inventory.go

```
func (inv *Inventory) Sell(
  sku string, qty int) error {
  inv.mu.Lock()
  defer inv.mu.Unlock()
  p, ok := inv.plants[sku]
  if !ok {
    return ErrNotFound
  }

// slices · interfaces · errors · generics
```

goroutines

channels

net/http

go test

```
$ go test -race ./... ok
$ sprig sell SPR-001 2
$ curl localhost:8080/plants/SPR-001
```

A COMPLETE, PRACTICAL GUIDE

Go Programming from Zero

Learn simple, fast, modern Go and build real tools, concurrent workers, and a web API

Go 1.25 · slices · interfaces · generics · iterators
goroutines · channels · testing · net/http · SQLite

LANGUAGE · CONCURRENCY · TOOLS · WEB

THENEXBOOKS

Go Programming from Zero

A Complete, Practical Guide to Simple, Fast, Modern Go

From your first program to concurrent tools and a JSON web API — for Go 1.25

First Edition

THENEXBOOKS

Go Programming from Zero
A Complete, Practical Guide to Simple, Fast, Modern Go

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Go and the libraries discussed in this book evolve; always consult the official documentation for the versions you use.

Go and the Go gopher are trademarks of Google LLC. Visual Studio Code is a trademark of Microsoft Corporation. GoLand is a trademark of JetBrains s.r.o. Docker is a trademark of Docker, Inc. SQLite is in the public domain; "SQLite" is a registered trademark of Hipp, Wyrick & Company, Inc. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by Google or the Go project.

Sprigwell Nursery, the business used as the running example in this book, is fictional, as are its people, suppliers, plants, and data. Domains such as example.com and fernline.example are reserved for examples.

Preface

Go was designed to make programming simple again. It has a small language you can hold in your head, a fast compiler, a standard library that covers most everyday needs, and a concurrency model that turns "do these ten things at once" from a research project into a few readable lines. Go programs compile to a single binary that you can copy to a server and run. That combination is why so much of today's cloud infrastructure, command-line tooling, and backend software is written in Go.

Simplicity does not mean there is nothing to learn. Slices share memory in ways that surprise newcomers. Interfaces are satisfied implicitly. Errors are ordinary values that you check and wrap. Goroutines are cheap, but they still leak, race, and deadlock when used carelessly. This book teaches all of it from the ground up, one idea at a time, and always explains *why* Go works the way it does.

We do it by building real software. Each chapter adds a piece to a small but complete system: a command-line inventory tool, a set of concurrent sensor and supplier workers, a SQLite-backed store, and a JSON web API. By the last page you will have written, tested, debugged, and shipped idiomatic Go — and you will know how to keep learning on your own.

Who This Book Is For

- **Beginners** who want to learn programming with a clean, modern language and are ready to practice.
- **Developers from other languages** such as Python, JavaScript, Java, C#, or C who want to become productive in Go quickly and write it the way experienced Go developers do.
- **Students and career changers** who want a portfolio project built with professional habits: tests, clear errors, and small packages.
- **Go users with gaps** who learned by copying examples and now want a solid understanding of slices, interfaces, errors, generics, iterators, and concurrency.

What You Will Learn

- How to install Go, set up an editor, create modules, and use the `go` command to run, build, format, vet, and test code.
- The language from the ground up: types, constants, control flow, functions, strings and runes, arrays, slices, maps, structs, methods, and pointers.
- Interfaces, type switches, idiomatic error handling with wrapping and custom error types, and packages and project layout.
- Generics, the standard library essentials, files and I/O, JSON and CSV, and command-line tools with the `flag` package.
- Concurrency: goroutines, WaitGroups, channels, `select`, mutexes, atomics, the race detector, `context`, worker pools, pipelines, and errgroup.

- Testing, benchmarks, fuzzing, closures, and range-over-func iterators.
- HTTP clients and servers with the enhanced `ServeMux`, databases with `database/sql` and SQLite, structured logging with `log/slog`, and debugging with Delve.
- Idiomatic style, performance basics with pprof, cross-compiling, and shipping binaries and containers.

How This Book Is Organized

Part I — Getting Started (Chapters 1–5) installs Go and covers the fundamentals: types, control flow, functions, and strings. **Part II — Core Language** (Chapters 6–10) covers slices and maps, structs and methods, interfaces, error handling, and packages. **Part III — Building Real Programs** (Chapters 11–15) covers generics, the standard library, files, data encoding, and command-line tools. **Part IV — Concurrency** (Chapters 16–20) covers goroutines, channels, synchronization, context, and concurrency patterns. **Part V — Practical Go** (Chapters 21–25) covers testing, iterators, HTTP, databases, logging, and debugging. **Part VI — Projects and Next Steps** (Chapters 26–28) assembles the complete CLI and web API and prepares you to ship and keep growing. The appendices provide a cheat sheet, a tooling reference, a compiler-error field guide, and a glossary.

Each chapter ends with practice exercises, selected solutions, and a **Key Takeaways** box for quick review.

The Running Example: Sprigwell Nursery

Throughout the book we write software for **Sprigwell Nursery**, a fictional independent plant nursery and garden shop with three greenhouses. Lena owns the business, Tomas runs the greenhouses, and Priya, who helps in the shop, is learning Go alongside you. It starts with a simple question — which plants are in stock? — and grows into `sprig`, a command-line inventory tool; a set of concurrent workers that watch greenhouse sensors and compare supplier prices; and `sprigd`, a small JSON web API backed by SQLite.

Versions and Conventions

Examples target **Go 1.25**. Most code also runs on Go 1.23 and later; where a feature needs a specific release, we say so. The language changes slowly and carefully, but tools and libraries move faster, so where a detail may change, a Note points you to the official documentation rather than guessing.

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, version differences, or a useful aside.

> **WARNING**
>
> A common mistake or something that can cause bugs, data loss, or security problems.

Code appears in `monospace`. Long lines are wrapped to fit the page. File names above code blocks, such as `sprigwell/inventory/inventory.go`, tell you where the code lives. Terminal commands work the same on Windows, macOS, and Linux unless noted.

Contents

Preface

PART I — GETTING STARTED

1 Why Go? Simplicity, Speed, and Who Go Is For

2 Installing Go, Modules, Editors, and Your First Program

3 Variables, Constants, Basic Types, and Operators

4 Control Flow and Functions: if, for, switch, defer, and Multiple Return Values

5 Strings, Runes, Bytes, and Formatting with fmt

PART II — CORE LANGUAGE

6 Arrays, Slices, and Maps

7 Structs, Methods, and Pointers

8 Interfaces, Type Assertions, and Type Switches

9 Error Handling: Errors as Values, Wrapping, Custom Errors, panic, and recover

10 Packages, Modules, and Project Layout

PART III — BUILDING REAL PROGRAMS

11 Generics: Type Parameters and Constraints

12 Standard Library Essentials: strings, strconv, slices, maps, sort, and time

13 Files and I/O: io.Reader, io.Writer, bufio, and the os Package

14 Encoding Data: JSON, CSV, and Struct Tags

15 Building Command-Line Tools with flag, os.Args, and Subcommands

PART IV — CONCURRENCY

16 Goroutines and sync.WaitGroup

17 Channels and select

18 Sharing Memory Safely: Mutexes, sync, atomic, and the Race Detector

19 The context Package: Cancellation, Timeouts, and Deadlines

20 Concurrency Patterns: Worker Pools, Pipelines, Fan-Out/Fan-In, and errgroup

PART V — PRACTICAL GO

21 Testing in Go: Table Tests, Subtests, Benchmarks, Fuzzing, and Examples

22 Closures, Function Values, and Iterators (range over func)

23 HTTP with net/http: Clients, Servers, and JSON APIs

24 Working with Databases: database/sql and SQLite

25 Logging with log/slog, Configuration, and Debugging with Delve

PART VI — PROJECTS AND NEXT STEPS

26 Project: The Sprigwell Inventory CLI End to End

27 Project: A Small JSON Web API for Sprigwell

28 Idiomatic Go, Performance Basics, Building and Shipping Binaries, and Where to Go Next

APPENDICES

A Go Syntax and Standard Library Cheat Sheet

B The go Command and Tooling Reference

C Compiler Errors and Common Gotchas Field Guide

D Glossary

PART I

Getting Started

Why Go exists and how to set it up, your first module and program, and the fundamentals every Go program is built from: variables and types, control flow, functions, strings, runes, and formatting.

1 · Why Go?
2 · Installing Go and Your First Program
3 · Variables, Constants, and Types
4 · Control Flow and Functions
5 · Strings, Runes, and fmt

CHAPTER 1 · PART I — GETTING STARTED

Why Go? Simplicity, Speed, and Who Go Is For

Before we type a single command, we want you to understand what kind of language Go is, why it was designed the way it was, and what you will be able to build by the end of this book. This chapter also introduces Sprigwell Nursery, the small business whose software we will write together.

A language designed on purpose

Most programming languages grow by accumulation. A feature is added because another language has it, then another feature is added to patch the interaction between the first two, and after a couple of decades the language is powerful but hard to hold in your head. Go took a different path. It was designed deliberately, by a small group of engineers, to solve a specific set of problems they were running into every day.

Go was created at Google, starting in 2007, by Robert Griesemer, Rob Pike, and Ken Thompson. It was released as an open source project in November 2009, and the first stable version, Go 1.0, arrived in March 2012. The designers were working on very large server programs. They watched builds take a long time, saw code bases where every team used a different subset of a complicated language, and noticed that the machines they deployed to had many processor cores that their languages made awkward to use. Go was their answer: a language that compiles quickly, reads clearly, scales to large teams, and makes concurrent programs natural to write.

That origin story matters to you as a learner because it explains almost every design choice you are about to meet. When something in Go feels unusually strict — the compiler refusing to build a program because of one unused variable, for example — the reason is nearly always the same: the designers chose to make the code easier to read and maintain, even if it costs the writer a few extra seconds. Go optimizes for the person who reads the program six months later, and very often that person is you.

Go's design goals

You will find these ideas repeated throughout the book. It helps to see them all in one place first.

Simplicity you can hold in your head

Go has only 25 keywords. The full language specification is a single web page that an experienced programmer can read in an afternoon. There is usually one obvious way to write a loop, one way to declare a type, and one way to handle an error. This does not mean Go programs are trivial. It means

the complexity lives in your problem, not in the language. When you open a Go file written by a stranger, you will rarely meet a construct you have never seen before.

Readability and one formatting style

Go ships with a formatter, `gofmt`, and nearly every Go project in the world uses it. Indentation, spacing, and alignment are decided by the tool, not by personal taste. The result is that arguments about brace placement simply do not happen, and all Go code looks familiar. You will run the formatter yourself in Chapter 2, and your editor will probably run it every time you save.

Fast compilation

Go compiles to native machine code, and it does so quickly. The language was shaped to make this possible: dependencies are declared explicitly with `import`, unused imports are errors, and there are no header files to parse again and again. For you, this means the edit-compile-run loop feels almost as quick as running a script. You change a line, run the program, and see the result a moment later.

Static types without the ceremony

Every value in Go has a type known at compile time, so a whole category of mistakes — adding a number to a piece of text, calling a function with the wrong arguments — is caught before the program ever runs. At the same time, Go lets you skip writing the type when it is obvious. The short declaration `stock := 8` creates a variable of type `int` without you having to say so. You get the safety of static types with much of the convenience of a dynamic language.

Garbage collection

Go manages memory for you. You create values, and when nothing refers to them any more the garbage collector reclaims the memory. You never call `free`, and you cannot accidentally use memory that was already released. Go still gives you pointers, which you will meet in Chapter 7, but they are safe pointers: there is no pointer arithmetic in everyday Go code.

Concurrency built into the language

Modern computers have many cores, and modern programs talk to networks, disks, and other services that make them wait. Go gives you two simple building blocks for doing several things at once: **goroutines**, which are very lightweight functions that run concurrently, and **channels**, which let goroutines pass values to each other safely. Starting a goroutine is as simple as writing the keyword `go` before a function call. Part IV of this book is devoted to using these tools well.

Composition instead of inheritance

Go has no classes and no inheritance. Instead, you define types (usually structs), attach methods to them, and describe behavior with **interfaces**. A type satisfies an interface simply by having the right methods; it never has to declare that it does. This keeps types small and loosely coupled. If you come from Java or C#, this will feel unusual for a week and liberating after that.

Errors are ordinary values

Go has no exceptions in the usual sense. A function that can fail returns an `error` value alongside its normal result, and the caller checks it. You will write `if err != nil` many times in this book. It looks repetitive at first, but it makes every failure path visible in the code, right where it happens. Chapter 9 covers errors in depth.

A strong standard library and single binaries

Go's standard library includes an HTTP client and server, JSON and CSV encoding, file and network I/O, cryptography, testing, structured logging, and much more. You can build real programs without downloading anything. When you build, Go produces a single executable file that contains your program and the parts of the runtime it needs. You can copy that one file to another machine and run it, with no interpreter or virtual machine to install. Go can also cross-compile: from a laptop running Windows you can build a binary for Linux by setting two environment variables, `GOOS` and `GOARCH`.

A promise of compatibility

When Go 1.0 was released, the Go team published the **Go 1 compatibility promise**: programs written to the Go 1 specification should keep compiling and running correctly with later Go 1 releases. New versions add features and improve performance, but they do not break your code. A new Go version is released twice a year, and upgrading is usually uneventful. This book targets **Go 1.25**, and most of the code also runs on Go 1.23 or later. When a feature needs a newer version, we say so.

> **NOTE**
> The language is called Go. "Golang" is a nickname that grew out of the old website address, golang.org, and it is useful when searching the web. The official home today is `go.dev`. The friendly blue mascot is called the Go gopher.

What Go deliberately leaves out

A good way to understand a language is to look at what it chose not to include. Each omission below is intentional, and each one removes a source of confusion or bugs.

Missing feature	What Go does instead
Classes and inheritance	Structs, methods, interfaces, embedding
Exceptions	Errors returned as values
Implicit number conversions	Explicit conversions such as `float64(n)`
Function overloading	Different function names
Default parameter values	Options structs or separate functions
The ternary `?:` operator	A plain `if` statement

Unused variables and imports	Compile errors that force cleanup
A `while` keyword	One loop keyword, `for`

Go does have generics — type parameters were added in Go 1.18 — but they are deliberately modest. You will use them in Chapter 11 when a function really should work with many types, and you will notice that most Go code does not need them at all.

The language is small so that the programs can be big without becoming unreadable.

A first taste of Go

You do not need to understand every detail yet; Chapters 3 to 6 explain each piece. Still, it helps to see real Go before we talk more about it. Here is a complete program that checks Sprigwell's stock list and prints the plants that are running low.

lowstock/main.go

```
package main

import "fmt"

func main() {
    names := []string{"Monstera Deliciosa", "Lavender Hidcote", "Japanese Maple",
        "Basil Genovese", "Snake Plant"}
    stock := []int{8, 24, 2, 0, 12}

    for i, name := range names {
        if stock[i] < 3 {
            fmt.Println("Low stock:", name, stock[i])
        }
    }
}
```

Running it prints:

```
Low stock: Japanese Maple 2
Low stock: Basil Genovese 0
```

Reading the code

Even without any Go knowledge, you can probably follow it. Let us walk through it line by line so the shape becomes familiar.

- `package main` says this file belongs to the package named `main`. A program that you run, rather than a library that others import, always lives in package `main`.
- `import "fmt"` brings in the standard library's formatting package, which knows how to print things.

- `func main()` is where the program starts. When `main` returns, the program ends.
- `names := []string{...}` creates a **slice** of strings — an ordered list — and `:=` declares the variable and infers its type. The second slice, `stock`, holds the matching quantities.
- `for i, name := range names` loops over the slice, giving us both the position `i` and the value `name` on each pass.
- `if stock[i] < 3` applies Sprigwell's rule that fewer than three plants counts as low stock.
- `fmt.Println` prints its arguments separated by spaces and ends with a newline.

Notice what is *not* there: no semicolons at the ends of lines, no parentheses around the `if` condition, and no type annotations on the variables. The compiler inserts semicolons for you. Braces, on the other hand, are required, even around a single-line body.

Reading the output

Only two of the five plants are printed: the Japanese Maple with 2 in stock and the Basil Genovese with none. The Monstera (8), Lavender (24), and Snake Plant (12) all have at least three and are skipped. The order of the output matches the order of the slice, because a slice always keeps its elements in the order you wrote them. (Maps, which you meet in Chapter 6, do *not* promise any order.)

A taste of concurrency

Concurrency is the feature people most often mention when they explain why they chose Go, so here is a small preview. Sprigwell has three greenhouses, and each one reports temperatures through the day. This program averages the readings of two greenhouses at the same time, each in its own goroutine, and collects the answers through channels.

averages/main.go

```
package main

import "fmt"

func average(temps []float64, out chan<- float64) {
    sum := 0.0
    for _, t := range temps {
        sum += t
    }
    out <- sum / float64(len(temps))
}

func main() {
    gh1 := make(chan float64)
    gh2 := make(chan float64)

    go average([]float64{24.0, 25.5, 27.0}, gh1)
    go average([]float64{30.5, 31.5, 32.5}, gh2)
```

```
    fmt.Println("GH-1 average:", <-gh1)
    fmt.Println("GH-2 average:", <-gh2)
}
```

The output is:

```
GH-1 average: 25.5
GH-2 average: 31.5
```

The two calls to `average` start with the keyword `go`, so each one runs as a separate goroutine while `main` carries on. The expression `<-gh1` receives a value from the channel `gh1`, waiting until the goroutine sends its result. Because `main` receives from `gh1` first and `gh2` second, the output always appears in that order, even if the second goroutine happens to finish first. Notice also the explicit conversion `float64(len(temps))`: `len` returns an `int`, and Go will not quietly mix an `int` with a `float64`.

Tomas, who looks after the greenhouses, would point out that GH-2 averages above 30 °C. Sprigwell's alert rule says a temperature above 30 or a humidity below 40 percent should raise an alert, and in Part IV we will build exactly that: a small system that watches readings from all three greenhouses concurrently and raises alerts.

> **TIP**
> You can paste either program into the Go Playground at `go.dev/play` and press Run. You do not need anything installed to start experimenting, although Chapter 2 sets up a proper local environment, which you will want for everything after that.

Where Go fits

No language is the best choice for everything, and you will make better decisions if you know where Go shines and where another tool may serve you better.

Where Go is a strong choice

- **Network services and web APIs.** The standard library's `net/http` package is production quality, and goroutines make it natural to handle many connections at once. Chapter 23 builds an HTTP API using nothing but the standard library.
- **Command-line tools.** A Go CLI is a single binary that starts instantly and runs on Windows, macOS, and Linux. Many popular developer tools are written in Go for exactly this reason. Chapter 15 builds one.
- **Cloud and infrastructure software.** Container tools, orchestration systems, proxies, and monitoring agents are a natural fit: they are long-running, concurrent, and need to be easy to deploy. Docker, Kubernetes, and Terraform are all written in Go.

- **Team projects that must last.** Uniform code and a compatible language make old projects easy to join.

Where another tool may be better

- **Interactive data science.** Python's notebooks and numerical libraries are hard to beat for exploring data. Go can process data well, but the exploration experience is better elsewhere.
- **Desktop and mobile user interfaces.** There are Go libraries for graphical applications, but the mainstream toolkits for these platforms live in other languages.
- **Code that must never pause.** Go's garbage collector is designed for very short pauses, which is fine for servers. For hard real-time systems or the tightest embedded devices, languages without a garbage collector, such as C or Rust, are more common.

For the kind of software most businesses need — services, tools, integrations, and automation — Go is an excellent default, and it is a pleasant first compiled language to learn.

If you come from another language

Many readers already know some programming. These short notes tell you what will feel familiar and what will surprise you.

- **From Python:** Go will feel more verbose at first because types are checked and errors are returned explicitly. In exchange, your programs run much faster, ship as a single file, and the compiler catches many bugs that Python would only reveal at runtime. Indentation is not significant; braces are.
- **From JavaScript or TypeScript:** goroutines replace `async` / `await`. You write ordinary sequential code and run it concurrently with `go`; there is no "colored function" split between async and sync code.
- **From Java or C#:** there are no classes, no inheritance, no exceptions, and no annotations. Interfaces are satisfied implicitly. Programs start in milliseconds, with no virtual machine to warm up.
- **From C or C++:** you keep pointers and value semantics but lose manual memory management, header files, and undefined behavior from out-of-bounds access. Go checks slice indexes at runtime and stops with a clear panic message instead of corrupting memory.

Meet Sprigwell Nursery

Learning a language is easier when the examples connect to something concrete. Throughout this book we will write software for **Sprigwell Nursery**, a fictional independent plant nursery and garden shop. Sprigwell is small enough to understand in one sitting and realistic enough to need real software: it sells plants, tracks stock, keeps prices, records sales, and grows its plants in three greenhouses called `GH-1`, `GH-2`, and `GH-3`.

Three people work there, and you will hear from them throughout the book:

- **Lena** owns the nursery. She cares about knowing what is in stock, what is selling, and how much money comes in. Her questions usually start with "Can the computer just tell me…".
- **Tomas** is the greenhouse lead. He is practical, a little skeptical of software, and asks the questions that keep our designs honest: what happens when a sensor stops responding, or when someone types a negative quantity?
- **Priya** is a part-time shop assistant who is learning Go alongside you. She will make the mistakes that every newcomer makes, and we will fix them together. When Priya gets something wrong, pay attention — the same thing will probably happen to you.

Today, Sprigwell keeps its stock list in a spreadsheet that only Lena really understands. Here is the starting catalog. We will use these exact five plants, in this order, as sample data for the whole book.

SKU	Name	Category	Price	Stock
`SPR-001`	Monstera Deliciosa	houseplant	$24.99	8
`SPR-002`	Lavender Hidcote	perennial	$11.99	24
`SPR-003`	Japanese Maple	tree	$89.00	2
`SPR-004`	Basil Genovese	herb	$3.99	0
`SPR-005`	Snake Plant	houseplant	$18.50	12

A **SKU** (stock keeping unit) is a short code that identifies each product. Sprigwell treats any plant with fewer than three in stock as "low", which makes the Japanese Maple and the Basil low, and the Basil completely out of stock.

One decision is worth making right away, because it affects every chapter. Prices look like decimals, but we will store them as whole numbers of **cents**: the Monstera costs `2499` cents, not `24.99` dollars. Floating-point numbers cannot represent most decimal fractions exactly, and small rounding errors add up when you total a day's sales. Integers do not have that problem. When we need to show a price to a person, a small function called `FormatPrice` will turn `2499` into `"$24.99"`.

> **WARNING**
>
> Never store money in a `float64`. It works in quick tests and then fails in production when 0.1 plus 0.2 is not exactly 0.3. In Go, use an integer number of the smallest currency unit — for Sprigwell, cents stored in an `int64`.

What we will build

Over the course of the book, Sprigwell's spreadsheet turns into three pieces of software. Each one grows chapter by chapter as you learn the features it needs.

The `sprig` command-line tool

Lena wants to manage stock from a terminal on the shop computer. The `sprig` tool reads and writes an `inventory.json` file and supports a handful of subcommands. Chapter 15 builds it with the standard library's `flag` package, and Chapter 26 completes it end to end.

```
sprig list
sprig add -sku SPR-006 -name "Rosemary Arp" -category herb -price 450 -stock 10
sprig sell SPR-001 2
sprig restock SPR-004 20
sprig report
```

Every subcommand also accepts a `-data` flag to choose a different file. When something goes wrong — selling five Japanese Maples when only two are left, say — `sprig` prints a clear message to standard error and exits with status code 1, so that scripts can detect the failure. Sales are recorded in a `sales.csv` file that Lena can open in her spreadsheet program.

The `sprigd` web API

Priya would like the shop's tablet to show live stock. For that we build `sprigd`, a small HTTP server listening on port 8080 and written only with the standard library. It answers requests such as `GET /plants` with JSON, and it accepts sales through `POST /plants/{sku}/sell`. Chapter 23 introduces it, Chapter 24 gives it a SQLite database, and Chapter 27 finishes it.

The greenhouse sensors

Tomas has temperature and humidity sensors in each greenhouse. In Part IV we write the `sensors` package, which processes readings from all three greenhouses concurrently, raises alerts when a greenhouse gets too hot or too dry, and fetches prices from two fictional suppliers, Fernline Growers and Oakhill Seeds, at the same time, with timeouts so that a slow supplier cannot hold everything up.

The finished layout

By Chapter 10 our code will be organized as a Go **module** with several packages. You do not need to understand this tree yet; it is here so you can see where the journey ends.

```
sprigwell/
  go.mod                 module example.com/sprigwell
  cmd/sprig/main.go      the CLI binary "sprig"
  cmd/sprigd/main.go     the small HTTP API server
  inventory/             plants, stock rules, errors, FormatPrice
  store/                 saving inventory to JSON files and SQLite
  sensors/               greenhouse readings and alerts
```

The module path `example.com/sprigwell` uses the domain `example.com`, which is reserved for examples and documentation.

How this book is organized

The book has six parts. Each part builds on the ones before it, so we recommend reading in order the first time.

Part	Chapters	What you learn
I — Getting Started	1–5	Tools, basic types, control flow, functions, strings
II — Core Language	6–10	Slices, maps, structs, interfaces, errors, packages
III — Building Real Programs	11–15	Generics, standard library, files, JSON, CLIs
IV — Concurrency	16–20	Goroutines, channels, mutexes, context, patterns
V — Practical Go	21–25	Testing, iterators, HTTP, databases, logging
VI — Projects and Next Steps	26–28	The full CLI, the web API, shipping binaries

Part II ends with packages and modules, the point where Sprigwell's code moves into a proper project layout. Part IV teaches concurrency slowly and carefully, because it is both Go's greatest strength and the place where beginners get hurt. Part VI brings everything together in two complete projects.

Four appendices sit at the back: a syntax and standard library cheat sheet, a reference for the `go` command, a field guide to compiler errors and common gotchas, and a glossary. Keep them in mind when you need to look something up quickly.

This book teaches the Go language itself. If, after finishing it, you want to build production services with gRPC, PostgreSQL, message queues, observability, and Kubernetes, our companion title *Go (Golang) for Backend Developers* picks up where this one leaves off.

How each chapter works

Every chapter follows the same rhythm, so you always know where you are.

1. **Concepts first, with small examples.** Each idea is introduced with a short, complete program you can run. We explain why a feature exists before showing how to use it.
2. **Applying it to Sprigwell.** We then use the new idea in Sprigwell's code, so you see it in a realistic setting rather than only in toy examples.
3. **Reading the compiler error or the output.** We deliberately break things and read the messages Go gives us. Learning to read these messages calmly is one of the most valuable skills you can build.
4. **Common mistakes.** The traps that catch most newcomers, with the fix for each.
5. **Practice.** Three to five exercises, followed by brief selected solutions. Try each exercise before you look.
6. **Key Takeaways.** A short summary to review before moving on.

Callout boxes mark a **Tip** (a good habit), a **Note** (background), or a **Warning** (a costly mistake).

The single most important piece of advice we can give you is this: **type the code yourself**. Copying and pasting feels faster, but typing makes you notice details — a missing brace, a capital letter, a `:=` instead of `=` — and the compiler's reaction to your typos teaches you more than any paragraph we can write.

Common misconceptions

Before we move on, here are a few ideas newcomers often bring to Go that will slow them down.

- **"Go is just a faster Python."** Go is statically typed and compiled. It rewards you for thinking about types and errors up front. Trying to write Python in Go leads to frustration; writing Go in Go leads to clear, fast programs.
- **"Concurrency makes everything faster."** Running things concurrently only helps when there is independent work to do, such as waiting on several network calls. Adding goroutines to sequential work just adds complexity. Part IV teaches when to reach for concurrency and when not to.
- **"I need a framework to build a web service."** The standard library is enough for a great many services, and this book uses it exclusively. You will understand any framework better once you know what lies underneath.
- **"The compiler is being picky."** When Go refuses an unused import or a mismatched type, it is protecting future readers of the code. Treat compiler errors as a conversation, not a scolding.

Practice

These exercises are about thinking rather than typing, since we install Go in the next chapter. If you want to try the code exercises now, use the Go Playground.

1. **Design goals in your own words.** Pick three of Go's design goals from this chapter and, for each one, write a sentence describing a problem it solves that you have personally experienced in another language or tool.
2. **Change the threshold.** In the low-stock program, Lena wants to see every plant with fewer than 10 in stock. Which single character sequence do you change, and what would the program print?
3. **Add a plant.** Extend the same program with a sixth plant, "Rosemary Arp", with 1 in stock. Which two lines must change, and what is the new output?
4. **Predict the averages.** In the concurrency program, suppose GH-3 reports `18.0`, `20.0`, and `22.0`. What average would be printed, and would it trigger Sprigwell's temperature alert?
5. **Is Go a fit?** For each project, decide whether Go is a strong choice and say why in one sentence: (a) a nightly job that reads sales files and emails a summary; (b) an exploratory analysis of five years of sales with charts; (c) a small service that answers stock queries from the shop tablet.