a single valid document, which sounds tidy until a live server tries to use it. Appending an event means rewriting the closing bracket, so the file is only valid between writes; reading it means parsing every event ever recorded to see the last one; and a process that dies partway through a write leaves a document that no parser will accept.

JSON Lines fixes all three by giving up the outer document. One JSON object per line, no commas, no brackets around the whole thing, and the file is not one value but a stream of them. A crash costs at most the line being written. Reading the last event is `tail -1`. Two runs of history compare with `diff`, line by line, the same way two source files do. The format has no committee and barely has a specification, which is the point: any tool that understands lines

already understands half of it, and any JSON parser understands the other half.

**▣ BUILD · STAGE 3: THE APPEND-ONLY LOG**

```
// internal/sim/log.go — continued
import (
        "encoding/json"
        "fmt"
        "os"
)

// Log is the world's memory: one JSON object per line, appended
and
// never rewritten.
type Log struct {
        f   *os.File
        enc *json.Encoder
        n   uint64
}

// OpenLog opens the event log at path for appending, creating
it if
// this is the world's first run. An existing log is never
truncated.
func OpenLog(path string) (*Log, error) {
        f, err := os.OpenFile(path,
os.O_WRONLY|os.O_CREATE|os.O_APPEND, 0o644)
        if err ≠ nil {
                return nil, fmt.Errorf("open event log %s: %w",
path, err)
        }
        return &Log{f: f, enc: json.NewEncoder(f)}, nil
}

// Append writes one event as one line.
func (l *Log) Append(e Event) error {
        if err := l.enc.Encode(e); err ≠ nil {
                return fmt.Errorf("append event at tick %d: %w",
e.Tick, err)
        }
        l.n++
        return nil
}

// Count reports how many lines this Log has written.
func (l *Log) Count() uint64 { return l.n }

// Close releases the file.
func (l *Log) Close() error { return l.f.Close() }
```

```
// cmd/worldd/main.go — write three events by hand, twice
func main() {
        lg, err := sim.OpenLog("demo.jsonl")
        must(err)
        defer lg.Close()

        must(lg.Append(sim.Event{Tick: 0, Kind: "start", What:
"seed 5"}))
        must(lg.Append(sim.Event{Tick: 0, Kind: "spawn", ID: 1,
                To: &sim.Coord{X: 2, Y: 5}, What: "shrub"}))
        must(lg.Append(sim.Event{Tick: 4, Kind: "move", ID: 3,
                From: &sim.Coord{X: 10, Y: 6}, To: &sim.Coord{X:
10, Y: 5}}))

        fmt.Println("wrote", lg.Count(), "events")
}
```

```
$ go run ./cmd/worldd && cat demo.jsonl

wrote 3 events
{"t":0,"ev":"start","what":"seed 5"}
{"t":0,"ev":"spawn","id":1,"to":{"x":2,"y":5},"what":"shrub"}
{"t":4,"ev":"move","id":3,"from":{"x":10,"y":6},"to":
{"x":10,"y":5}}
```

```
$ go run ./cmd/worldd && wc -l < demo.jsonl

wrote 3 events
6
```

Two mechanisms carry this stage. `os.OpenFile` takes a set of flags combined with `|`, and the three here mean write only (`O_WRONLY`), create the file if it is absent (`O_CREATE`), and, the important one, `O_APPEND`: before every write the kernel moves the file offset to the current end of the file. Not to where this program last wrote. To the end, checked at write time, atomically with the write. Nothing this process does can put bytes anywhere but after everything already there, so "append-only" stops being a promise the code makes and becomes a property of how the file is open. The permission `0o644` is the usual owner-writes, everyone-reads.

`json.Encoder` is the streaming half of `encoding/json`: where `Marshal` hands you bytes, `Encode` writes them straight to an

`io.Writer`, and it appends a newline after each value. That newline is the whole reason this code has no line-joining logic. The format the world writes falls out of what the encoder already does, and every `Encode` is one write of one complete line to the file, with nothing held back in a buffer for later. The second run proves the flags: three more events, six lines total, and the first run's history still exactly where it was. The file has two lives in it now, and so each run opens with its own `start` line naming the seed. The world's history records the world's name.

**◆ NOTE: DO NOT WRAP THIS FILE IN A BUFFER**

The reflex when a program writes small pieces often is to wrap the file in a `bufio.Writer`, and here that reflex costs exactly what the format was chosen to protect. A buffer holds the last few kilobytes of history in memory and hands them to the kernel later; a process that dies in between loses every event it was told to remember since the last flush. The log is small (a few thousand bytes a second at valley scale) and the kernel is good at absorbing small writes, so this book pays for a write per event and keeps the guarantee.

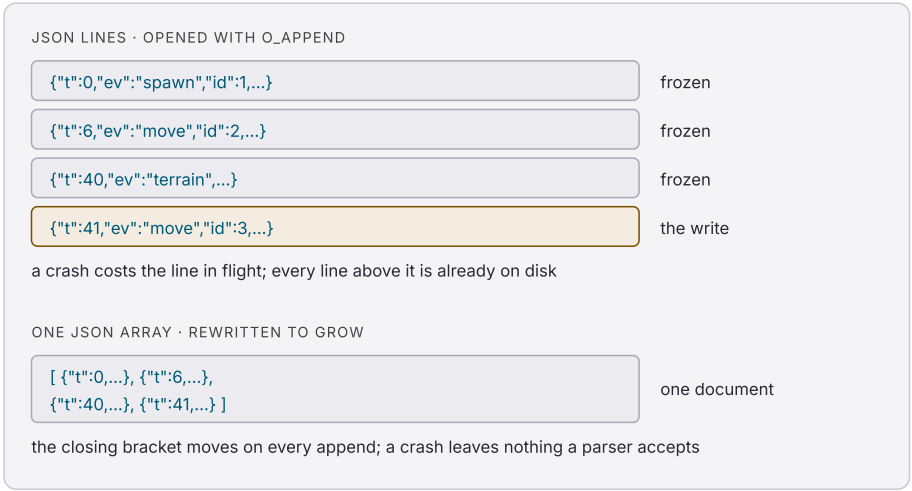


**Figure 8.1**: the same four events in both formats: a stack of finished lines, or one document that has to be reopened and reclosed to grow.

## 4. Recording events

Now put the log where the events are. Every fact the log records already passes through one of the hardened methods from chapter 6: an entity comes into the world through `Spawn`, changes position through `Move`, and the ground changes through `Flood`, which the spring is the only caller of. Those methods are the only routes to state, so they are the only places an event can be missed, and each gets one line added at the point where the change has definitely happened.

▣ BUILD · STAGE 4: THE WORLD RECORDS ITSELF

```
// internal/sim/world.go — the world learns to keep records
// Record hands the world a log to write its events to.
func (w *World) Record(l *Log) { w.log = l }

// record writes one event, remembering the first failure
instead of
// pretending nothing happened.
func (w *World) record(e Event) {
        if w.log == nil || w.logErr ≠ nil {
                return
        }
        if err := w.log.Append(e); err ≠ nil {
                w.logErr = err
        }
}

// Faulted returns the first error the world's log ran into.
func (w *World) Faulted() error { return w.logErr }


// internal/sim/world.go — three methods, one new line each
        // in Spawn, after the entity is in the map:
        w.record(Event{Tick: w.tick, Kind: "spawn", ID: id,
                To: at.ptr(), What: k.String()})

        // in Move, after e.At = to:
        w.record(Event{Tick: w.tick, Kind: "move", ID: id,
                From: from.ptr(), To: to.ptr()})
```

```
        // in Flood, after the ground is water:
        w.record(Event{Tick: w.tick, Kind: "terrain",
                To: c.ptr(), What: Water.String()})


// cmd/worldd/main.go — last chapter's loop, now with a memory
func main() {
        const seed = 5
        const ticks = 120

        lg, err := sim.OpenLog("world.jsonl")
        must(err)
        defer lg.Close()
        must(lg.Append(sim.Event{Kind: "start",
                What: fmt.Sprintf("seed %d", seed)}))

        w := sim.NewWorld(sim.Generate(12, 8, seed), seed)
        w.Record(lg)
        _, err = w.Spawn(sim.Shrub, sim.Coord{X: 2, Y: 5})
        must(err)
        _, err = w.Spawn(sim.Walker, sim.Coord{X: 10, Y: 6})
        must(err)
        _, err = w.Spawn(sim.Walker, sim.Coord{X: 2, Y: 2})
        must(err)

        fmt.Println("worldd", version, "seed", seed, "— logging
to world.jsonl")
        next := time.Now()
        for w.Tick() < ticks {
                must(w.Step())
                must(w.Faulted())
                next = next.Add(tickDur)
                time.Sleep(time.Until(next))
        }
        fmt.Print(w.Render())
        fmt.Println("ticks:", w.Tick(), " events written:",
lg.Count())
}
```

```
$ go run ./cmd/worldd

worldd 0.0.1 seed 5 — logging to world.jsonl
############
#..........#
#...~....@.#
#.~~~~~....#
#.~~~~~~...#
#.o~~~~~...#
#...~.~....#
#######@####
ticks: 120  events written: 210
```

Twelve seconds of world, 120 ticks, 210 lines. Read which events exist and which do not. Two walkers taking one step each per tick could produce 240 moves; the log holds 203, because a walker that draws a direction into water or off the map gets an error back from `Move` and nothing happens, and an event that did not happen gets no line. The log is a record of changes to the world, not of intentions about it. That distinction keeps the file honest and, at valley scale, keeps it a great deal smaller.

`record` is deliberately quiet in one direction and loud in another. It takes no error return, because a caller like `Move` has nothing useful to do about a failed write and its own error already means something specific to the caller. But it does not swallow the failure either: the first write error is kept in `logErr`, and the tick loop asks `Faulted` once per tick. A world whose disk has filled should stop, immediately and noisily, instead of cheerfully advancing another million ticks whose history nobody is keeping. A simulation that cannot remember what it is doing has no business continuing to do it.

**⚠ WORKED FAILURE: THE HISTORY THAT WAS STILL IN MEMORY**

The first version of `Log` was simpler and felt cleaner: collect the events in a slice, and write them all as one array when the world shuts down. One open, one write, one valid JSON document at the end.

```
// internal/sim/log.go — the version that lost a run
type Log struct {
        path   string
        events []Event
}

func OpenLog(path string) (*Log, error) { return &Log{path:
path}, nil }

func (l *Log) Append(e Event) error {
        l.events = append(l.events, e)
        return nil
}
```

```
func (l *Log) Close() error {
        blob, err := json.Marshal(l.events)
        if err ≠ nil {
                return fmt.Errorf("marshal history: %w", err)
        }
        return os.WriteFile(l.path, blob, 0o644)
}


// cmd/worldd/main.go — inside the tick loop, a power cut
modelled
        if w.Tick() == 7 {
                fmt.Println("*** the machine loses power at tick
7 ***")
                os.Exit(1)
        }
```

```
$ go run ./cmd/worldd; ls -l history.json

worldd 0.0.1 seed 5 — logging to history.json
*** the machine loses power at tick 7 ***
exit status 1
ls: cannot access 'history.json': No such file or directory
```

Not a truncated file. Not a corrupt file. No file. Seven ticks of world happened, sixteen events were dutifully handed to `Append`, and every one of them died in a slice when the process did. `os.Exit` was chosen to model the outage precisely because it skips deferred calls, which is what a power cut does to `defer lg.Close()` and to every other tidy-up a program had planned. The bug is not the array format by itself, but that the moment of writing was moved away from the moment of remembering, and everything in between existed only in a process that is not guaranteed to survive.

Put the appending version back, keep the same power cut, and the difference is the whole chapter:

```
$ go run ./cmd/worldd; wc -l < world.jsonl; tail -1
world.jsonl

worldd 0.0.1 seed 5 — logging to world.jsonl
*** the machine loses power at tick 7 ***
exit status 1
16
```

```
{"t":7,"ev":"move","id":3,"from":{"x":1,"y":3},"to":
{"x":0,"y":3}}
```

Sixteen lines survived a process that never got to shut down, and the last of them is the last thing that happened before the lights went out: walker 3, at tick 7, stepping west onto the rim. Nothing was flushed, because nothing was waiting. Each line was already on disk before the next event existed, and that is the property the append-only rule is really about: durability per event instead of per run.

## 5. Append-only storage

Append-only storage is a bargain in which you surrender one power to gain several. The power surrendered is editing: you cannot go back and correct line 400. Concurrency gets simpler, because writers contend for one place, the end. Reading gets simpler, because a line, once written, never changes under a reader. Crash recovery gets simpler, because the only damaged region possible is the tail. Correcting a mistake stays possible too, in the form every serious record-keeping system uses: a later entry that supersedes an earlier one, leaving both visible.

The hardware agrees with the design. A spinning disk writing sequentially never seeks, and an SSD's flash is erased in large blocks, so appending suits it far better than rewriting a middle. Database engines use the same file pattern under the name write-ahead log: record the change, durably and in order, before touching the structure it changes, and a crash becomes a question of replaying a suffix. What is being taught here is not a trick for a toy simulation. It is the shared foundation of filesystems, databases, and replicated systems.

There is one more reason, particular to this world. The file being written is not debugging output; it is the world's testimony about itself, the only account of The Hollow's past that will ever exist. Every creature that lives and dies and every square that floods passes through a line in a file like this one, timestamped by tick and never

altered afterward. History that can be edited is not history. Keep it append-only from the first walker and the record stays trustworthy for as long as the world runs.

## 6. Reading the log

A record nobody can read is a record in name only, so end where the format pays off. Nothing below is written for this file. `wc`, `grep`, `awk`, `sort` and `uniq` know nothing about worlds, ticks or JSON; they know lines. Because history is lines, they are enough to interrogate it.

▣ BUILD · STAGE 5: TWELVE SECONDS OF HISTORY, CROSS-EXAMINED

```
$ wc -l world.jsonl; ls -l world.jsonl | awk '{print $5,
"bytes"}'

210 world.jsonl
14268 bytes
```

```
$ awk -F'"ev":"' '{split($2, a, "\""); print a[1]}'
world.jsonl | sort | uniq -c

    203 move
      3 spawn
      1 start
      3 terrain
```

```
$ grep '"ev":"terrain"' world.jsonl

{"t":40,"ev":"terrain","to":{"x":4,"y":2},"what":"water"}
{"t":80,"ev":"terrain","to":{"x":6,"y":6},"what":"water"}
{"t":120,"ev":"terrain","to":{"x":3,"y":5},"what":"water"}
```

```
$ grep '"id":1,' world.jsonl

{"t":0,"ev":"spawn","id":1,"to":{"x":2,"y":5},"what":"shrub"}
```

The census by event kind is one `awk` line: split each line on the text `"ev":`, take what follows up to the next quote, then let `sort`

and `uniq -c` tally. 14,268 bytes for 210 events is 68 bytes a line on average, within a rounding error of the 70 the arithmetic predicted. The three terrain lines answer a question the final map could not: not just that the pond is bigger, but that it grew on ticks 40, 80 and 120, one square at a time, and in which order those three squares fell. Shrub 1 was spawned at {2 5} and never appears again, which is correct, since shrubs do not move.

The tick-80 flood is where this run keeps its surprise. Ask the file for the lines on either side of it:

```
$ grep -n -B1 -A1 '"ev":"terrain","to":{"x":6,"y":6}'
world.jsonl

136-{"t":79,"ev":"move","id":2,"from":{"x":7,"y":6},"to":
{"x":6,"y":6}}
137:{"t":80,"ev":"terrain","to":{"x":6,"y":6},"what":"water"}
138-{"t":80,"ev":"move","id":2,"from":{"x":6,"y":6},"to":
{"x":6,"y":7}}
```

Three consecutive lines, and they convict the simulation. On tick 79 walker 2 stepped onto {6 6}. On tick 80 the spring turned that exact square to water, with the walker standing on it. And then, later in the same tick, the walker moved off south under its own draw, so the run continued as if nothing had happened. `Move` refuses to put a walker *into* water; `Flood` never asks who is already there. A creature stood in the pond for one tick and the only reason it got out is that its direction that tick happened to point away. The final map cannot show any of this: by the time it is printed the walker is three squares away, and the cell is ordinary water. A printed map shows a state. A history lets you prosecute it.

**✓ CHECKPOINT: WHAT YOU CAN NOW DO**

- Write a law that draws from the world's own seeded stream, and say why `NewWorld` now takes a seed when chapter 5's version did not.

- Name the two orderings that keep a seeded law reproducible: the spring's row-by-row candidate scan and `Roster()`'s sort by ID.
- Define an `Event` struct whose JSON keys and omitted fields are controlled by struct tags, and explain why `From` and `To` have to be pointers for `omitempty` to work on them.
- Price a log format before adopting it: bytes per line times events per second times 86,400, with the day-sized answer for a 200-creature valley.
- Open a file with `O_WRONLY|O_CREATE|O_APPEND`, say what the kernel does at each write, and explain what the array-at-shutdown logger lost when the power went out at tick 7.
- Catch a bug in the simulation using `grep` alone, by reading two adjacent lines of a finished run that describe the same square one tick apart.

⚡ EXERCISES: TRY FIRST, THEN REVEAL

▼ **Exercise 1: the cell that flooded underfoot. The log caught the spring turning a square to water while a walker stood on it, and the file says so only because two lines happened to sit next to each other. Make it explicit: have `Flood` look for entities standing on the square it is about to drown and write a line naming each one. Run 120 ticks and grep for the new kind.**

In `Flood`, after the terrain line is recorded, walk `Roster()` for an entity whose `At` equals the flooded coordinate (struct equality works, since `Coord` is two comparable ints) and record `Event{Tick: w.tick, Kind: "stranded", ID: e.ID, To: c.ptr(), What: e.Kind.String()}`. Recording the terrain change first and the stranding second matters: anyone reading the file forward then sees the cause before the consequence. With seed 5 the run grows from 210 lines to 211, and the new one is `{"t":80,"ev":"stranded","id":2,"to":{"x":6,"y":6},"what":"walker"}`. Nothing else in the

history moves, because the exercise adds a line and changes no decision. One stranding in 120 ticks is also a measurement: rare enough to be easy to miss by watching, which is what the log is for.

▼ **Exercise 2: measure a day. Raise `ticks` to 1,200, run the two minutes, and use the resulting file to predict what a full day of this world costs. How close does the prediction get to the interlude's method?**

The two-minute run writes 1,768 events and 122,516 bytes, which is 102.1 bytes per tick. At 10 ticks a second that is about 1,021 bytes a second, so a day of this three-entity world costs roughly 88 MB and a year about 32 GB, for one spring and two walkers. Rerun it and the byte count is identical, since the run is deterministic from the seed: the same 1,768 events in the same order. The interlude's estimate assumed 200 creatures moving half the time and got 6 GB a day, so the two answers differ by the population, not by the method. The lesson to carry: measure a real run and multiply, instead of trusting an estimate you have not weighed on a scale.

▼ **Exercise 3: forge a line. With the server stopped, append a plausible but false event to `world.jsonl` by hand: `echo '{"t":60,"ev":"move","id":9,"to":{"x":1,"y":1}}' >> world.jsonl`. Nothing objects. What does that tell you about what "append-only" is protecting, and what it is not?**

`grep '"id":9'` finds your forgery sitting in the record as if it belonged there, and no part of the program noticed. The `O_APPEND` flag constrains one process's writes; it does not make the file immutable to its owner. Append-only, at this stage, is a property of how the world writes its own history, which is what protects it from the world's own bugs and crashes. That covers the failures this book is currently up

against. Protecting a record from someone with write access is a different problem with different tools, from filesystem permissions and immutable-file flags to a hash of each line folded into the next, and none of them are needed until somebody untrusted can reach the file. Delete the forged line before continuing, because the next thing this world does with its history is take it seriously.

VOLUME 1 · CHAPTER 9

# Running Two at Once

A tick loop on its own goroutine, stopped through a channel

## 1. Goroutine tick loop

worldd does one thing, finishes, and exits, which is a fine description of a batch job and a poor one for a server. The rule for making it a server is narrow: **the tick loop runs on its own goroutine, and main reaches it only through a channel.**

Two chapters ago the server grew a heartbeat, and one chapter ago it grew a memory: a fixed tick advancing the world in equal steps, and one JSON line appended to `world.jsonl` the moment anything happened.

Both of them ended up in the same place. The loop sits inside `main`, counting up to a fixed number of ticks, and while it runs there is no other code in this program.

A world that never resets cannot be a batch job. It has to keep ticking while *something else* is going on: drawing a frame, accepting a request, or noticing a Ctrl-C.

Chapter 8 already made the log durable, so the orderly-stop rule is not about rescuing lost bytes. An orderly stop means the world's last recorded event belongs to a tick that finished.

The log file is closed by the code that was writing to it, not abandoned by a dying process. When `main` prints its final report, the numbers in it describe a world that has actually stopped moving.

Not a shared variable. Not a flag one side writes and the other side reads. A channel, for a reason the end of this chapter proves with a tool instead of asserting.

## 2. The go keyword

Go spells concurrent work with a single keyword. Write `go f(x)` and Go calls `f` with `x` without waiting for it to return. The argument is evaluated right away, in the goroutine doing the calling; the body runs somewhere else, whenever the runtime gets to it. The statement produces no value at all, because by the time there could be one the caller has moved on to its next line.

A goroutine is not an operating-system thread. The Go runtime keeps a small pool of threads and multiplexes goroutines onto them, and each goroutine starts with a stack of a few kilobytes that grows when it needs to, so starting one costs closer to a function call than to a thread. A server can hold thousands. Today the world needs exactly one.

▣ BUILD · STAGE 1: THE LOOP MOVES OUT OF MAIN'S WAY

```
// cmd/worldd/main.go — chapter 8's setup, then chapter 7's loop
with
// one new word in front of it
        w := sim.NewWorld(sim.Generate(12, 8, seed), seed)
        w.Record(lg)
        // ... the shrub and two walkers spawn here, as they did
last chapter ...

        go func() {
                ticker := time.NewTicker(sim.TickDuration)
                defer ticker.Stop()
                for range ticker.C {
                        must(w.Step())
                        must(w.Faulted())
                }
        }()

        fmt.Println("worldd", version, "seed", seed, "ticking")
}
```

```
$ go run ./cmd/worldd

worldd 0.0.1 seed 5 ticking
```

```
$ cat world.jsonl

{"t":0,"ev":"start","what":"seed 5"}
{"t":0,"ev":"spawn","id":1,"to":{"x":2,"y":5},"what":"shrub"}
{"t":0,"ev":"spawn","id":2,"to":{"x":10,"y":6},"what":"walker"}
{"t":0,"ev":"spawn","id":3,"to":{"x":2,"y":2},"what":"walker"}
```

Four lines, every one of them stamped tick 0, and not a single move. The program came back to the prompt instantly. Read it the way the runtime did: `go` handed the loop to the scheduler and returned, `main` printed its banner and reached its closing brace, and when `main` returns the process exits. Every other goroutine stops where it stands, mid-instruction, with no deferred call run and nothing finished. `ticker.Stop()` never happened. The first tick was a hundred milliseconds away and never arrived at all.

The bill for concurrency arrives with the very first goroutine. `main` needs a reason to stay alive while the world ticks, a way to tell the loop to stop, and proof that the loop finished stopping. One mechanism covers the bill.

## 3. make, close and select

A channel is a typed pipe between goroutines, built with `make(chan T)`, and one rule does most of the work: a receive blocks until a value is there, and a send on an unbuffered channel blocks until somebody receives it. Blocking sounds like a cost and is really the coordination. A goroutine parked on a receive burns no CPU, holds no lock, and wakes the moment a value lands.

Closing supplies the second property. `close(ch)` announces that no value will ever be sent on it again, and from then on every receive returns instantly, forever, with the zero value of the channel's type.

One close reaches every receiver and none of them have to be counted, so a shutdown channel is normally closed and never sent on. The third piece is `select`, which waits on several channels at once and runs the branch belonging to whichever becomes ready first; if two are ready together it picks between them at random. Those three ideas are the whole concurrency vocabulary this volume needs.

**▣ BUILD · STAGE 2: A LOOP THAT CAN BE TOLD TO STOP**

```
// internal/sim/loop.go
package sim

import "time"

// Run ticks the world at the fixed rate until stop is closed.
However
// this loop ends, it closes the event log on its way out.
func Run(w *World, lg *Log, stop <-chan struct{}) error {
        ticker := time.NewTicker(TickDuration)
        defer ticker.Stop()

        for {
                select {
                case <-stop:
                        return lg.Close()
                case <-ticker.C:
                        if err := w.Step(); err != nil {
                                lg.Close()
                                return err
                        }
                        if err := w.Faulted(); err != nil {
                                lg.Close()
                                return err
                        }
                }
        }
}


// cmd/worldd/main.go — main starts the world, then goes back to
its own job
        stop := make(chan struct{})
        done := make(chan error)

        go func() {
                done <- sim.Run(w, lg, stop)
        }()
```

```
        fmt.Println("worldd", version, "seed", seed, "ticking
at", sim.TickRate, "Hz; ^C to stop")

        sigs := make(chan os.Signal, 1)
        signal.Notify(sigs, os.Interrupt)
        ←sigs
        fmt.Println("\nstop requested")
        close(stop)
        must(←done)

        fmt.Print(w.Render())
        fmt.Println("ticks:", w.Tick(), " events written:",
lg.Count())
```

```
$ go run ./cmd/worldd

worldd 0.0.1 seed 5 ticking at 10 Hz; ^C to stop
^C
stop requested
#@##########
#..........#
#...~......#
#.~~~~~....#
#.~~~~~~.@.#
#.o.~~~~...#
#...~......#
############
ticks: 40  events written: 77
```

Read `Run` as ordinary sequential code, because that is all it is: a loop around a select, with no `go` anywhere inside it. The concurrency lives at the call site, and keeping the two apart pays twice. `Run` can be read and tested as a loop, and the decision to run the world beside `main` instead of inside it is one line you can point at.

Two of the three channels here belong to the world and each carries traffic in one direction. `stop` is a `chan struct{}`: the empty struct is a type with no data in it at all, so this channel never carries a payload, only the fact of having been closed. `done` carries an `error`, which turns the loop's last act into a report: closing the log either worked or it did not, and `main` is what decides. The third channel, `sigs`, is filled by the runtime. `signal.Notify` asks Go to deliver a Ctrl-C as a value on a channel instead of killing the process outright, and the buffer of

one means a signal arriving a moment before `main` reaches the receive is held instead of dropped.

One line moved rather than appeared: `defer lg.Close()` is gone from `main`, and `Run` closes the log on every path out of the loop. The move follows the resource rule: the goroutine that writes to a resource is the one that closes it, because it is the only one that knows when the last write happened. Your run will report a different tick number than this one, since it counts tenths of a second between the banner and your hand leaving the key.

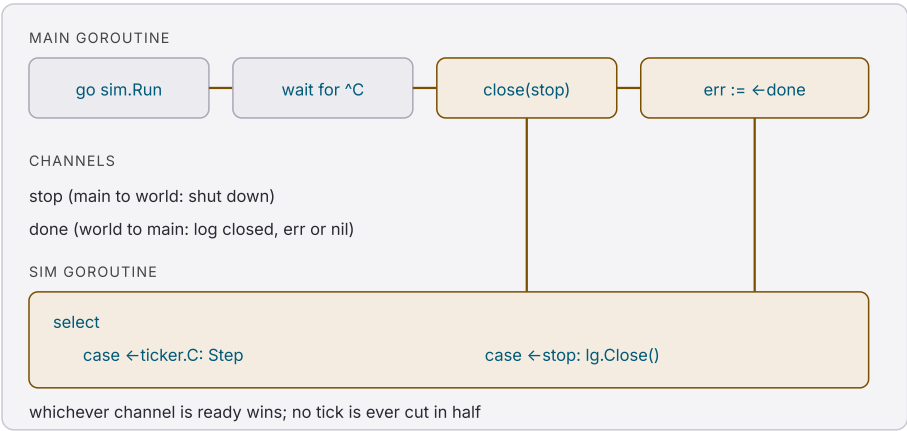



**Figure 9.1**: two goroutines, two channels, and one select deciding on every pass whether this is another tick or the end of the run.

## 4. 77 lines at tick 40

Those last two printed lines are a claim about a file, so check them against the file. `main` said the world reached tick 40 and the log accepted 77 events.

**▣ BUILD · STAGE 3: THE WORLD AND ITS RECORD, COMPARED**

```
$ wc -l world.jsonl
```

```
77 world.jsonl
```

```
$ tail -1 world.jsonl

{"t":40,"ev":"move","id":3,"from":{"x":0,"y":0},"to":
{"x":1,"y":0}}
```

Seventy-seven lines for seventy-seven events, and the file's last line carries tick 40, the same tick `main` reported. The run lands the same way every time: interrupt this world after any 40 ticks and you get these 77 lines, because the seed fixes what happens and only the tick count depends on your thumb. The loop finished a tick, saw the stop branch ready on its next pass, closed the file, and reported the result; only then did `main` read the world and print anything about it.

Trace the chain once: every link in it is a channel operation waiting on the link before. `close(stop)` makes the stop branch ready. The loop's next trip through `select` takes that branch, so it is never mid-tick when it leaves, and calls `lg.Close()`. `Run` returns whatever that reported. `done ← err` blocks until `main` receives, and `main` has been sitting at `←done` waiting for it. Only then does `main` render the map, read the counters, and return, ending the process. Nothing overlaps, and nothing had to be timed.

Delete the `←done` and the printed report becomes fiction of a specific kind: `main` would be reading terrain and entity positions out of a world that is still being stepped by another goroutine, and the map it drew might show a walker in a cell the log never recorded it entering. Exercise 3 makes the tool say so out loud. The shutdown signal alone was never the point; the acknowledgement is.

## 5. Channel ordering

A channel operation does two jobs, and the second one is invisible in the code. The obvious job is delivery: a value, or the news that a

channel is closed, moves from one goroutine to another. The quiet job is ordering. Go's memory model guarantees that everything a goroutine did before it sent on a channel is visible to the goroutine that receives, once the receive completes. The send and the matching receive are one event both goroutines agree on, and it cuts time into a before and an after.

That guarantee is the whole reason `main` can call `w.Render()` straight after `←done` with no lock in sight. The tick loop moved the walkers, appended their lines, closed the file, and then sent on `done`; `main` read the world after receiving. Same memory, two goroutines, no race, because a channel stands between the writing and the reading.

Take that event away and two goroutines touching the same variable is a data race, which is a worse thing than an occasional stale value. A compiler with no reason to expect another goroutine may load a variable into a register once and reuse it for the rest of a loop; processors reorder loads and stores for their own reasons; and Go's memory model declines to define what a racing program does at all. "It printed the right answer" is not evidence of correctness under those terms, only evidence that the luck held for one run.

**⚠ WORKED FAILURE: THE FLAG THAT WORKED FINE**

Here is the shutdown a channel looks like overkill for. A package-level boolean, written by `main` when the interrupt arrives, read by the loop at the top of every tick. It is shorter than the select, it needs no `Run` function, and on the bench it behaves.

```
// cmd/worldd/main.go — the stop a boolean seemed able to serve
var stopped bool

func main() {
        // ... the same log, the same world, the same three
entities ...

        done := make(chan error)
        go func() {
                ticker := time.NewTicker(sim.TickDuration)
```

```
                defer ticker.Stop()
                for range ticker.C {
                        if stopped {
                                done <- lg.Close()
                                return
                        }
                        must(w.Step())
                }
        }()

        // ... wait for Ctrl-C, then raise the flag ...
        <-sigs
        fmt.Println("\nstop requested")
        stopped = true
        must(<-done)

        fmt.Println("ticks:", w.Tick(), " events written:",
lg.Count())
}
```

```
$ go run -race ./cmd/worldd

worldd 0.0.1 seed 5 ticking at 10 Hz; ^C to stop
^C
stop requested
==================
WARNING: DATA RACE
Read at 0x0000006a8448 by goroutine 8:
  main.main.func1()
      /home/you/theworld/cmd/worldd/main.go:38 +0x130

Previous write at 0x0000006a8448 by main goroutine:
  main.main()
      /home/you/theworld/cmd/worldd/main.go:52 +0x7ec

Goroutine 8 (running) created at:
  main.main()
      /home/you/theworld/cmd/worldd/main.go:34 +0x524
==================
ticks: 38  events written: 72
Found 1 data race(s)
exit status 66
```

Look at what happened before the warning: the world stopped, the log closed, the numbers printed. By the only test available without a tool, the flag works. The detector disagrees, and its report is a list of facts. One address was read by goroutine 8 at line 38 and previously written by the main goroutine at line 52, and the two goroutines share no channel operation, no lock,

nothing that puts those two accesses in an order. That is a data race by definition, and exit status 66 is the program's own tooling refusing to call the run a success. Your addresses and line numbers will differ, and the two accesses can be reported in either order; the pairing of a write in `main` against a read inside the loop is what to look for.

Reasoning from symptom to cause runs backwards here, which is what makes races dangerous: the symptom was that everything looked correct. A boolean carries no ordering event, so the compiler is entitled to hoist that read out of the loop and the processor to publish the write late; neither did today, on this machine, with this build. Underneath the memory-model defect sits a design one. A flag can only be polled, so the loop notices it at the top of an iteration, up to a full tick late, and if the loop ever parks on something else it never notices at all. A closed channel works as a wakeup, not as a value to be checked, and the select is already waiting for it.

**⚑ TOOL: THE RACE DETECTOR**

`-race` works on `go run`, `go build` and `go test`. It instruments memory accesses and reports pairs it cannot put in an order, so it finds races that actually execute during that run, at roughly ten times the cost in time and memory. Run the sim under it whenever you add a goroutine, and never ship the instrumented binary. Details: go.dev/doc/articles/race_detector.

## 6. Checkpoint

**✓ CHECKPOINT: WHAT YOU CAN NOW DO**

- Say what `go func() { ... }()` starts and what it promises: work begins, nothing is waited for, and every goroutine dies the instant `main` returns, deferred calls and all.

- Explain why a shutdown channel is closed instead of written to, and what a receive on a closed channel hands back.
- Read the select inside `Run` and say what each branch does, including what happens when the ticker and the stop channel come ready together.
- Trace the stop chain from `close(stop)` to the log file being closed, and name which step forces the next one.
- Check an orderly stop against the evidence: the last line's tick against the tick `main` reported, and `wc -l` against the event count.
- Prove a shutdown boolean is broken while it appears to work, using `go run -race`, and read the detector's two-address report.

⚡ EXERCISES: TRY FIRST, THEN REVEAL

▼ **Exercise 1: forget to close. Comment out `close(stop)` in `main`, run worldd, and press Ctrl-C. Predict what you will see before you run it, then work out how to get your shell back.**

`stop requested` prints, and after that nothing ever happens again. The world keeps ticking, because its stop branch never becomes ready, and `main` is parked at `←done` waiting on a loop that has no reason to return. A second Ctrl-C does not help: `signal.Notify` took the interrupt away from the default handler and no goroutine is receiving from `sigs` any more. You end it from another terminal with `pkill -9 worldd`. Two goroutines waiting on each other is a deadlock even when one of them is busy, and this one is a single commented-out line.

▼ **Exercise 2: kill it the rude way. Start worldd, wait a few seconds, and run `pkill -9 worldd` from another terminal.**

**Count the lines in `world.jsonl`. Given how little is missing, say precisely what the stop channel is protecting.**

Nothing is missing: signal 9 cannot be caught or delayed, and the log still holds every event, because chapter 8 spends a write on each line as it happens. What the process loses is everything after that. The file is never closed by its writer, no report is printed, and the world stops wherever the kernel found it, so a tick that writes three lines can leave the log after the first. Today that window is microseconds wide. When a tick moves three thousand creatures it is a real fraction of a second, and a history ending halfway through a tick cannot be compared as a complete run.

▼ **Exercise 3: report on a moving world. Delete `must(<- done)`, keeping `close(stop)`, so `main` renders the map the instant it asks for a stop. Run it under the race detector.**

`go run -race ./cmd/worldd` puts the read inside `Render` against the write inside `Move`, several frames deep in the tick loop:

```
$ go run -race ./cmd/worldd

WARNING: DATA RACE
Read at 0x00c0000201f8 by main goroutine:
  theworld/internal/sim.(*World).Render()
      /home/you/theworld/internal/sim/world.go:110 +0x11b
  main.main()
      /home/you/theworld/cmd/worldd/main.go:44 +0x7b7

Previous write at 0x00c0000201f8 by goroutine 8:
  theworld/internal/sim.(*World).Move()
      /home/you/theworld/internal/sim/world.go:82 +0x290
  theworld/internal/sim.(*World).driftWalkers()
      /home/you/theworld/internal/sim/tick.go:75 +0x1ac
```

The trace continues down through `Step` and `Run` to the goroutine `main` started. Both fixes are one line, and choosing between them is a design decision. Put the receive back and the report is ordered, at the price of only being able to look

once the world has stopped. Or give `Run` a channel it sends snapshots on, so what `main` reads is a copy the loop chose to publish. A player asking what the valley looks like cannot be told to wait for shutdown, so a live view needs the second answer.

VOLUME 1 · CHAPTER 10

# Prove It Twice

Running the same seed twice and diffing the logs

## 1. Same seed, same bytes

Six hundred ticks now produces tens of thousands of bytes of history, and none of them are being checked. The rule is exact: **same seed, same tick count, same bytes: two runs of this world produce event logs that are equal byte for byte, and a single differing byte is a bug in the simulation, never noise.**

Back when the world was 96 cells of rock and water, proving that a seed determined it took one line: generate the grid twice, compare the two renderings with `==`, print `true`. That proof was honest and it is now badly out of date.

The world has entities with identities, a tick counter, laws that draw from a seeded stream every tick, moves that get refused by the ground, a spring that floods a cell every forty ticks, and a log recording all of it one line at a time.

The gap matters more than it looks. Suppose a walker ends up standing in water, and the log says it happened at tick 4,102 of seed 5. In a deterministic world you rerun seed 5, stop at tick 4,102, and stare at the same walker in the same puddle as often as you need.

Otherwise you have a story about something that happened once to somebody's process. No tolerance, no "mostly the same", no fields exempted for being unimportant. The test has to catch one differing byte.

## 2. The _test.go file

Go has testing built into the toolchain, so there is no library to choose and nothing to install. Put a file whose name ends in `_test.go` beside the code it tests; write functions named `TestSomething` taking one argument of type `*testing.T`; run `go test`. Those files are invisible to `go build`, so test code never ships inside `worldd`, and because this one declares `package sim` rather than an external test package it can reach unexported names in the package it is testing.

Start with chapter 4's generator claim, the smallest true thing this test file needs to defend. It fits in nine lines and its job here is to introduce the machinery.

▣ BUILD · STAGE 1: THE FIRST TEST FILE IN THIS BOOK

```
// internal/sim/gen_test.go
package sim

import "testing"

func TestGenerateRepeatsItself(t *testing.T) {
        first := Generate(12, 8, 5).Render()
        second := Generate(12, 8, 5).Render()

        if first ≠ second {
                t.Fatalf("seed 5 grew two different
worlds:\n%s\n%s", first, second)
        }
}
```

```
$ go test -v -run TestGenerate ./internal/sim/

(elapsed times measured on the author's machine; yours will
differ)

=== RUN   TestGenerateRepeatsItself
--- PASS: TestGenerateRepeatsItself (0.00s)
PASS
ok      theworld/internal/sim   0.002s
```

```
$ go test ./...
```

```
?       theworld/cmd/worldd     [no test files]
ok      theworld/internal/sim   0.010s
```

A test reports failure by calling a method on `t`, and never by returning a value or panicking. `t.Fatalf` formats a message, marks the test failed, and stops that test function there; its sibling `t.Errorf` marks the failure and keeps going, for when several independent checks each deserve reporting. Nothing prints on success unless you ask, which is what `-v` does. The message matters as much as the check: it is written for the version of you who reads it at midnight with no memory of this file, so it says what was compared and shows both sides.

The second command runs every package in the module. `cmd/worldd` has no tests, and Go says so plainly instead of pretending it passed. The tool is also aggressive about not repeating work: run `go test` twice without changing anything and the second run answers `ok theworld/internal/sim (cached)` in no time at all, because the result of a test whose code and inputs are unchanged cannot have changed either. That is a convenience most of the time and a trap when you are chasing something that only fails sometimes, so a chapter about nondeterminism keeps `-count=1` within reach.

**⚙ TOOL · THE FOUR GO TEST FLAGS THIS BOOK USES**

`-v` prints every test as it runs, along with anything `t.Logf` wrote. `-run PATTERN` selects test functions by regular expression, so `-run TestSameSeed` runs one of them. `-count=1` disables the result cache and forces a real run. `-count=N` runs the whole selection N times over, which is how you make an intermittent failure show itself. Full reference: `pkg.go.dev/cmd/go#hdr-Testing_flags`.

## 3. The run moves into sim

The generator was easy to test because it is a function you can call. The world is not: setting it up, spawning entities, opening a log, and running the loop all live in `main`, and `main` is the one function no test can call. That is not a testing quirk to work around. It is a design problem the test just exposed, and the fix improves the program on its own merits: everything about what a run *is* moves into the `sim` package behind one call, and `main` keeps what only a command-line program needs, which is flags, signals, and last chapter's goroutine and stop channel.

▣ BUILD · STAGE 2: ONE RUN, DESCRIBED BY A VALUE

```
// internal/sim/run.go
package sim

import (
        "fmt"
        "time"
)

// Config is everything one run of the world needs to know.
Every
// field is a decision the caller makes; nothing here is read
from the
// machine.
type Config struct {
        Seed  uint64 // names the world and every draw inside it
        Ticks uint64 // how many ticks to take before stopping
        Log   string // where the event log is written
        Paced bool   // deliver ticks at TickRate, or as fast as
they compute
}

// Run plays one world from beginning to end and writes its
history to
// cfg.Log. It returns when the world has taken cfg.Ticks ticks,
or
// when stop is closed, whichever happens first.
func Run(cfg Config, stop ←chan struct{}) error {
        lg, err := OpenLog(cfg.Log)
        if err ≠ nil {
                return err
        }
        defer lg.Close()
```

```
        if err := lg.Append(Event{Kind: "start", What:
fmt.Sprintf("seed %d", cfg.Seed)}); err ≠ nil {
                return err
        }

        w := NewWorld(Generate(12, 8, cfg.Seed), cfg.Seed)
        w.Record(lg)
        for _, s := range []struct {
                kind EntityKind
                at   Coord
        }{
                {Shrub, Coord{X: 2, Y: 5}},
                {Walker, Coord{X: 10, Y: 6}},
                {Walker, Coord{X: 2, Y: 2}},
        } {
                if _, err := w.Spawn(s.kind, s.at); err ≠ nil {
                        return err
                }
        }

        var beat ←chan time.Time
        if cfg.Paced {
                t := time.NewTicker(TickDuration)
                defer t.Stop()
                beat = t.C
        }

        reason := "ticks complete"
        for w.Tick() < cfg.Ticks {
                if beat ≠ nil {
                        ←beat
                }
                select {
                case ←stop:
                        reason = "stopped"
                default:
                }
                if reason == "stopped" {
                        break
                }
                if err := w.Step(); err ≠ nil {
                        return err
                }
                if err := w.Faulted(); err ≠ nil {
                        return err
                }
        }
        return lg.Append(Event{Tick: w.Tick(), Kind: "stop",
What: reason})
}
```

```
// cmd/worldd/main.go — flags, signals, and the goroutine
func main() {
        seed := flag.Uint64("seed", 5, "the number that names
this world")
        ticks := flag.Uint64("ticks", 600, "how many ticks to
take")
        logPath := flag.String("log", "world.jsonl", "where to
write the event log")
        fast := flag.Bool("fast", false, "deliver ticks as fast
as they compute")
        flag.Parse()

        cfg := sim.Config{Seed: *seed, Ticks: *ticks, Log:
*logPath, Paced: !*fast}
        fmt.Printf("worldd %s seed %d ticks %d → %s\n",
version, cfg.Seed, cfg.Ticks, cfg.Log)

        stop := make(chan struct{})
        sig := make(chan os.Signal, 1)
        signal.Notify(sig, os.Interrupt, syscall.SIGTERM)
        go func() {
                ←sig
                close(stop)
        }()

        done := make(chan error, 1)
        go func() { done ← sim.Run(cfg, stop) }()

        if err := ←done; err ≠ nil {
                fmt.Fprintln(os.Stderr, "worldd:", err)
                os.Exit(1)
        }
        fmt.Println("worldd: the world stopped cleanly")
}
```

```
$ go run ./cmd/worldd -fast -ticks 600 -log a.jsonl

worldd 0.0.1 seed 5 ticks 600 → a.jsonl
worldd: the world stopped cleanly
```

Two decisions in `Config` are deliberate. The seed and the tick count are inputs a caller supplies, so a run is fully described by a value you can print, store, and paste into a bug report. And `Paced` separates the delivery of ticks from their content, which is chapter 7's rule finally paying a dividend. Paced, the ticker hands out a tick every hundred milliseconds and 600 ticks take a minute; with `-fast` the same 600 ticks land as quickly as the processor can compute them, and the resulting world is identical

in every respect, because nothing inside the simulation has any way to ask how long a tick took. A minute of world, verifiable in milliseconds.

`Run` ends by writing a `stop` line naming why it stopped. Two runs that both finished their tick count agree on that line; a run cut short by a signal records the tick it actually reached, which is a real difference in what happened and belongs in the history.

Now the test can hold a whole world. It runs one, runs another, and compares every byte of the two files.

**▣ BUILD · STAGE 3: THE REPLAY TEST**

```
// internal/sim/replay_test.go
package sim

import (
        "bytes"
        "os"
        "path/filepath"
        "testing"
)

// history plays one world into a file of its own and returns
every
// byte the run wrote.
func history(t *testing.T, cfg Config) []byte {
        t.Helper()
        cfg.Log = filepath.Join(t.TempDir(), "world.jsonl")
        if err := Run(cfg, nil); err ≠ nil {
                t.Fatalf("run seed %d: %v", cfg.Seed, err)
        }
        b, err := os.ReadFile(cfg.Log)
        if err ≠ nil {
                t.Fatalf("read log: %v", err)
        }
        return b
}

// firstDiff reports the number of the first line on which two
// histories disagree, and both versions of it.
func firstDiff(a, b []byte) (int, string, string) {
        x := bytes.Split(a, []byte("\n"))
        y := bytes.Split(b, []byte("\n"))
        for i := 0; i < len(x) && i < len(y); i++ {
```

```
                if !bytes.Equal(x[i], y[i]) {
                        return i + 1, string(x[i]), string(y[i])
                }
        }
        return min(len(x), len(y)) + 1, "", ""
}

// body drops the start line, which names the seed and so is
expected
// to differ between two worlds.
func body(b []byte) []byte {
        _, rest, _ := bytes.Cut(b, []byte("\n"))
        return rest
}

func TestSameSeedSameHistory(t *testing.T) {
        cfg := Config{Seed: 5, Ticks: 600}

        first := history(t, cfg)
        second := history(t, cfg)

        if len(first) == 0 {
                t.Fatal("the run wrote no history at all")
        }
        if !bytes.Equal(first, second) {
                n, a, b := firstDiff(first, second)
                t.Fatalf("two runs of seed %d disagree at line
%d\n  run 1: %s\n  run 2: %s",
                        cfg.Seed, n, a, b)
        }
        t.Logf("seed %d: %d bytes of history, identical twice",
cfg.Seed, len(first))
}

func TestDifferentSeedsDifferentHistories(t *testing.T) {
        five := history(t, Config{Seed: 5, Ticks: 600})
        nine := history(t, Config{Seed: 9, Ticks: 600})

        if bytes.Equal(body(five), body(nine)) {
                t.Fatal("seeds 5 and 9 wrote the same history;
the seed is being ignored")
        }
        n, _, _ := firstDiff(body(five), body(nine))
        t.Logf("seeds 5 and 9 part company at event %d", n)
}
```

```
$ go test -v ./internal/sim/

(elapsed times measured on the author's machine; yours will
differ)

=== RUN   TestGenerateRepeatsItself
```

```
--- PASS: TestGenerateRepeatsItself (0.00s)
=== RUN   TestSameSeedSameHistory
    replay_test.go:59: seed 5: 65804 bytes of history,
identical twice
--- PASS: TestSameSeedSameHistory (0.00s)
=== RUN   TestDifferentSeedsDifferentHistories
    replay_test.go:70: seeds 5 and 9 part company at event 4
--- PASS: TestDifferentSeedsDifferentHistories (0.00s)
PASS
ok      theworld/internal/sim   0.010s
```

Sixty-five thousand bytes of world, written twice, with no byte out of place. Three small decisions hold that result up. `t.TempDir()` hands back a fresh directory per call, which the log's design demands: the file is opened for append, so two runs pointed at one path would stack their histories into one file and the test would compare something other than what it claims. `t.Helper()` marks `history` as plumbing, so a failure inside it is reported at the calling line in the test. And the second test stops the first from being a fraud: a `Run` that ignored its seed, or wrote nothing at all, would sail through an equality check. Something has to insist that different seeds still produce different worlds, and that they diverge at the fourth event, the first move either walker makes.

The test proves the property inside one process. The claim is bigger than that, so make it again the hard way, with two separate executions of a compiled binary. Delete the logs first, since an append-only file happily survives its own program.

**▣ BUILD · STAGE 4: TWO BINARIES, ONE HISTORY**

```
go build -o worldd ./cmd/worldd
rm -f a.jsonl b.jsonl
./worldd -fast -seed 5 -ticks 600 -log a.jsonl
./worldd -fast -seed 5 -ticks 600 -log b.jsonl
diff -u a.jsonl b.jsonl && echo "identical"
```

```
$ ./worldd -fast -seed 5 -ticks 600 -log a.jsonl; ./worldd -
fast -seed 5 -ticks 600 -log b.jsonl
```

```
worldd 0.0.1 seed 5 ticks 600 → a.jsonl
worldd: the world stopped cleanly
worldd 0.0.1 seed 5 ticks 600 → b.jsonl
worldd: the world stopped cleanly
```

```
$ diff -u a.jsonl b.jsonl && echo "identical"; wc -l a.jsonl
b.jsonl

identical
   953 a.jsonl
   953 b.jsonl
  1906 total
```

```
$ sha256sum a.jsonl b.jsonl

4cb81462cf8875fc307dceacba66f8e51fc7b8777866edaf3c271ea06c5d513
9  a.jsonl
4cb81462cf8875fc307dceacba66f8e51fc7b8777866edaf3c271ea06c5d513
9  b.jsonl
```

`diff` printing nothing is the whole result: 953 lines apiece and no line that differs, from two processes that shared no memory and ran at different moments on a machine doing other things in between. The checksums say it in one number each. Two tools, two definitions of "the same", one verdict.

## 4. Nondeterminism bug

A passing test proves nothing until you have watched it fail for the right reason. So break the world the way it actually gets broken, with a change that is faster, simpler, and correct-looking.

**⚠ WORKED FAILURE: THE TIDY OPTIMIZATION THAT ERASED TWO HUNDRED MOVES**

`wander` asks for `Roster()` every tick, and `Roster` allocates a fresh slice and fills it by walking IDs. Ten times a second, forever, for a list the world already holds in a map. Deleting the middleman is a one-word edit.

```
// internal/sim/tick.go — the "obvious" improvement
func (w *World) wander() error {
        for _, e := range w.ents { // was: for _, e := range
w.Roster()
                if e.Kind ≠ Walker {
                        continue
                }
                d := steps[w.rng.IntN(len(steps))]
                err := w.Move(e.ID, e.At.Offset(d.X, d.Y))
                // ... unchanged
        }
        return nil
}
```

Run the server and nothing looks wrong. It starts, it ticks, it stops cleanly, it writes a log full of plausible walkers taking plausible steps. Run the test:

```
$ go test ./internal/sim/

(a broken world fails differently every time; these two runs
are the author's)

--- FAIL: TestSameSeedSameHistory (0.00s)
    replay_test.go:56: two runs of seed 5 disagree at line 5
          run 1: {"t":1,"ev":"move","id":3,"from":
{"x":2,"y":2},"to":{"x":2,"y":1}}
          run 2: {"t":1,"ev":"move","id":2,"from":
{"x":10,"y":6},"to":{"x":10,"y":5}}
FAIL
FAIL    theworld/internal/sim   0.011s
FAIL
```

```
$ go test -count=1 ./internal/sim/

--- FAIL: TestSameSeedSameHistory (0.00s)
    replay_test.go:56: two runs of seed 5 disagree at line 13
          run 1: {"t":6,"ev":"move","id":2,"from":
{"x":9,"y":4},"to":{"x":10,"y":4}}
          run 2: {"t":6,"ev":"move","id":2,"from":
{"x":9,"y":4},"to":{"x":8,"y":4}}
FAIL
FAIL    theworld/internal/sim   0.011s
FAIL
```

Read the first failure. At tick 1 one run moved walker 3 and the other moved walker 2 first: the same two moves, in opposite

order. Go randomizes the starting point of every `range` over a map, on purpose, precisely so that nobody can come to depend on an order the language does not promise. The second failure shows what that costs once the order touches the seeded stream. Both runs move walker 2 out of {9 4} at tick 6, and one sends it east while the other sends it west, because whichever walker is served first takes the next draw from `w.rng`, and after one swap the two runs are drawing from the same sequence in different places forever. Neither failure will be the failure you get: a broken world lands somewhere new every time, which is the entire complaint.

The size of the damage is easier to see from the shell, comparing two runs of the broken binary. What follows is one such pair, captured once; yours will land elsewhere, and that is the point:

```
$ diff a.jsonl b.jsonl | head -4; wc -l a.jsonl b.jsonl

(one roll of a broken binary; no two are alike)

8d7
< {"t":3,"ev":"move","id":3,"from":{"x":1,"y":2},"to":
{"x":0,"y":2}}
9a9
> {"t":3,"ev":"move","id":3,"from":{"x":1,"y":2},"to":
{"x":0,"y":2}}
   977 a.jsonl
   957 b.jsonl
  1934 total
```

It opens as one line displaced by a single position and ends as two different worlds: 977 events against 957 on this roll, and of those, 815 lines of the first history appear nowhere in the second. The correct binary writes 953 lines for seed 5, every time; neither of these is that number, and running the pair again gives two more numbers that are neither. Look closely and the ground itself survives intact — the spring floods the same fifteen squares in both runs, because two walkers still draw twice per tick and the pond's own draws land in the same places. It is the walkers that come apart, and once they are on different cells the two logs are describing different afternoons. Nothing about that is visible in

> either run on its own, and no amount of reading the code would tell you which of the two is the real seed 5. Putting `Roster()` back fixes it, and the cost of the fix is one small allocation per tick, which is a bargain for the only property that makes any of this history worth keeping.
>
> The general lesson is bigger than maps. Any decision the world makes by consulting something the seed does not control (map order, wall-clock time, a goroutine that happened to finish first) leaks the machine's mood into the world's history. The replay test does not care which. It reports the first line where the world stopped agreeing with itself, and you go and find out why.

## 5. Determinism boundary

Determinism at this scale is not luck, and not something you sprinkle on later. It holds because every input to a decision inside the world passes through a boundary you drew, and only two things are allowed across: the seed and the tick count. Draws come from a generator built from the seed, in a fixed order. Entities are served in ID order, which the world assigns. Sim time comes from a counter the world increments. The laws consult none of the machine's ambient facts, because the code never asks for one. Same two inputs, same sequence of decisions, and the log records that sequence in the order it happened.

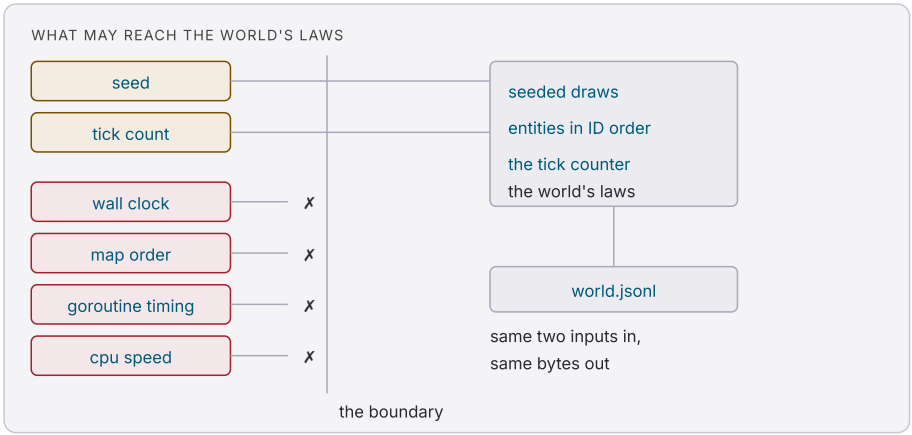



**Figure 10.1**: the replay test guards this boundary. Anything that gets across it on the left without being an input shows up on the right as a differing line.

That property is the contract every larger part of this book signs. A faithful snapshot can be checked by replaying the log from the beginning and comparing the rebuilt state, byte for byte, which only means anything if replaying is exact.

Server and client agreement uses the same property: replay the seed and the tick range on both sides, then compare what happened. A result you cannot reproduce is not a result, and "run it again with seed 5" is the difference between an experiment and an anecdote.

Determinism is not a feature you add once; it is a property one careless line can destroy at any point, and it destroys quietly: no crash, no error, just a world that answers the same question differently on Tuesday. The test runs before everything else because it is cheap, exact, and the witness that notices.

> ◆ **NOTE: WHY THE RUNS IN THIS BOOK ARE REPLAYED, NOT DESCRIBED**
>
> Every output printed in these pages comes from actually running the code in a container, and the ones from deterministic systems get checked byte for byte against what the page claims. That is the same machine you just built, pointed at the book instead of at

the world: a fixed seed, a run, and a comparison with no tolerance in it. Where a number cannot work that way, because a language model wrote it or because it measures your hardware, the page says so and tells you your run will differ.

## 6. Checkpoint

✓ CHECKPOINT: WHAT YOU CAN NOW DO

- Write a `_test.go` file in the same package as the code it tests, and say why test files never end up inside the `worldd` binary.
- Choose between `t.Fatalf` and `t.Errorf`, use `t.Helper()` so failures point at the calling line, and use `t.TempDir()` so an append-only log never carries over between runs.
- Restructure a program so a test can reach it, moving a run behind `sim.Run(cfg, stop)` and leaving flags and signals in `main`.
- Explain why `-fast` and the paced ticker produce identical histories, and use that to verify a minute of sim time in milliseconds.
- Prove byte-identical replay two ways: `go test` comparing two in-process runs, and `diff` plus `sha256sum` over the logs of two separate binaries.
- Recognize a nondeterminism bug from a diff that starts with two swapped lines and ends with two different worlds, and name the usual culprits.

⚡ EXERCISES: TRY FIRST, THEN REVEAL

▼ **Exercise 1: leak the clock on purpose. Add a `Wall string` field to `Event` with the tag `json:"wall,omitempty"`, and set it on the `start` event from `time.Now().Format(time.RFC3339Nano)`. Predict what the test says before running it.**

It fails on line 1, and the message hands you the culprit with no investigation required:

```
$ go test ./internal/sim/

--- FAIL: TestSameSeedSameHistory (0.00s)
    replay_test.go:56: two runs of seed 5 disagree at line
1
          run 1: {"t":0,"ev":"start","what":"seed
5","wall":"2026-08-24T17:29:29.887000744-05:00"}
          run 2: {"t":0,"ev":"start","what":"seed
5","wall":"2026-08-24T17:29:29.888931288-05:00"}
FAIL
```

Under two milliseconds apart, and the histories are already different files. The answer is not to forbid wall time but to keep it out of the event stream: an operator's question about when a run happened belongs in the operator's own logs, where nothing replays. Take the field back out before continuing.

▼ **Exercise 2: stop it by hand. Run `./worldd -seed 5 -ticks 600 -log c.jsonl` without `-fast`, press Ctrl-C after a couple of seconds, then look at the tail of the log. Is this a determinism bug?**

A run interrupted after two seconds ends like this, with the tick reached depending on exactly when your finger landed:

```
$ tail -2 c.jsonl; wc -l < c.jsonl

{"t":22,"ev":"move","id":3,"from":{"x":3,"y":1},"to":
{"x":4,"y":1}}
{"t":22,"ev":"stop","what":"stopped"}
43
```

Not a bug. The signal is an input from outside the world, as real as the seed, and the log records a run that stopped at tick 22 instead of 600. A bug would be the first 22 ticks disagreeing with the first 22 ticks of a full run, and they do not: `head -42` of the uninterrupted log matches this file

exactly. So the test fixes `Ticks` instead of stopping by signal, because a test controls its inputs and a keystroke is not one of them.

▼ **Exercise 3: five worlds instead of one. Write a table-driven test that replays seeds 5, 9, 10, 11 and 12 for 300 ticks each, using `t.Run` so every seed reports separately. Then run it with `-count=5`.**

`t.Run(name, func(t *testing.T){...})` starts a subtest with its own `t`, so one failing seed names itself instead of hiding inside a failure about "the seeds".

```
$ go test -v -run TestManySeeds ./internal/sim/

=== RUN   TestManySeedsReplay
=== RUN   TestManySeedsReplay/seed5
=== RUN   TestManySeedsReplay/seed9
=== RUN   TestManySeedsReplay/seed10
=== RUN   TestManySeedsReplay/seed11
=== RUN   TestManySeedsReplay/seed12
--- PASS: TestManySeedsReplay (0.01s)
    --- PASS: TestManySeedsReplay/seed5 (0.00s)
    --- PASS: TestManySeedsReplay/seed9 (0.00s)
    --- PASS: TestManySeedsReplay/seed10 (0.00s)
    --- PASS: TestManySeedsReplay/seed11 (0.00s)
    --- PASS: TestManySeedsReplay/seed12 (0.00s)
PASS
ok      theworld/internal/sim   0.012s
```

Add seeds 6, 7 and 8 to the table and the subtest naming pays for itself immediately:

```
$ go test -v -run TestManySeeds ./internal/sim/ 2>&1 |
grep -E "FAIL|seeds_test"

    seeds_test.go:13: run seed 6: spawn: walker at {2 2} is
water: terrain refuses the entity
    seeds_test.go:13: run seed 7: spawn: walker at {2 2} is
water: terrain refuses the entity
    seeds_test.go:13: run seed 8: spawn: walker at {10 6}
is water: terrain refuses the entity
--- FAIL: TestManySeedsReplay (0.01s)
    --- FAIL: TestManySeedsReplay/seed6 (0.00s)
```

```
    --- FAIL: TestManySeedsReplay/seed7 (0.00s)
    --- FAIL: TestManySeedsReplay/seed8 (0.00s)
FAIL
```

Not a replay failure at all, and not the same failure three times either. The line number is the giveaway: 13 is where the subtest calls `history`, and `history` is a `t.Helper()`, so a failure raised deep inside `Run` is reported against the line you would actually go and read. Seeds 6 and 7 put water on {2 2}, so it is the second walker `Run` spawns that the ground refuses; seed 8 floods {10 6} instead and stops the first one. Three of the eight seeds a reasonable person would try cannot even reach tick 1, because two starting positions are written into `Run` as literals and no terrain was ever asked to agree with them. A test that runs many worlds finds the ones your hard-coded starting positions were never designed for, which is a real defect in `Run` waiting for a real fix.

VOLUME 1 · CHAPTER 11

# A World in a Container

Building worldd's image and running it under podman

## 1. Code history and identity

`go run ./cmd/worldd` makes the valley exist for exactly as long as that terminal stays open and your hand stays off Ctrl-C. The hosted rule is a three-way split: **a world you can host keeps its code, its history and its identity in three separate places, and none of the three is a property of the machine it happens to be running on.**

Close the laptop and time stops. Whatever else this is, it is not yet something you can host.

The running world borrows a Go toolchain, of a version nobody checked at startup. It borrows the source tree, because `go run` compiles from source every single time.

It borrows your current directory, since `-log world.jsonl` means whatever directory you were standing in. It borrows your attention, because the program dies with the shell that launched it.

Code goes into a container image, built once and identical everywhere it lands. History goes into a volume, which outlives every process that writes to it. Identity, which here means the seed, arrives from the environment at startup.

Hold those three apart and you can start the world with one command, walk away, and read what it did through `podman logs`.

## 2. Environment defaults

Identity first, because it changes the program least. Last chapter gave worldd flags: `-seed`, `-ticks`, `-log`, `-fast`. Flags are the right interface for a person at a keyboard, and they keep working here, because a container run command can pass arguments straight through to the program inside. They are the wrong interface for everything else that starts this world, though. Compose files, unit files and orchestrators all configure a container the same way: by handing it environment variables. The fix is not to choose. Let the environment set the flag *defaults*, and let a flag on the command line beat the environment whenever somebody bothers to type one.

▣ BUILD · STAGE 1: DEFAULTS THAT COME FROM OUTSIDE THE BINARY

```
// cmd/worldd/main.go — above main

// defaults are the flag defaults, which the environment is
allowed to
// change before the command line gets its turn.
type defaults struct {
        seed   uint64
        ticks  uint64
        report uint64
        log    string
}

// envNum reads an unsigned number from the environment, or
returns
// def if the variable is unset or empty.
func envNum(name string, def uint64) (uint64, error) {
        raw, ok := os.LookupEnv(name)
        if !ok || raw == "" {
                return def, nil
        }
        n, err := strconv.ParseUint(raw, 10, 64)
        if err ≠ nil {
                return 0, fmt.Errorf("%s=%q: not a whole
number", name, raw)
        }
        return n, nil
}

// envStr reads a string from the environment, or returns def.
func envStr(name, def string) string {
        if raw, ok := os.LookupEnv(name); ok && raw ≠ "" {
```

```
                return raw
        }
        return def
}

// fromEnv collects the defaults worldd will start with,
refusing to
// start at all on a value it cannot read.
func fromEnv() (defaults, error) {
        d := defaults{log: envStr("WORLD_LOG", "world.jsonl")}
        var err error
        if d.seed, err = envNum("WORLD_SEED", 5); err ≠ nil {
                return d, err
        }
        if d.ticks, err = envNum("WORLD_TICKS", 600); err ≠ nil
{
                return d, err
        }
        d.report, err = envNum("WORLD_REPORT", 0)
        return d, err
}
```

`os.LookupEnv` returns the value and a second result saying whether the variable was set at all, and that second result is what makes defaults possible: an unset `WORLD_SEED` is a different situation from `WORLD_SEED=0`, and seed zero is a perfectly good world somebody might want. `strconv.ParseUint` turns text into a number and, more usefully, reports when it cannot. `fromEnv` passes that failure up instead of falling back to 5, because a world that ran seed 5 after you fat-fingered the seed would be the worst available outcome: you would stare at a valley and draw conclusions about a number that never ran.

The other new value in that struct is `report`, and it exists because of where this program runs. Detached inside a container, worldd has no terminal and nobody sitting in front of it. Its standard output becomes the only sign of life anybody outside can see, and a program that prints two lines in ten minutes is indistinguishable from a program that hung. So the world gets a heartbeat: every N ticks, one line saying what it looks like now.

**▣ BUILD · STAGE 2: FLAGS ON TOP OF THE ENVIRONMENT, AND A HEARTBEAT**

```
// cmd/worldd/main.go — the top of main, replacing last
chapter's flag block
        d, err := fromEnv()
        if err ≠ nil {
                fmt.Fprintln(os.Stderr, "worldd:", err)
                os.Exit(1)
        }

        seed := flag.Uint64("seed", d.seed, "the number that
names this world")
        ticks := flag.Uint64("ticks", d.ticks, "how many ticks
to take")
        logPath := flag.String("log", d.log, "where to write the
event log")
        report := flag.Uint64("report", d.report, "print a
status line every N ticks; 0 for silence")
        fast := flag.Bool("fast", false, "deliver ticks as fast
as they compute")
        flag.Parse()

        cfg := sim.Config{Seed: *seed, Ticks: *ticks, Log:
*logPath, Paced: !*fast}
        if *report > 0 {
                cfg.Watch = func(w *sim.World) {
                        if w.Tick()%*report == 0 {
                                fmt.Printf("tick %d: water %d
cells, %d entities\n",
                                        w.Tick(),
w.Ground.Count(sim.Water), w.Population())
                        }
                }
        }
```

```
$ WORLD_SEED=5 WORLD_TICKS=200 WORLD_REPORT=50
WORLD_LOG=/tmp/h.jsonl go run ./cmd/worldd -fast

worldd 0.0.1 seed 5 ticks 200 → /tmp/h.jsonl
tick 50: water 17 cells, 3 entities
tick 100: water 18 cells, 3 entities
tick 150: water 19 cells, 3 entities
tick 200: water 21 cells, 3 entities
worldd: the world stopped cleanly
```

```
$ WORLD_SEED=5 go run ./cmd/worldd -fast -seed 9 -ticks 20 -
log /tmp/i.jsonl

worldd 0.0.1 seed 9 ticks 20 → /tmp/i.jsonl
worldd: the world stopped cleanly
```

```
$ WORLD_TICKS=soon go run ./cmd/worldd

worldd: WORLD_TICKS="soon": not a whole number
exit status 1
```

Three runs show three behaviours. The first is configured entirely from the environment and it reads like any other invocation, because a flag default is still a flag default no matter where the number came from. The second sets `WORLD_SEED=5` and then asks for seed 9 on the command line, and 9 wins: `flag.Parse` overwrites the default with whatever the argument list says, so precedence falls out of the ordering rather than out of any code you had to write. The third refuses before the log is opened, before the grid is generated, before a single tick runs. A server that is going to be wrong should be wrong immediately.

`cfg.Watch` is the first function value in this book: `Config.Watch` is a field of type `func(*sim.World)`, which `sim.Run` calls after every successful tick when it is not nil. The closure assigned here captures `report`, a variable belonging to `main`, and keeps reading it long after `main` has moved on to waiting for a signal. So the simulation reports on itself while the `sim` package still contains no printing at all, exactly as chapter 2 promised. It does not print. It calls back to whoever asked to be told.

## 3. The container image

A container image is a filesystem plus a note saying which program to start inside it. You describe one in a `Containerfile`: a list of instructions, each taking the filesystem built so far and producing a new one. `FROM` picks the filesystem to begin with, `COPY` brings files in from your project, `RUN` executes a command inside the half-built image and keeps whatever it changed, `ENV` writes a default environment variable into the image, and `ENTRYPOINT` records the program to start when somebody runs a container from the finished result.

The obvious version begins `FROM` the official Go image, copies the source in, and builds. It works first time, and it ships a 900 MB image holding a complete Go compiler, the standard library source, a package cache and a Debian userland, in order to run a binary of about three megabytes. Everything in there is a thing that can have a security flaw, and none of it is needed once the build finishes. So the Containerfile describes two images and keeps only the second.

▣ BUILD · STAGE 3: THE CONTAINERFILE

```
# Containerfile — build worldd, then ship only worldd.

# Stage 1: the whole Go toolchain, used once and then discarded.
FROM docker.io/library/golang:1.26 AS builder
WORKDIR /src
COPY go.mod ./
COPY cmd ./cmd
COPY internal ./internal
RUN CGO_ENABLED=0 go build -o /out/worldd ./cmd/worldd

# Stage 2: the image that actually ships.
FROM docker.io/library/alpine:3.22
COPY --from=builder /out/worldd /usr/local/bin/worldd
ENV WORLD_LOG=/data/world.jsonl
ENTRYPOINT ["worldd"]
```

The second `FROM` is what makes this a multi-stage build: it discards everything above it and starts again from a bare Alpine, the same eight-megabyte image you ran back in chapter 1. The only thing that survives the discard is what `COPY --from=builder` explicitly carries across, which here is one file. `CGO_ENABLED=0` tells the toolchain to build with no C dependency at all, producing a binary that carries everything it needs and asks the host for no shared libraries; that is what lets a binary compiled against Debian's userland run unmodified on Alpine's very different one. `ENV WORLD_LOG` puts a default into the image itself, so the image knows where a world's history belongs and the person starting a container does not have to. And `ENTRYPOINT` in bracket form starts `worldd` as the container's first process with no shell in between, which matters a great deal when a signal arrives.

These `FROM` lines retain the original teaching tags. Both can move: `golang:1.26` can gain a compiler patch and `alpine:3.22` can gain updated packages. For an exact reference build, resolve both images with `podman image inspect`, record their `RepoDigests`, replace the tags with those digest references and select the reference platform. The original volume did not record those digests, so the build log below documents its capture rather than a permanent identity for either tag.

**▣ BUILD · STAGE 4: BUILD IT, THEN WEIGH WHAT CAME OUT**

```
podman build -t worldd:0.0.1 .
```

```
$ podman build -t worldd:0.0.1 .   (blob-copying lines cut;
the ids below are content hashes and yours will read
differently)

[1/2] STEP 1/6: FROM docker.io/library/golang:1.26 AS builder
[1/2] STEP 2/6: WORKDIR /src
⟶ c2ad00eb14b0
[1/2] STEP 3/6: COPY go.mod ./
⟶ fb066eb79825
[1/2] STEP 4/6: COPY cmd ./cmd
⟶ daade6316d7c
[1/2] STEP 5/6: COPY internal ./internal
⟶ fa12bc59a867
[1/2] STEP 6/6: RUN CGO_ENABLED=0 go build -o /out/worldd
./cmd/worldd
⟶ c15e3e443d21
[2/2] STEP 1/4: FROM docker.io/library/alpine:3.22
[2/2] STEP 2/4: COPY --from=builder /out/worldd
/usr/local/bin/worldd
⟶ fdec6d3d6206
[2/2] STEP 3/4: ENV WORLD_LOG=/data/world.jsonl
⟶ 84826267cd3a
[2/2] STEP 4/4: ENTRYPOINT ["worldd"]
[2/2] COMMIT worldd:0.0.1
⟶ adab9d6f6eae
Successfully tagged localhost/worldd:0.0.1
```

```
$ podman images --format "{{.Repository}}:{{.Tag}}  {{.Size}}"
localhost/worldd
```

```
localhost/worldd:0.0.1  11.7 MB
```

The step numbering shows the whole design: `[1/2]` for the builder's six instructions, `[2/2]` for the four that make the image you keep. Every line ending in a hexadecimal id is a layer, a saved snapshot of the filesystem after that instruction, and those ids are content hashes, so yours will read differently while meaning the same thing. The shipped image weighs 11.7 MB against the builder's 900: Alpine's 8.6 plus three megabytes of worldd, and no compiler at all. Layers are also a cache, which is why the two `COPY` lines are separate instead of one `COPY . .`. Change a file under `internal/` and podman re-runs from `COPY internal` downward; the steps above it, holding source you almost never edit, stay cached.

## 4. The podman volume

Start a container from that image and it gets a writable layer of its own: a scratch filesystem stacked on the image's read-only one, holding anything the process writes. It is genuinely writable, the log appends into it exactly as it appends to your disk, and it is deleted the moment the container is removed. For a world whose entire memory is one file, that arrangement is a trap.

A volume is podman's answer: storage that belongs to the machine instead of to any container, mounted into a container at a path you pick. Create one, mount it at `/data`, and the log the world writes lands somewhere no `podman rm` can reach.

▣ BUILD · STAGE 5: THE WORLD STARTS, AND YOU WALK AWAY

```
podman volume create valley
podman run -d --name worldd \
  -e WORLD_SEED=5 -e WORLD_TICKS=200 -e WORLD_REPORT=50 \
  -v valley:/data \
  worldd:0.0.1
```

```
$ podman volume create valley

valley
```

```
$ podman run -d --name worldd -e WORLD_SEED=5 -e
WORLD_TICKS=200 -e WORLD_REPORT=50 -v valley:/data
worldd:0.0.1

c4b835fe418c85c85670a7ca51f7858ac9d538e1730cdd7748c65dc75846940
c
```

```
$ podman ps --format "{{.Names}}  {{.Image}}  {{.Status}}"

worldd  localhost/worldd:0.0.1  Up Less than a second
```

`-d` is the chapter's whole argument in two characters: run detached. podman prints the container's id, hands your terminal straight back, and the world goes on ticking with no terminal attached to it. `--name` gives it something you can type instead of that id, each `-e` sets one environment variable inside the container, which is precisely where `fromEnv` looks, and `-v valley:/data` mounts the volume at the directory the image's own `ENV` already named.

Nothing is printing to your screen any more. The window into a detached container is `podman logs`, which replays everything the container's first process has written to standard output since it started. Run it twice, seconds apart, and watch the world accumulate.

**▣ BUILD · STAGE 6: READING A WORLD YOU ARE NOT ATTACHED TO**

```
podman logs worldd
podman ps -a --format "{{.Names}}  {{.Status}}"
```

```
$ podman logs worldd   (six seconds in)

worldd 0.0.1 seed 5 ticks 200 → /data/world.jsonl
tick 50: water 17 cells, 3 entities
```

```
$ podman logs worldd   (twenty-three seconds in)

worldd 0.0.1 seed 5 ticks 200 → /data/world.jsonl
tick 50: water 17 cells, 3 entities
tick 100: water 18 cells, 3 entities
tick 150: water 19 cells, 3 entities
tick 200: water 21 cells, 3 entities
worldd: the world stopped cleanly
```

```
$ podman ps -a --format "{{.Names}}  {{.Status}}"

worldd  Exited (0) 3 seconds ago
```

Two hundred paced ticks is twenty seconds of world, and the second reading catches it after the loop finished. The heartbeat lines are already earning their place: the pond went from 17 cells to 21 over those twenty seconds, and you know it without having watched a single tick. The last reading gains two squares instead of one because the spring seeps every forty ticks and tick 200 is itself a seeping tick, so that line reports the world one instant after its fifth seep. The exit status is podman reporting what the process returned, and 0 is the difference between a world that finished and a world that fell over. `podman logs -f worldd` follows the stream live, the way `tail -f` follows a file.

The container has exited, so remove it and see what is left behind. Your host cannot read the volume's files directly without help, but that is a small obstacle: mount the same volume into a throwaway Alpine container and use ordinary tools on it.

**▣ BUILD · STAGE 7: THE HISTORY OUTLIVES THE CONTAINER**

```
podman rm worldd
podman run --rm -v valley:/data docker.io/library/alpine:3.22 \
  sh -c 'wc -l /data/world.jsonl; tail -1 /data/world.jsonl'
```

```
$ podman rm worldd
```

```
worldd
```

```
$ podman run --rm -v valley:/data
docker.io/library/alpine:3.22 sh -c 'wc -l /data/world.jsonl;
tail -1 /data/world.jsonl'

322 /data/world.jsonl
{"t":200,"ev":"stop","what":"ticks complete"}
```

The process that wrote those lines is gone, and so is the container it ran inside. The 322 lines are not. The container was the world's body for twenty seconds; the volume is where the world's memory lives regardless of which body wrote it. Any container mounting `valley` can read that file, including one running a program with no connection to worldd, which is exactly what the Alpine container just was.

**⚠ WORKED FAILURE: THE CONVENIENCE THAT QUIETLY ATE A RUN**

Forget the `-v` flag with the image as written and worldd refuses to start: `open /data/world.jsonl: no such file or directory`, because `/data` only exists when something is mounted there. The obvious tidy-up is to stop the image from being fussy. Add one line to the Containerfile so the directory is always present:

```
RUN mkdir /data
ENV WORLD_LOG=/data/world.jsonl
```

Now it starts anywhere, mounted or not, and the error is gone. Run it twice without a volume, copying the log out of each container afterward, and look at what the fix bought:

```
$ podman run --name ghost -e WORLD_TICKS=200 worldd:mkdir -
fast; podman cp ghost:/data/world.jsonl /tmp/g1.jsonl; wc -l <
/tmp/g1.jsonl

worldd 0.0.1 seed 5 ticks 200 → /data/world.jsonl
```

```
worldd: the world stopped cleanly
322
```

```
$ podman rm ghost; podman run --name ghost -e WORLD_TICKS=200
worldd:mkdir -fast; podman cp ghost:/data/world.jsonl
/tmp/g2.jsonl; wc -l < /tmp/g2.jsonl

ghost
worldd 0.0.1 seed 5 ticks 200 → /data/world.jsonl
worldd: the world stopped cleanly
322
```

322, then 322. Against the volume those same two runs leave a file of 644 lines, because the second finds the first one's history waiting and appends to it. Reason from the symptom: the log is opened with `O_APPEND` and never truncates, so a count that fails to grow can only mean the second run opened a different file from the one the first wrote. Same path, different filesystem. Nothing in the Go code is wrong; twenty seconds of history went faithfully into storage that was scheduled for deletion, and `podman rm` carried out the sentence. The lesson is about which failure you would rather have. The unfixed image refuses to start when storage is missing; the `mkdir` makes it start happily and lose everything quietly. When state has nowhere durable to go, refusing to run is the useful behaviour.

One thing has to work before you can leave a world running for real: stopping it without hurting it. `podman stop` sends the container's first process a SIGTERM, the polite request to shut down, and waits before resorting to force. Last chapter's signal handler is already listening for exactly that. Give the container a tick count large enough that it will never reach it, and stop it by hand.

**▣ BUILD · STAGE 8: A WORLD WITH NO END IN SIGHT, ENDED ON PURPOSE**

```
podman run -d --name forever -e WORLD_TICKS=100000 -e
WORLD_REPORT=50 \
  -e WORLD_LOG=/data/forever.jsonl -v valley:/data worldd:0.0.1
```

```
podman stop forever
podman logs forever
```

```
$ podman stop forever; podman logs forever   (stopped by hand
after nine seconds; your run will get further or less far)

forever
worldd 0.0.1 seed 5 ticks 100000 → /data/forever.jsonl
tick 50: water 17 cells, 3 entities
worldd: the world stopped cleanly
```

```
$ podman run --rm -v valley:/data
docker.io/library/alpine:3.22 tail -1 /data/forever.jsonl

{"t":90,"ev":"stop","what":"stopped"}
```

The world stopped between two ticks, wrote its final line, and closed its log, because a signal from outside the container reached the goroutine holding the stop channel. Tick 90 is where the clock happened to be when the request landed, so your number will differ; that is the one figure on this page allowed to vary, and it varies honestly, since how long you waited before typing `stop` is no part of the simulation. The last line reads `"stopped"` instead of `"ticks complete"`, so a reader months from now can tell an operator's decision from a run that finished on its own. The `ENTRYPOINT` bracket form is what makes any of it work: written as a plain string, the container's first process would be a shell, the shell would swallow the SIGTERM, and podman would wait ten seconds before killing the world mid-tick with a line half written.

## 5. Four separate lifetimes

The split this chapter enforces is not a container convention that containers invented. It is a division by how often each part changes, and containers only make the division impossible to ignore.

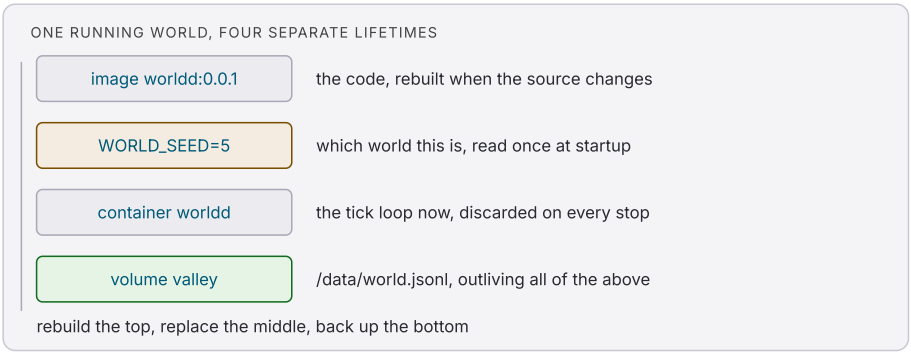



**Figure 11.1**: The four pieces of a hosted world, ordered by how long each is meant to last. A code change rebuilds the image. A restart replaces the container. The seed picks which world is running. The volume is the only part meant to be permanent, and the only part `podman rm` cannot touch.

Code changes when you write code, which often. Identity changes when you decide to run a different world, rarely and deliberately. History changes several times a second and must never be lost. Keep all three in one place and the fastest-moving part sets the terms for everything else: edit a file and you put the history at risk. Split them and each can be handled on its own schedule. Images get rebuilt and thrown away without a thought, since they hold nothing that cannot be rebuilt from source, and containers get destroyed and recreated just as freely. The volume you back up, because it is the one part of the running system nobody could reconstruct.

That last claim wants a test, not a promise, and you built the test in chapter 10: one seed produces one history, byte for byte. What is new is where the two runs happen. Run the world twice more from the same image, into two files of its own, and then let a third container be the judge.

**▣ BUILD · STAGE 9: THE SAME SEED, IN CONTAINERS THAT NEVER MET**

```
podman run --rm -e WORLD_SEED=5 -e WORLD_TICKS=200 -v
valley:/data \
  worldd:0.0.1 -fast -log /data/again.jsonl
podman run --rm -e WORLD_SEED=5 -e WORLD_TICKS=200 -v
```

```
valley:/data \
  worldd:0.0.1 -fast -log /data/again2.jsonl
podman run --rm -v valley:/data docker.io/library/alpine:3.22 \
  sh -c 'diff /data/again.jsonl /data/again2.jsonl; echo "diff
exit: $?"'
```

```
$ podman run --rm -e WORLD_SEED=5 -e WORLD_TICKS=200 -v
valley:/data worldd:0.0.1 -fast -log /data/again.jsonl

worldd 0.0.1 seed 5 ticks 200 → /data/again.jsonl
worldd: the world stopped cleanly
```

```
$ podman run --rm -v valley:/data
docker.io/library/alpine:3.22 sh -c 'diff /data/again.jsonl
/data/again2.jsonl; echo "diff exit: $?"; wc -l
/data/again.jsonl /data/again2.jsonl'

diff exit: 0
      322 /data/again.jsonl
      322 /data/again2.jsonl
      644 total
```

```
$ podman run --rm -v valley:/data
docker.io/library/alpine:3.22 sh -c 'diff /data/again.jsonl
/data/nine.jsonl | head -5'

--- /data/again.jsonl
+++ /data/nine.jsonl
@@ -1,322 +1,340 @@
-{"t":0,"ev":"start","what":"seed 5"}
+{"t":0,"ev":"start","what":"seed 9"}
```

A silent `diff` and an exit status of 0 mean the two files agree to the byte: two containers, started minutes apart, each generating its own terrain, spawning its own entities and taking its own two hundred ticks, wrote the same 322 lines in the same order. The flags arrive through the image command. Everything after the image name in a `podman run` command is appended to the image's `ENTRYPOINT`, so `-fast -log /data/again.jsonl` reaches `flag.Parse` as if you had typed it at a shell. A third run with `WORLD_SEED=9` settles the other half: same image, same command, one number different, and the histories part company on line 1 and stay parted for 340 lines. The seed is the whole of a

world's identity, and it now travels in an environment variable instead of a source file.

## 6. Checkpoint

What is in the box is small. Rock and water, three entities, a pond that swallows a square of soil now and then. What is true about it is not small. It advances on its own clock. It writes down everything that happens to it. It can be stopped and restarted without losing a line. Its entire past reproduces from one number. The box is small, but those four properties are already true of it.

**✓ CHECKPOINT: WHAT YOU CAN NOW DO**

- Make the environment supply flag defaults, explain why a flag typed on the command line still wins, and hand `sim` a `func(*sim.World)` so the simulation reports on itself without printing anything.
- Write a two-stage `Containerfile`, say what `COPY --from` carries across the discard, and explain what `CGO_ENABLED=0` buys when the builder is Debian and the runtime is Alpine.
- Read `podman build` output as a list of layers, predict which steps a given edit invalidates, and order `COPY` lines to keep the cache working.
- Start a detached container with a volume and an environment, then interrogate a world you are not attached to with `podman logs`, `podman ps -a` and an exit status.
- Explain the 322-then-322 result that exposes a missing volume, and argue why an image that refuses to start beats one that starts and loses the run.
- Say what `ENTRYPOINT ["worldd"]` makes possible when `podman stop` arrives, and what a shell in that position would have broken.

⚡ EXERCISES: TRY FIRST, THEN REVEAL

▼ **Exercise 1: two valleys at once. Create a second volume and start a second container from the same image with a different seed and a different name, both running at the same time. Read both logs. What does one image serving two live worlds tell you about what an image is?**

`podman volume create valley2`, then `podman run -d --name worldd2 -e WORLD_SEED=9 -e WORLD_TICKS=100000 -e WORLD_REPORT=50 -v valley2:/data worldd:0.0.1`, and `podman ps` shows both up with different heartbeats. Two worlds, two histories, one image and one copy of the binary on disk. An image is read-only and shared, so twenty worlds cost twenty writable layers and twenty processes, never twenty copies of worldd. That is the point of sharing an image: one binary can serve more than one live world.

▼ **Exercise 2: take Alpine out. Copy the Containerfile, change the second `FROM` to `FROM scratch`, build it as `worldd:scratch`, and run it against a volume. Measure the image. Then try `podman exec` into it and explain what comes back.**

It builds, runs and ticks identically: 3.08 MB against 11.7, because `scratch` is the absence of an image and a statically linked Go binary needs nothing else. The cost arrives the first time you want to look inside. `podman exec worldd ls /data` fails with `executable file 'ls' not found in $PATH`: no shell, nothing but your binary. That trade, the smallest possible attack surface against no way to poke around, is a real choice, and this book keeps Alpine because a world under construction is one you will want to open up.

▼ **Exercise 3: where the cache breaks. Rebuild with no changes and count the `Using cache` lines. Then add a comment to a file under `internal/sim/` and rebuild again. Which steps re-run, and which one surprises you?**

An unchanged rebuild caches every step, and so does a rebuild after merely touching a file, because `COPY` hashes contents and ignores timestamps. Adding a comment invalidates `COPY internal ./internal` and forces the `RUN go build` under it, exactly as you would predict. The surprise is the next line: `COPY --from=builder` still reports `Using cache`, because a comment changes no compiled instruction, the binary comes out byte-identical, and podman hashes the file being copied rather than the step that produced it. An identical binary makes an identical layer, so the image you ship is the image you already had.