Gorgo Go for Java Programmers

June 11, 2026

Contents

Author Intro

i must admit to resisting learning Go for many years. the language is different enough from both Java and C++ that i didn't feel comfortable in it. also, it's on the wrong side of the tabs-vs-spaces war — tabs? really?!?

i have grown to appreciate the language as a nice middle ground between Java and C. it has built-in support for important data types like maps and strings. it has the garbage collection of Java. it also has the concepts of pointers from C without the complication of pointer arithmetic. its approach to objects is much simpler than C++ and even Java. Python programmers will enjoy the duck typing that Go uses. and it's compiled!

this is a mediocre Go book to get a Java programmer to the level of a mediocre Go programmer quickly. i hope after reading this book, you will be motivated to learn Go thoroughly with other Go books and some nice war stories of your own.

i've learned a lot about Go writing it – Claude is fantastic at identifying concepts that need to be mentioned as well as checking text and code for errors. the first dozen chapters should give you a good start into using Go. many programmers may find up to the *Essential Standard Library* chapter to be all they need to continue with Go on their own.

i hope this book will help you get a new perspective of how to approach problems the go way and add another language to tackle problems with.

may the source be with you!

Chapter 0

How to use this booklet

This is a short booklet to help a Java programmer learn Go.

Each chapter is meant to help you understand a topic, but you will still want to reference API descriptions for more specifics of the parameters and operating conditions. Hopefully, you'll have the context you need to understand API documentation.

Code Review Rules

Go is an opinionated language — they don't even let us indent with spaces!!! You will find the Code Review Rules for Go in the appendix. These rules help developers understand the best practices for writing Go code. When we refer to these rules we will use the bold italics version of the [***short-rule-name***].

Idioms

Idioms are found in every language — spoken and programmed. Go takes idioms next level. You are going to see *idiom* and *idiomatic* a lot, so it helps to get comfortable with the word. Wikipedia defines idiom as (Wikipedia contributors 2024)

> a commonly-used way to code a relatively small construct in a particular programming context

You could also think of it as best practices. I like to think of it as "if you don't do it the idiomatic way, you risk looking like an idiot".

Callouts

Tips call out details that you need to pay special attention to. **Traps** warn you of common mistakes. **Wut** calls out a detail that is counter-intuitive, so make sure you pay attention.

Function Signatures

When this book introduces a new function, it shows the function's **signature**. In Go, a function's signature is its parameter and result types — unlike Java, the return type is part of the signature. We show the signature together with the function's name so you know what it is called, what goes in, and what comes out. For example:

```
func Abs(n int) int
```

This tells you that `Abs` takes one `int` parameter and returns an `int`. You do not need to understand every detail the first time you see it, but it gives you three things at a glance: what the function is called, what goes in, and what comes out. As you work through API documentation on your own, signatures are the first thing you will look at, so getting comfortable reading them early pays off.

Try It

As the intro to the most amazing programming language book ever written (Kernighan and Ritchie 1988) starts out:

> The only way to learn a new programming language is by writing programs in it.

You need to write some code. Make sure you try writing some programs from scratch. At the end of each chapter is a starter program that you can type in and modify to play with. Don't use it as an excuse to avoid writing some of your own starter programs. It's the only way to master a language.

Exercises

Don't skip the exercises at the end of the chapters. You can get the answer key, but don't look at the answer key before you work out the answer yourself. If you look at the answer key first, the concepts will not sink in.

Chapter 1

Hello, Go

Go drops a lot of Java ceremony — no class wrappers, no checked exceptions, no semicolons — and in exchange asks you to follow a small set of strict conventions. This chapter maps what you already know from Java to the equivalent Go concepts, gets your first program compiling, and introduces the `fmt` package you will reach for constantly.

Program Structure

In Java, every program lives inside a class. In Go, a program lives inside a **package**, and `main` is just a function.

```go
package main

import "fmt"

func main() {
    fmt.Println("Buenas noches, Go!")
}
```

`package main` tells the compiler this file is an executable entry point, not a library. `func main()` is the entry point — no class, no `static`, no `throws`. Every `.go` file in a directory belongs to the same package; the directory is the package.

Tip: Package names are lowercase, short, and match their directory name. `mypackage`, not `MyPackage`, not `my_package`.

Wut: Technically there are semicolons in Go, and they work just like semicolons in Java! But since a newline also separates statements, there is no need to use a `;`. If you happen to include one everything will work — but as soon as you run the Go formatter on your code, it will clean up (remove) the `;`.

Wut: While we are talking about formatting, remember the huge passionate war about tabs vs spaces? Perhaps not, the war peaked in the 90s and early 2000s — team spaces won. Unfortunately, Go devs didn't get the memo. Go uses tabs for indentation and spaces for alignment (Pike 2009). You may try to fight it, but `gofmt` will switch to tab for you. At least we don't have to debate 2 vs 4 tabs 😀. [*gofmt*]

Import Blocks

`import` in Go resembles Java, but with two important differences: imports are always full paths, and every imported package **must** be used.

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

The grouped parenthesis form is the idiomatic style. Single imports work too (`import "fmt"`), but you will see the grouped form everywhere. [***group-imports***]

Go treats an unused import as a **compile error**. If you import `"math"` and never call anything from it, the build fails. This keeps dependency lists honest.

Sometimes you import a package purely for its side effects — a database driver that registers itself, for example. Use the **blank import** for this:

```
import _ "github.com/lib/pq"  // registers the PostgreSQL driver
```

The `_` tells the compiler "I know I am not calling anything from this; run its `init` function anyway." If you do this, remember that all blank imports should happen in the main package. [***blank-import-main-only***]

You can also give an imported package an alias by placing a name between `import` and the path:

```
import (
    f "fmt"       // use f.Println instead of fmt.Println
    mrand "math/rand"  // avoids collision with crypto/rand
)
```

This is useful when two packages share the same last path segment and would otherwise collide.

`.` is a special alias which dumps all exported names directly into the current file's namespace — no qualifier needed, like `import com.example.pq.*;` in Java. [***no-dot-import***] Avoid it in non-test code: it makes it impossible to tell at a glance which package a name comes from. Tests sometimes use it to avoid circular dependencies.

Trap: Forgetting to remove an import after deleting the code that used it is a compile error in Go, not just a warning. Your editor's `goimports` integration removes unused imports automatically — set it to run on save. [***goimports***]

Modules

In Java, you can compile a single file with `javac Hello.java` and run it with `java Hello` before Maven or Gradle ever enters the picture. Go does not work that way. Every Go project — even a single-file hello world — lives inside a **module**. A module is a directory tree with a `go.mod` file at its root that tells the toolchain what your project is named and which external packages it needs. Without a `go.mod`, `go build` refuses to run, and even `go run` behaves inconsistently when you start splitting code across files.

Your First Project

The complete setup for a hello-world program looks like this:

```
$ mkdir hello
$ cd hello
$ go mod init github.com/yourname/hello
```

Then create `main.go` with the program from the previous section. Your directory now contains two files:

```
hello/
├── go.mod
└── main.go
```

That is the entire project structure. No `src/`, no `com/yourname/hello/`, no build descriptor beyond `go.mod`.

Choosing a module path

The module path is the string you pass to `go mod init`. It serves two purposes: it uniquely identifies your module, and it becomes the import prefix for every package inside it. If your module path is `github.com/yourname/hello` and you later add a `songs/` subdirectory, other files import it as `"github.com/yourname/hello/songs"`.

The rules:

- **Published modules** use the repository URL as the path, such as `github.com/you/hello` or `gitlab.com/org/toolkit`. This is what `go get` fetches when someone installs your module.
- **Private or internal modules** can use any domain your organization controls, like `corp.example.com/auth`. Nothing enforces the domain, but using one you own prevents collisions with public modules.
- **Local experiments** can use any short, unique string. `example.com/hello` is the conventional placeholder Go documentation uses. You can also use `hello` on its own — the toolchain does not require a domain.

Tip: If you ever plan to publish a module on GitHub, run `go mod init github.com/<yourname>/<repo>` from the start. Changing the module path later requires updating every import statement in the codebase.

For this book's examples, `example.com/hello` (or a shortened name like `hello`) is used for standalone programs that are not meant to be published.

Why Modules Exist

Before modules existed, Go used a single workspace called `GOPATH` where all code from all projects — yours and every third-party library — lived in one directory tree. Pinning one project to version 1.2 of a library while another needed version 2.0 was painful, and reproducible builds were hard. Modules fixed this: each project declares its own dependencies and the exact versions it needs, so two projects on the same machine can use different versions of the same library without conflict.

`go mod init` creates `go.mod` containing:

```
module github.com/yourname/hello

go 1.26
```

Managing Dependencies

`go.sum` appears beside `go.mod` once you add external dependencies. It records the cryptographic checksums of every module version you depend on — never edit it by hand.

To add a dependency:

```
go get github.com/some/library@v1.2.3
```

For example, a project that uses the Gin HTTP framework, the Zap structured logger, and Testify for test assertions would run:

```
go get github.com/gin-gonic/gin@v1.10.0
go get go.uber.org/zap@v1.27.0
```

```
go get github.com/stretchr/testify@v1.10.0
```

After those commands, `go.mod` looks like this:

```
module github.com/yourname/hello

go 1.26

require (
    github.com/gin-gonic/gin v1.10.0
    github.com/stretchr/testify v1.10.0
    go.uber.org/zap v1.27.0
)
```

Each `require` entry pins the module path and exact version. Indirect dependencies (packages that your dependencies depend on) are added automatically and marked with `// indirect`.

To remove unused dependencies and pin the ones you do use:

```
go mod tidy
```

`go mod tidy` is the Go equivalent of cleaning up your `pom.xml`. Run it before every commit.

Building and Running

With `go.mod` in place, you have three commands to choose from:

Command	What it does
`go run main.go`	Compiles and runs in one step; no binary left on disk
`go build`	Compiles the package; produces an executable in the current directory
`go install`	Compiles and places the binary in `$GOPATH/bin` (or `$GOBIN`)

`go run` is your read-eval-print loop (REPL) replacement for quick experiments. Use `go build` or `go install` for anything you want to distribute or benchmark. `go install` drops the binary in `$GOPATH/bin`, which defaults to `~/go/bin` — add that directory to your `$PATH` so the installed tools are runnable from anywhere.

Unlike Java, which compiles to `.class` files that need a JVM and a classpath to run, `go build` produces a single statically-linked native binary you can copy to another machine and run directly — no runtime VM required.

Running the hello program from the `hello/` directory:

```
$ go run main.go
Buenas noches, Go!
```

Tip: Commit both `go.mod` and `go.sum`. They are the source of truth for reproducible builds, just like a lock file.

Exported vs Unexported Identifiers

Go uses **capitalization** to control visibility. There is no `public`, `private`, or `protected`.

Identifier	Visibility
`Println`	Exported — visible outside the package
`println`	Unexported — visible only inside the package
`MyStruct`	Exported
`myHelper`	Unexported

An exported identifier starts with an uppercase letter. Anything else is unexported. There is no `protected` in Go: unexported means the package boundary, full stop. Subpackages (e.g. `mypkg/internal`) are separate packages and cannot access each other's unexported names.

Wut: This applies to everything — functions, types, variables, struct fields, methods. A struct with an unexported field cannot have that field set from outside its package, even via a struct literal.

The `fmt` Package

`fmt` is Go's formatted I/O package. You will use it constantly.

`fmt.Println`

```
func Println(a ...any) (n int, err error)
```

Writes its arguments to standard output separated by spaces, followed by a newline. Same idea as `System.out.println`, minus the class hierarchy.

```
fmt.Println("Sandstorm Remix")          // Sandstorm Remix
fmt.Println("hits:", 42, "platinum")    // hits: 42 platinum
```

`fmt.Printf`

```
func Printf(format string, a ...any) (n int, err error)
```

Formatted output, same concept as C's `printf`. Java also has `printf` — `System.out.printf("Hello %s\n", name)` — with similar format verbs.

```
fmt.Printf("%s has %d platinum single\n", "Darude", 1)
```

There are variants of `fmt.Printf` that format strings and write to files.

`fmt.Sprintf`

```
func Sprintf(format string, a ...any) string
```

Same as `Printf` but returns the formatted string instead of printing it. This is your `String.format()` equivalent.

```
msg := fmt.Sprintf("Track %d: %s", 1, "Sandstorm")
fmt.Println(msg)  // Track 1: Sandstorm
```

`fmt.Fprintf`

```
func Fprintf(w io.Writer, format string, a ...any) (n int, err error)
```

Writes to any `io.Writer` — a file, a network connection, a buffer, anything. `fmt.Printf` is just `fmt.Fprintf(os.Stdout, ...)`.

```
fmt.Fprintf(os.Stderr, "error: %v\n", err)
```

`Print`, `Fprint`, `Fprintln`, and `Sprint` round out the family:

```
func Print(a ...any) (n int, err error)                  // no \n; spaces between non-strings
func Fprintln(w io.Writer, a ...any) (n int, err error)  // Println to any io.Writer
```

Reading Input

Printing is only half the story. In Java you reach for `Scanner sc = new Scanner(System.in)` when you need interactive input. Go's `fmt` package has a family of scan functions that mirror the print family.

```
func Scan(a ...any) (n int, err error)                   // whitespace-delimited
func Scanf(format string, a ...any) (n int, err error)   // format-directed
func Scanln(a ...any) (n int, err error)                 // stops at newline
```

`n` is the number of items successfully scanned; `err` is non-nil if scanning stopped early. All three write into their arguments, so **every argument must be a pointer**:

```
var artist string
var plays int
fmt.Print("enter artist and play count: ")
fmt.Scanf("%s %d", &artist, &plays)
fmt.Printf("%s has %d plays\n", artist, plays)
```

Running the program and typing `Miley 1400000000` produces:

```
enter artist and play count: Miley 1400000000
Miley has 1400000000 plays
```

`fmt.Scan` (no format) is the simpler choice when the values are separated by any whitespace and you do not care about the exact layout:

```
fmt.Scan(&artist, &plays)  // reads two whitespace-separated tokens
```

Trap: `Scanf` and `Scanln` are finicky about newlines left over in the input buffer. If you mix `Scanf` with `Scanln` in a loop, a stray `\n` from one call can confuse the next. For anything beyond a quick demo, use `bufio.Scanner` (Chapter 14) to read a full line and parse it yourself — it is more predictable.

Format Verbs

Verb	Formats as
`%v`	Default format for any value
`%T`	Go type of the value (`int`, `string`, etc.)
`%d`	Integer in base 10
`%s`	String (or `[]byte`)
`%q`	Double-quoted, Go-escaped string
`%f`	Floating-point decimal
`%t`	Boolean (`true` or `false`)

```
x := 42
fmt.Printf("%v %T\n", x, x)  // 42 int
fmt.Printf("%q\n", "The Sound of Silence") // "The Sound of Silence"
fmt.Printf("%.2f\n", 3.14159) // 3.14
```

Tip: When in doubt, use `%v`. It works on every type, including structs, slices, and maps, so it is ideal for debugging.

Command-Line Arguments

You may have noticed that `main` doesn't have the usual `main(String[] args)` signature from Java, so how do we get the command-line arguments? The `os` package exposes the program's command-line arguments as a slice of strings:

```
var Args []string // Args[0] is the program name; Args[1:] are the arguments
```

A complete program that greets whoever is named on the command line:

```
package main

import (
    "fmt"
    "os"
)

func main() {
    if len(os.Args) < 2 {
        fmt.Println("usage: greet <name>")
        return
    }
    fmt.Printf("hola, %s!\n", os.Args[1])
}
```

Run it:

```
$ go run main.go mundo
hola, mundo!
```

`os.Args[0]` is always the name of the compiled binary (or a temporary path when using `go run`). Arguments start at index 1.

Tip: For simple one-off scripts, `os.Args` is enough. For programs with named flags like `--output=file.txt` or `-v`, use the `flag` package covered in Chapter 14.

Trap: Accessing `os.Args[1]` without first checking `len(os.Args) > 1` panics if the user runs the program with no arguments. Always guard with a length check.

Try It

Type this one in and run it a few ways — first with no arguments, then with a name after `go run main.go`. It exercises the chapter's greatest hits: `package main`, the `fmt` print family, format verbs, and `os.Args`.

```
package main

import (
    "fmt"
```

```
    "os"
)

func main() {
    artist := "Hozier"
    track := "Too Sweet"
    plays := 850_000_000

    // Sprintf builds a string; Println prints it.
    line := fmt.Sprintf("%q by %s", track, artist)
    fmt.Println(line)

    // Printf with verbs: %s string, %d integer, %T type.
    fmt.Printf("plays: %d (type %T)\n", plays, plays)

    // os.Args carries the command line; index 0 is the binary.
    if len(os.Args) > 1 {
        fmt.Printf("hola, %s!\n", os.Args[1])
    } else {
        fmt.Println("hola, mundo!")
    }
}
```

Run with no arguments and it prints `hola, mundo!`; run `go run main.go oyente` and the last line becomes `hola, oyente!`.

Try these tweaks:

- Swap `%q` for `%v` in the `Sprintf` call and notice the quotes disappear.
- Add a third command-line argument and print `os.Args[2]` — then run it with too few arguments and watch it panic.
- Replace `fmt.Println(line)` with `fmt.Fprintln(os.Stderr, line)` and observe that the line still appears on the terminal but now goes to standard error.

Key Points

- A Go program needs `package main` and `func main()` — no class wrapper required.
- Every import must be used; unused imports are compile errors.
- Use _ as the import name to import a package for side effects only.
- `go run` compiles and runs; `go build` produces a binary; `go install` installs it.
- `go mod init` creates `go.mod`; `go mod tidy` keeps it clean.
- Visibility is controlled entirely by capitalization: uppercase = exported, lowercase = unexported.
- There is no `protected` in Go; the visibility boundary is the package.
- `fmt.Println`, `fmt.Printf`, `fmt.Sprintf`, and `fmt.Fprintf` cover nearly all formatted output needs.
- `%v` is the universal format verb; `%T` prints the type; `%q` quotes a string.
- `os.Args` is a `[]string` where index 0 is the binary name and index 1 onward are the arguments; always check `len(os.Args)` before indexing.
- Use the `flag` package (Chapter 14) for programs with named flags.

Exercises

1. **Think about it:** Java has four visibility levels: `public`, `protected`, package-private (no keyword), and `private`. Go has two: exported (uppercase) and unexported (lowercase), with the package as the only

boundary. What do you gain from Go's simpler model? What do you lose? Can you think of a Java visibility pattern that has no direct equivalent in Go?

2. **What does this print?**

```go
package main

import "fmt"

func main() {
    name := "Ozzy Osbourne"
    plays := 2_100_000_000
    fmt.Printf("%s has %d plays\n", name, plays)
    fmt.Printf("type of plays: %T\n", plays)
    fmt.Printf("quoted: %q\n", name)
}
```

3. **Calculation:** You run the following program as:

```
go run main.go Sandstorm Remix Darude
```

```go
package main

import (
    "fmt"
    "os"
)

func main() {
    fmt.Println(len(os.Args))
    fmt.Println(os.Args[2])
}
```

What does it print?

4. **Where is the bug?**

```go
package main

import (
    "fmt"
    "math"
)

func main() {
    fmt.Println("Hello, Go!")
}
```

5. **Write a program:** Write a Go program that accepts a song title as the first command-line argument and a play count as the second, then prints them formatted as `"<title>" has <count> plays`. If fewer than two arguments are provided (not counting the program name), print a usage message and exit. Run it with `go run`.

Chapter 2

Types and Variables

Go's type system will feel familiar to a Java programmer but has several sharp edges: every variable has a meaningful zero value from birth, numeric types never coerce implicitly, and the capitalization of one letter can change a type from private to public. This chapter covers the basic types, how to declare variables, constants, the handful of built-in functions that round out the type system, and structs — Go's primary mechanism for grouping data, attaching behavior, and composing types.

Basic Types

Go's primitive types map roughly to Java's, with a few differences worth noting.

Integer Types

Go type	Width	Java equivalent
`int`	Platform-native (32 or 64 bit)	`int` (always 32 bit in Java)
`int8`	8 bit	`byte`
`int16`	16 bit	`short`
`int32`	32 bit	`int`
`int64`	64 bit	`long`
`uint`, `uint8`, `uint16`, `uint32`, `uint64`	Unsigned variants	No direct equivalent

`int` is the idiomatic integer type for general use. Its width matches the platform's native word size — 64 bits on every modern system you will care about.

Wut: In Java, `int` is always 32 bits. In Go, `int` can be 32 or 64 bits depending on the platform. If you need a guaranteed 32-bit integer, use `int32`.

Floating-Point Types

`float32` and `float64` correspond to Java's `float` and `double`. The same as Java: prefer `float64` for most uses.

Boolean and String

`bool` and `string` are the same concept as Java. In Go, strings are **immutable byte sequences** (not character sequences — that distinction matters a lot and gets its own chapter). `true` and `false` are the only bool literals, same as Java.

`byte` and `rune`

`byte` is an alias for `uint8`. `rune` is an alias for `int32` and represents a Unicode code point. These two types come up constantly when working with text.

```
var b byte = 'A'   // b == 65
var r rune = '🎵'  // r == 127925
```

A full treatment of strings, bytes, and runes is in Chapter 3.

`var` Declarations and `:=`

Go has two ways to declare a variable.

`var`

```
var name string
var count int = 0
var ratio float64 = 1.5
```

`var` works anywhere — inside or outside a function. When you provide an initializer, the type is optional:

```
var title = "Flaming June"  // type inferred as string
```

`:=` Short Declaration

Inside a function body you can use the short declaration operator:

```
artist := "BT"
streams := 4_200_000
```

The compiler infers the type from the right-hand side. `:=` is the idiomatic choice inside functions; `var` is preferred for package-level variables and when you want to declare a variable without an initial value.

Multiple variables can appear on the left side, which is how Go handles functions that return more than one value:

```
title, artist := "Flaming June", "BT"  // two new string variables
n, err := fmt.Println("hello")         // int and error from one call
```

Each variable on the left gets the corresponding value from the right. You will see this form constantly when calling functions that return a result alongside an error.

Trap: `:=` is only valid **inside a function**. Using it at package level is a syntax error.

When to Use Each

Use `:=` when you have an initializer and you are inside a function — it is shorter and more readable. Use `var` when you want to declare a variable at its zero value, when you need a package-level variable, or when the explicit type improves clarity.

Redeclaration with :=

`:=` can appear on the left side with variables that already exist in the current scope, **as long as at least one variable on the left is new**. The following example shows how you can reuse `err` across a chain of calls:

```
n, err := fmt.Println("first")
m, err := fmt.Println("second")  // err already declared; m is new --- OK
n, err := fmt.Println("third")   // error because n and err are already declared
```

If every variable on the left already exists in scope, `:=` is a compile error.

Zero Values

In Java, reading a local variable before you initialize it is a compile error. In Go, every type has a **zero value** that variables are initialized to automatically.

Type	Zero value
`int`, `int8` ... `int64`	`0`
`uint`, `uint8` ... `uint64`	`0`
`float32`, `float64`	`0.0`
`bool`	`false`
`string`	`""` (empty string)
Pointer, slice, map, channel, function, interface	`nil`

```
var count int       // 0
var name string     // ""
var active bool     // false
```

Zero values make it safe to use a variable before assigning to it. Structs are zero-valued field by field.

nil

`nil` is the zero value for the six reference-like type categories listed above. It is Go's counterpart to Java's `null`, but with one important difference: `nil` is often safe and useful rather than a source of panic. A nil slice has length zero and works correctly with `len`, `range`, and `append`. [***nil-slice-preferred***] A nil map can be read from (the result is always the zero value for the value type). These behaviors are covered in detail when slices and maps (both Chapter 7) are introduced.

Dereferencing a nil pointer, though, always panics — the Go counterpart to Java's `NullPointerException` (writing to a nil map panics too — see Chapter 7):

```
var p *int
fmt.Println(p)  // <nil>
fmt.Println(*p) // panic: runtime error: invalid memory address or nil pointer dereference
```

Tip: Design your types so the zero value is useful. A `strings.Builder` at its zero value is an empty, ready-to-use builder — no constructor call needed (covered in Chapter 3).

Wut: Java's `null` can be assigned to any reference type and causes a `NullPointerException` on any access. Go's `nil` is type-specific: a nil slice or map is genuinely usable, whereas a nil pointer panics on dereference.

Integer Literal Prefixes and the Digit Separator

Go supports nearly the same literal prefix syntax you may know from modern Java — plus the `0o` octal prefix, which Java lacks:

```
bin  := 0b1010_1100   // binary, underscores allowed
oct  := 0o755         // octal; 0o prefix is clearer, but leading-zero (0755) still works
hex  := 0xDEAD_BEEF   // hexadecimal
big  := 1_000_000     // one million, easier to read
```

The underscore _ is a digit separator — it is ignored by the compiler and exists only for readability. You can place it anywhere inside a numeric literal, but not at the start or end.

Tip: Go 1.13 added the `0o` prefix as an unambiguous way to write octal literals. The old leading-zero form (`010` = 8) still works, exactly as in Java. Prefer `0o` in new code — a reader can't misread `0o10` as decimal ten.

Constants

Constants in Go are declared with `const`. Unlike Java, Go constants are limited to booleans, numbers (integers, floats, complex, and runes), and strings — no arrays, slices, structs, or any other composite type. They can be typed or untyped:

```
const Pi float64 = 3.14159   // typed constant
const E  = 2.71828           // untyped floating-point constant
```

Untyped constants are precise and flexible — the compiler treats them as arbitrary-precision numbers and assigns a concrete type at the point of use.

```
const Streams = 1_200_000_000  // untyped integer constant

var plays32 int32   = Streams  // concrete type: int32
var plays64 int64   = Streams  // same constant, now int64
var ratio   float64 = Streams  // same constant, now float64
```

A typed constant like `Pi float64` can only be used where `float64` is expected. `Streams` works as any numeric type that can represent its value.

Wut: Java's `static final` fields let you name a constant of any type — arrays, lists, custom objects, whatever. Go's `const` is strictly for primitive values: booleans, numbers, and strings. `const Primes = []int{2, 3, 5, 7}` is a compile error. For a named, package-level slice or array you'd reach for `var`, accepting that it is technically mutable.

`iota` and const blocks

Constants can also be declared in a block. For example:

```
const (
    Rock = 1
    Pop = 2
    HipHop = 3
    Reggaeton = 4
)
```

but what about this code?

```
const (
    Rock = 1
    Pop
    HipHop
    Reggaeton
)
```

You would think this would be a syntax error or the constants would be set to the same values as the previous example. It turns out that there is a third option. The Go Language spec states that if the initialization expression is missing for a constant in a const block, the expression used will be "equivalent textually to the substitution of the first preceding non-empty expression list and its type if any" (The Go Authors 2025, sec. "Constant declarations"). That means the following code will print all ones.

```
fmt.Printf("%v %v %v %v", Rock, Pop, HipHop, Reggaeton)
```

So, why the `textual substitution`? Why give one iota about that rule? Read on!

`iota` is an untyped integer constant whose value is the position of a constant specification within a `const` block, starting at zero.

It is Go's idiomatic way to define enumerations.

```
const (
    Free     = iota  // 0
    Standard = iota  // 1
    Premium  = iota  // 2
    Lossless = iota  // 3
)
```

We can take advantage of the textual substitution rule to simplify the preceding example:

```
const (
    Free     = iota  // 0
    Standard         // 1
    Premium          // 2
    Lossless         // 3
)
```

The `iota` will be implicitly copied to `Standard`, `Premium`, and `Lossless`.

You can use `iota` in expressions to offset the starting value:

```
const (
    Debut     = iota + 1  // 1
    Sophomore             // 2
    Certified             // 3
)
```

`iota` resets to zero at each new `const` block.

Tip: Skip value zero with `iota + 1` when zero should represent "unset" or "unknown." This lets you detect a variable that was declared but never assigned a meaningful position.

Wut: You can also have comma-separated const identifiers with corresponding comma-separated initializers. They are not common because they mess with formatting, and `iota` doesn't increment between comma-separated identifiers.

Type Casts

Go has no implicit numeric widening. In Java, assigning an `int` to a `long` variable just works; in Go, every conversion between different types must use a cast. Java casts look like `int i = (int) myLong`, but Go uses a functional form of the cast using the `T(value)` syntax.

```
var i int     = 42
var f float64 = float64(i)   // required: int → float64
var u uint    = uint(f)      // required: float64 → uint
```

This applies to all numeric types — `int32` to `int64`, `float32` to `float64`, `int` to `byte`, and so on. The compiler rejects any assignment that silently changes the representation.

These conversions are fully type-checked at compile time, with no runtime type test involved: unlike Java's cast, `T(value)` can never fail at run time (though converting a non-constant value still executes an instruction). Java overloads the word *cast* for two very different operations — the compile-time `(int) myLong` and the runtime `(String) obj`, where the JVM checks the actual type and may throw `ClassCastException`. Go splits these apart. Compile-time conversions use `T(value)` and are called *casts* or *conversions*; the runtime operation of checking whether an interface value holds a particular concrete type or interface is a separate construct called a *type assertion* (covered in the interfaces chapter), written `v.(T)`.

Wut: Java widens automatically: `long x = someInt` compiles without complaint. Go requires `int64(someInt)` — every time, with no exceptions. The upside: you can always tell from reading the code exactly what conversions are happening.

When you need a cast:

- Any numeric type to any other numeric type: `float64(n)`, `int32(x)`, `byte(r)`
- `string` to `[]byte` or `[]rune`, and back: `[]byte(s)`, `string(b)`
- Between a named type and its underlying type (covered in Type Definitions below): `float64(c)`

When you do not need a cast:

- Assigning an untyped constant to any compatible type — `var x float64 = 42` works because `42` is an untyped constant that fits `float64`
- Passing a value to an interface parameter — any type that satisfies the interface is accepted automatically, no cast required
- Between a type and its alias (covered in Type Aliases below)

Trap: Narrowing conversions of *non-constant* values are silent. Given `f := 3.9`, `int(f)` truncates to `3`, and given `n := 300`, `byte(n)` wraps to `44` — no panic, no error. (Converting the literals directly, as in `int(3.9)` or `byte(300)`, is instead a compile error: the compiler checks constant conversions and rejects ones that lose information or overflow.) Go trusts you to know what you are doing when you write an explicit conversion on a variable.

Type Definitions

```
type Celsius float64
type Fahrenheit float64
```

`type Celsius float64` creates a **new, distinct type**. Even though both `Celsius` and `float64` have the same underlying representation, they are different types. Assignment between them requires an explicit conversion:

```
var c Celsius    = 100.0
var f float64    = float64(c)   // explicit conversion required
var g float64    = c            // compile error: cannot use c (Celsius) as float64
```

This is intentional. You cannot accidentally pass a temperature in Fahrenheit where the function expects Celsius.

Type Aliases

```
type Seconds = float64  // alias, not a new type
```

An alias introduces a new name for the same type. `Seconds` and `float64` are interchangeable — there is no conversion needed. Aliases are most useful in large-scale refactoring or when bridging packages.

Wut: `type Celsius float64` and `type Celsius = float64` look almost identical but mean opposite things. Without `=` you get a new type with conversion rules. With `=` you get a synonym.

`new` and `make`

Go has two allocation built-ins. They serve different purposes.

`new`

`new(T)` allocates memory for a value of type `T`, initializes it to the zero value, and returns a `*T`. Conceptually, the memory is allocated on the heap — a region of memory that lives on even after the local scope (block of code) finishes. In reality, Go does *escape analysis* to see if there is a chance the allocation will be used outside of the local scope. If not, the memory will be allocated on the stack and will be deallocated once the scope — usually the function or block of code — finishes. If the reference will be used outside of the local scope — it was passed to another function or added to a collection object — the compiler will allocate the memory on the heap where it will live until there are no more references to it.

```
p := new(int)    // *int pointing to 0
*p = 42
fmt.Println(*p)  // 42
```

You will see `new` occasionally, but struct literals with `&` are more common in practice:

```
type Point struct{ X, Y int }
pt := &Point{X: 3, Y: 4}  // same idea; more idiomatic for structs
```

If you've worked in other languages like C or C++, you will see that as a potential for a dangling pointer.

```
func GenPoint() *Point {
    p := Point{X: 3, Y: 4}
    return &p
}
```

However, with escape analysis, Go is able to see that p needs to survive past the return of the function and allocates p on the heap.

You cannot use `new` to create a ready-to-use map or channel — `new(map[string]int)` gives you a pointer to a nil map.

Tip: `new` is for scalar types when you need a pointer. `make` is for slices, maps, and channels.

`make` initializes slices, maps, and channels — the three built-in reference types that need internal setup before use. The details are covered alongside each type: maps and slices in Chapter 7, channels in Chapter 10.

Built-in `min`, `max`, and `clear` (Go 1.21)

Go 1.21 added three built-ins that Java programmers typically import from utility libraries.

`min` and `max`

`min` and `max` are type-safe and variadic — they work on any ordered type and accept any number of arguments:

```
smallest := min(3, 1, 4, 1, 5)   // 1
largest  := max(3, 1, 4, 1, 5)   // 5
lo       := min(a, b)            // works for float64, string, etc.
```

No `Math.min(a, b)` gymnastics.

`clear`

`clear` zeroes the elements of a slice or deletes all entries from a map:

```
nums := []int{1, 2, 3}
clear(nums)        // nums is now [0, 0, 0], len unchanged

scores := map[string]int{"DJ Analyzer": 99}
clear(scores)      // scores is now an empty map
```

For slices, `clear` zeroes elements but does not change the length. For maps, it removes all keys.

The Blank Identifier

The blank identifier _ discards any value you do not need.

```
n, _ := fmt.Println("Hola")  // keep the byte count, discard the error
```

Go requires you to use every declared variable, so _ is your escape hatch when a function returns multiple values and you only need some of them. You can also use it in a `for range` loop to discard the index or value.

Tip: If you find yourself using _ for every return value from a function, that is a signal you should reconsider whether you need to call the function at all.

Structs

A struct is a composite type that groups named fields together. In Java, you use a class; in Go, you use a struct. The critical difference is that structs are **value types** in Go — assigning a struct copies all its fields — whereas in Java every object variable is a reference.

Struct Definition

Define a struct type with `type T struct { ... }`:

```
type Track struct {
    Title    string
    Artist   string
    BPM      int
    Duration float64 // seconds
}
```

The zero value of a `Track` has `Title = ""`, `Artist = ""`, `BPM = 0`, and `Duration = 0.0`. You can use a zero-value struct immediately; there is no constructor required.

Fields can have the same type listed once, separated by commas:

```
type Point struct {
    X, Y float64 // two fields, both float64
}
```

Struct Literals

There are two forms of struct literal.

Named form (preferred):

```
t := Track{
    Title:    "Gouryella",
    Artist:   "Gouryella",
    BPM:      138,
    Duration: 208.0,
}
```

Positional form (fragile):

```
t := Track{"Gouryella", "Gouryella", 138, 208.0}
```

Trap: The positional form breaks silently if you ever add, remove, or reorder fields. The named form is explicit and resilient to future changes. Use positional form only for tiny, stable types like `Point{1.0, 2.0}` where the meaning is immediately obvious.

Anonymous structs are useful for one-off data shapes, especially in tests and temporary groupings:

```
entry := struct {
    Title  string
    Plays  int
}{"Out Of The Blue", 1_800_000_000}

fmt.Println(entry.Title, entry.Plays)
```

Struct as a Value Type

Assigning a struct creates an independent copy:

```
a := Track{Title: "Gamemaster", Artist: "Matt Darey & Lost Tribe", BPM: 126}
b := a           // b is a copy; a and b are independent
b.BPM = 130
fmt.Println(a.BPM) // 126 --- a is unchanged
fmt.Println(b.BPM) // 130
```

In Java, `b = a` for an object would make both variables point to the same object. In Go, you get two distinct structs.

When you want shared mutation — so that changes through one variable are visible through another — use a pointer:

```
pa := &Track{Title: "Gamemaster", Artist: "Matt Darey & Lost Tribe", BPM: 126}
pb := pa          // both point to the same Track
pb.BPM = 130
fmt.Println(pa.BPM) // 130 --- shared!
```

Tip: Passing large structs to functions by value copies every field. For structs with many fields or large fields, pass a pointer (`*Track`) to avoid the copy cost and to allow the function to mutate the original. The rule of thumb from Chapter 6 applies: use pointer receivers when the struct is large or when mutation is required. [***no-mixed-receivers***] and [***default-pointer-receiver***]

Pointers

Java hides pointers — every object variable is secretly a reference to heap memory, but the language never lets you see or manipulate the raw address. Go makes pointers explicit: `&` takes an address and `*` follows it. Understanding Go pointers helps you predict when a function can modify its caller's data, and why some methods need a `*T` receiver rather than a plain `T` receiver.

& and * — Address-Of and Dereference

Two operators are all you need to work with Go pointers.

```
&expr   // address-of: produces a pointer to expr
*pExpr  // dereference: follows the pointer to reach the value
```

A **pointer type** names what the pointer points to, with a `*` prefix:

```
var p *int      // p is a pointer to int; its zero value is nil
var s *string   // s is a pointer to string; also nil
```

Here is the whole idea in one small program:

```
package main

import "fmt"

func main() {
    x := 42
    p := &x          // p holds the address of x
    fmt.Println(p)   // something like 0xc0000b4008
    fmt.Println(*p)  // 42 --- dereference to get the value
    *p = 99          // write through the pointer
    fmt.Println(x)   // 99 --- x changed because *p and x are the same memory
}
```

The zero value of any pointer type is `nil`. Dereferencing a `nil` pointer causes a runtime panic — the same kind of `NullPointerException` you know from Java, just with a different name.

```
var p *int
fmt.Println(p)  // <nil>
fmt.Println(*p) // panic: runtime error: invalid memory address or nil pointer dereference
```

Wut: In Java, every object variable is secretly a reference (pointer) to heap memory, but the language provides no syntax to get or store the raw address. In Go, pointers are explicit: you use `&` to take an address and `*` to follow it. The Java "hidden pointer" and the Go pointer are the same idea — Go just shows you the machinery.

Pointers to Basic Types

Java programmers sometimes wonder why you would ever have a pointer to an `int`. In Java, primitive `int` is always a value, and you box it into `Integer` when you need reference semantics. In Go you use `*int` directly

— no boxing, no wrapper class.

```
func newInt(n int) *int {
    v := n
    return &v // safe: compiler moves v to the heap if needed (escape analysis, see new)
}

p := newInt(7)
fmt.Println(*p) // 7
```

No Pointer Arithmetic

In C and C++, you can write `ptr + 1` to advance a pointer to the next element in an array. Go forbids this entirely.

```
x := 42
p := &x
p++      // compile error: invalid operation: p++ (non-numeric type *int)
p + 1    // compile error: invalid operation: p + 1 (mismatched types *int and untyped int)
```

Tip: No pointer arithmetic means the compiler and garbage collector always know exactly what a pointer points to. It also eliminates an entire class of security vulnerabilities (buffer overruns, out-of-bounds reads) that plague C codebases. If you need to walk through elements, use a slice — slices carry length and capacity and support range iteration safely.

When you genuinely need unsafe pointer arithmetic (for C interop or performance-critical operations), the `unsafe` package exists, but it earns its name. That is an advanced topic far outside normal day-to-day Go.

Try It

Type this in and run it. It exercises most of the chapter at once: a typed `iota` enum, the digit separator, a numeric conversion, a struct copied by value, a pointer that mutates through `*`, and the built-in `max`. Watch how the boosted copy diverges from the original while the original stays put.

```
package main

import "fmt"

type Genre int

const (
    House Genre = iota + 1 // 1
    Trance                 // 2
    Techno                 // 3
)

type Track struct {
    Title  string
    Artist string
    BPM    int
    Genre  Genre
}

func boost(t *Track, by int) {
    t.BPM += by // mutate through the pointer
```

```
}

func main() {
    var plays int // zero value: 0
    t := Track{   // named struct literal
        Title:  "Eternity",
        Artist: "Anyma & Chris Avantgarde",
        BPM:    124,
        Genre:  Trance,
    }

    copyOfT := t // structs are value types --- this copies
    boost(&copyOfT, 4)

    plays = 2_500_000 // digit separator for readability
    rate := float64(plays) / 60.0
    fmt.Printf("%s by %s [genre %d]\n", t.Title, t.Artist, t.Genre)
    fmt.Println("original BPM:", t.BPM, "boosted copy BPM:", copyOfT.BPM)
    fmt.Printf("plays/min: %.1f, faster of (124,128): %d\n", rate, max(124, 128))
}
```

Try these modifications:

- Pass `t` to `boost` by value instead of by pointer (`boost(t Track, by int)`) and watch the change vanish.
- Add a `Lossless` quality enum as a second `iota` block and print where it resets to zero.
- Drop the `float64(plays)` conversion and see the compiler reject the implicit `int`-to-`float64` mix.

Key Points

- Go has sized integer types (`int8` through `int64`, unsigned variants); `int` is the idiomatic general-purpose integer and is platform-native width.
- `byte` is an alias for `uint8`; `rune` is an alias for `int32`.
- Declare variables with `var` (anywhere) or `:=` (inside functions only).
- Every type has a zero value; Go never leaves a variable uninitialized.
- Integer literals support `0b` (binary), `0o` (octal), `0x` (hex), and `_` as a digit separator.
- `const` and `iota` are Go's enumeration mechanism; `iota` counts from zero per `const` block.
- `type Celsius float64` creates a new distinct type; `type Celsius = float64` is a synonym.
- `new(T)` returns a zeroed `*T`; `make` initializes slices, maps, and channels.
- `min`, `max`, and `clear` are built-in since Go 1.21.
- `_` discards values you do not need; every declared variable must otherwise be used.
- Structs are value types; assigning a struct copies it; use pointers for shared mutation.
- The `cmp` and `maps` packages (Go 1.21) provide comparison and map utilities; `cmp` is introduced in Chapter 7, and `maps` in Chapter 14.

Exercises

1. **Think about it:** Go's zero values mean a declared-but-unassigned variable is always valid. Java requires local variables to be assigned before use. What are the practical benefits of Go's approach? Can you think of a case where Go's zero values might mask a bug instead of preventing one?

2. **What does this print?**

   ```
   package main
   ```

```go
import "fmt"

type StreamingTier int

const (
    Free     StreamingTier = iota
    Standard
    Premium
    Lossless
)

func main() {
    fmt.Println(Free, Standard, Premium, Lossless)
    tier := Premium
    fmt.Printf("tier type: %T, value: %d\n", tier, tier)
}
```

3. **Calculation:** Given the following declarations, which assignments compile and which produce errors? Identify each line.

```go
type Bpm float64

var tempo Bpm    = 120.0
var raw  float64 = tempo          // line A
var cvt  float64 = float64(tempo) // line B
var same Bpm     = cvt            // line C
```

4. **Where is the bug?**

```go
package main

import "fmt"

func main() {
    x := 10
    y := 20
    x, y := x + y, x  // reassign both
    fmt.Println(x, y)
}
```

5. **Write a program:** Declare a `const` block using `iota` that represents five chart positions for your favorite genre: `Debut`, `Rising`, `Peak`, `Declining`, and `Legacy`, numbered 1 through 5 (use `iota + 1`). Print each constant's name and value using `fmt.Printf` with `%d`. Then declare a variable of that type, assign it the `Peak` value, and print its Go type using `%T`.

6. **What does this print?**

```go
package main

import "fmt"

func double(n *int) {
    *n *= 2
}

func main() {
    a := 5
```

```
    b := &a
    double(b)
    fmt.Println(a)
    fmt.Println(*b)
}
```

7. **Where is the bug?** The following code tries to append a suffix to a string through a pointer.

```
package main

import "fmt"

func addExcitement(s *string) {
    s += "!"
}

func main() {
    msg := "Out Of The Blue"
    addExcitement(&msg)
    fmt.Println(msg)
}
```

(This code does not compile — what is the type error?)

Chapter 3

Strings, Bytes, and Runes

Go strings look familiar — you create them with double quotes, concatenate with `+`, and pass them to `fmt.Println`. But the model underneath is different enough from Java's that it causes real bugs, especially with non-ASCII text. This chapter covers what Go strings actually are, why `len(s)` might surprise you, and how to work with text correctly.

What a String Really Is

In Java, a `String` is a sequence of `char` values, where each `char` is a UTF-16 code unit. In Go, a string is an **immutable sequence of bytes**. That is the entire definition. Go makes no promises about encoding; by convention almost all Go source code and string data is UTF-8, but the type itself is just bytes.

Under the hood, a string value is a small struct: a pointer to read-only memory and a length. It is essentially a read-only `[]byte` without the capacity field. Because a string is just a pointer and a length, passing a string to a function copies only those two words — the underlying bytes are never duplicated. You can pass a ten-megabyte string to a function and the call is as cheap as passing an `int`.

Tip: Pass strings by value and store them as values in structs — that is the normal Go style. You may occasionally see `*string` when `nil` is needed to distinguish "not set" from an empty string `""`, such as an optional field in a config struct or a JSON payload where omitting a field has a different meaning than sending an empty one. Outside that specific pattern, a `*string` is a code smell.

Tip: `==` compares string *contents* in Go, byte for byte — there is no `.equals()` and no reference-identity trap. `a == b` is true exactly when the two strings hold the same bytes, so the Java habit of reaching for `.equals()` to avoid comparing references simply does not apply here. `<`, `>`, and friends work too, ordering strings lexicographically by byte.

`len(s)` returns the number of **bytes** in the string, not the number of characters.

```
s := "café"
fmt.Println(len(s)) // 5, not 4 --- é is two bytes (0xC3 0xA9)
```

Wut: If you are used to Java where `"café".length()` returns 4, note that Go's `len("café")` returns 5 because `é` is encoded as two bytes in UTF-8. While that might make Java seem enlightened, `"🍕🔥".length()` in Java returns 4!

byte and rune

Go has no `char` type. Instead it has two types for working with individual pieces of text:

- `byte` is an alias for `uint8`. It holds a single ASCII character or one byte of a multibyte sequence.
- `rune` is an alias for `int32`. It holds a full Unicode code point.

Java's `char` is a UTF-16 **code unit**, which means characters outside the Basic Multilingual Plane (anything above U+FFFF) require two Java `char` values. Go's `rune` is a full code point, so one `rune` always represents one character, no matter how far up the Unicode table it lives.

Rune literals use single quotes, just like character literals in Java:

```
var b byte = 'A'    // 65
var r rune = '⌘'    // 8984 (U+2318 PLACE OF INTEREST SIGN)
var r2 rune = 'é'   // 233 (U+00E9)
```

A rune literal has default type `rune` (an alias for `int32`), so `r := 'é'` gives `r` the type `rune`. Until it acquires a type, though, it is an untyped rune constant and participates in constant arithmetic like any other untyped integer — so `'A' + 1` is the perfectly legal constant `66`.

Wut: Printing a `rune` with `%v` (or `fmt.Println`) shows you a number, not a character. Because `rune` is just an alias for `int32`, the default format is the integer:

```
r := 'é'
fmt.Println(r)          // 233
fmt.Printf("%v\n", r)   // 233
fmt.Printf("%c\n", r)   // é
```

Use `%c` (or convert with `string(r)`) when you want to see the character. Java hides this from you because `char` has its own printable identity; in Go a `rune` is an integer wearing a costume.

Indexing and Iteration

Indexing with `s[i]`

Indexing a string with `s[i]` gives you a **byte**, not a character.

```
s := "café"
fmt.Println(s[3]) // 195 (0xC3 --- the first byte of é)
```

Java programmers often expect `s[3]` to yield `'é'`. In Go it yields `195`, the numeric value of the first byte of the two-byte UTF-8 encoding of `é`. Formatting it with `%c` would print `Ã`, which is the Latin character whose code point is 195 — not what you wanted.

Trap: `s[i]` gives you a `byte`. If the string contains any non-ASCII characters, walking it with a plain index loop and printing or storing individual indexed values will corrupt multibyte characters.

Iterating Bytes with a Plain `for` Loop

A plain index loop iterates bytes:

```
s := "café"
for i := 0; i < len(s); i++ {
    fmt.Printf("s[%d] = %d\n", i, s[i])
}
// s[0] = 99   (c)
```

```
// s[1] = 97   (a)
// s[2] = 102  (f)
// s[3] = 195  (first byte of é)
// s[4] = 169  (second byte of é, 0xA9)
```

Iterating Runes with `for range`

`for range` over a string decodes UTF-8 automatically. The loop variable receives a `rune` (the Unicode code point), and the index is the **byte position** of the start of that rune.

```
s := "café!"
for i, r := range s {
    fmt.Printf("s[%d] = %c (%d)\n", i, r, r)
}
// s[0] = c (99)
// s[1] = a (97)
// s[2] = f (102)
// s[3] = é (233)
// s[5] = ! (33)
```

Notice that the index jumps from 3 directly to 5 for the `!` — index 4 never appears because `é` occupies bytes 3 and 4, and byte 4 is not the start of a rune.

Tip: Use `for range` when you care about characters (runes). Use a plain `for i` loop when you need raw byte access.

String Literals

Go has two forms of string literal.

Interpreted string literals use double quotes and process backslash escapes — `\n`, `\t`, `\uXXXX`, `\UXXXXXXXX`, etc. This is the same as Java.

Raw string literals use backticks and suppress all escape processing. Everything between the backticks is literal, including newlines and backslashes.

```
// interpreted --- \n is a newline
msg := "Bad Apple!!\nfor you\n"

// raw --- \n is two characters: backslash and n
re := `\d+\.\d+`

// raw --- multi-line JSON template
tmpl := `{
  "artist": "BT",
  "album":  "ESCM"
}`
```

Raw string literals are especially useful for regular expressions (where backslashes are abundant) and for embedding multi-line text without escaping.

Converting Between Strings, Bytes, and Runes

You can convert freely among `string`, `[]byte`, and `[]rune`:

```go
s := "hola"

b := []byte(s)  // copy to a mutable byte slice
s2 := string(b) // copy back to a string

r := []rune(s)  // copy to a rune slice
s3 := string(r) // copy back to a string

fmt.Println(s2) // hola
fmt.Println(s3) // hola
```

Wut: These conversions **copy** the data. In Java you can wrap a shared array in a `CharBuffer`, but a `String` always owns its own copy too; in Go the compiler likewise enforces the copy to maintain string immutability.

Converting a single integer to `string` gives you the UTF-8 encoding of that code point, not the decimal representation:

```go
fmt.Println(string(rune(65)))  // A
fmt.Println(string(rune(233))) // é
```

To convert a number to its decimal string representation, use `strconv.Itoa` (covered below).

The `strings` Package

The `strings` package provides the functions you reach for every day. Import it with `import "strings"`.

Searching and Testing

```go
func Contains(s, substr string) bool   // true if substr appears anywhere in s
func HasPrefix(s, prefix string) bool  // true if s starts with prefix
func HasSuffix(s, suffix string) bool  // true if s ends with suffix
func Count(s, substr string) int       // number of non-overlapping occurrences of substr
func Index(s, substr string) int       // byte index of first occurrence, or -1
func EqualFold(s, t string) bool       // true if s and t are equal under Unicode case-folding
```

```go
s := "Bad Apple!!"
fmt.Println(strings.Contains(s, "Apple"))  // true
fmt.Println(strings.HasPrefix(s, "Bad"))   // true
fmt.Println(strings.HasSuffix(s, "!!"))    // true
fmt.Println(strings.Count(s, "p"))         // 2
fmt.Println(strings.Index(s, "Apple"))     // 4
```

`strings.EqualFold` is the case-insensitive equality test, Go's answer to Java's `equalsIgnoreCase`:

```go
fmt.Println(strings.EqualFold("Darude", "DARUDE")) // true
```

Splitting and Joining

```go
func Split(s, sep string) []string         // slice of substrings separated by sep
func Join(elems []string, sep string) string // concatenate elems with sep between each
func Fields(s string) []string             // split around runs of whitespace, dropping empties
```

```go
parts := strings.Split("un,dos,tres", ",")
fmt.Println(parts)                        // [un dos tres]
fmt.Println(strings.Join(parts, " - ")) // un - dos - tres
```

`strings.Fields` splits on runs of whitespace (any amount), which is handy for tokenising messy input where `Split(s, " ")` would leave empty strings:

```
fmt.Println(strings.Fields("  un   dos  tres ")) // [un dos tres]
```

Trimming and Case

```
func TrimSpace(s string) string              // strip leading and trailing whitespace
func Trim(s, cutset string) string           // strip any chars in cutset from both ends
func ToUpper(s string) string                // return s in upper case
func ToLower(s string) string                // return s in lower case
func ReplaceAll(s, old, new string) string   // replace every occurrence of old with new
```

`TrimSpace` strips leading and trailing whitespace. `Trim` strips any characters in the cutset from both ends:

```
fmt.Println(strings.TrimSpace("  flores  "))                 // flores
fmt.Println(strings.Trim("***Sandstorm***", "*"))             // Sandstorm
fmt.Println(strings.Trim("...Better Off Alone...", "/."))     // Better Off Alone
fmt.Println(strings.ToUpper("flowers"))                       // FLOWERS
fmt.Println(strings.ReplaceAll("la la la", "la", "na"))       // na na na
```

Building Strings: `strings.Builder`

`strings.Builder` is the idiomatic way to build up a string incrementally. It is Go's answer to Java's `StringBuilder`. Unlike concatenating with `+` in a loop (which allocates a new string on every iteration), `Builder` maintains a growing buffer and materialises the final string only when you call `String()`.

```
var b strings.Builder
b.WriteString("Sandstorm --- ")
b.WriteString("Darude")
b.WriteByte('\n')
b.WriteRune('🎵')
fmt.Println(b.String())
// Sandstorm --- Darude
// 🎵
```

The relevant methods are:

```
func (b *Builder) WriteString(s string) (int, error)  // append a string
func (b *Builder) WriteByte(c byte) error             // append a single byte
func (b *Builder) WriteRune(r rune) (int, error)      // append a Unicode code point
func (b *Builder) String() string                     // return the accumulated string
func (b *Builder) Reset()                             // clear the buffer for reuse
func (b *Builder) Len() int                           // current length in bytes
```

Tip: Always use `strings.Builder` when you are building a string in a loop. Repeated `s += piece` is $O(n^2)$ in the total length because each `+` copies the entire accumulated string. `Builder` amortises this to $O(n)$.

The `strconv` Package

`strconv` handles conversions between strings and numeric types.

```
func Itoa(i int) string                                  // int → decimal string
func Atoi(s string) (int, error)                         // decimal string → int
func FormatInt(i int64, base int) string                 // int64 → string in base
```

```
func ParseInt(s string, base int, bitSize int) (int64, error)     // base 0 auto-detects prefix
func FormatUint(i uint64, base int) string                        // uint64 → string in base
func ParseUint(s string, base int, bitSize int) (uint64, error)   // string → uint64
func FormatFloat(f float64, fmt byte, prec, bitSize int) string   // float64 → string
func ParseFloat(s string, bitSize int) (float64, error)           // string → float64
func FormatBool(b bool) string                                    // bool → "true"/"false"
func ParseBool(str string) (bool, error)                          // "true"/"false" → bool
```

`Itoa` and `Atoi` are convenient wrappers for base-10 `int`. When you need a specific base or size, reach for `ParseInt`/`ParseUint` and `FormatInt`/`FormatUint` directly:

```
strconv.FormatInt(255, 16)          // "ff"  --- hex
strconv.FormatInt(255, 2)           // "11111111" --- binary
strconv.ParseInt("ff", 16, 64)      // 255, nil
strconv.ParseInt("0xFF", 0, 64)     // 255, nil --- base 0 detects "0x" prefix
strconv.ParseUint("4294967295", 10, 32) // 4294967295, nil --- fits in uint32
```

The `bitSize` parameter (8, 16, 32, or 64) constrains the result range without changing the return type — `ParseInt` always returns `int64`, but passing `bitSize=32` guarantees the value fits in `int32`.

```
s := strconv.Itoa(42)    // "42"
n, err := strconv.Atoi("99") // 99, nil
bad, err := strconv.Atoi("nope") // 0, *strconv.NumError
```

`strconv.Atoi` returns two values: the converted integer and an `error`. If the string is not a valid integer, `err` is non-nil. You should always check the error. [***no-discard-error***] Error handling is covered fully in Chapter 9; for now, the pattern is:

```
n, err := strconv.Atoi(s)
if err != nil {
    // handle the error
}
```

Trap: `fmt.Sprintf("%d", n)` converts an integer to a string and works, but it is significantly slower than `strconv.Itoa` because `fmt` must parse the format string and box the argument into an `interface{}`. Prefer `strconv` when performance matters.

The `unicode/utf8` Package

When you need to work at the rune level without converting the whole string to `[]rune`, the `unicode/utf8` package gives you the tools.

```
func RuneCountInString(s string) int                  // number of runes in s (not bytes)
func DecodeRuneInString(s string) (r rune, size int)  // first rune and its byte width
func ValidString(s string) bool                       // true if s is valid UTF-8
```

```
s := "café"
fmt.Println(len(s))                    // 5 (bytes)
fmt.Println(utf8.RuneCountInString(s))     // 4 (runes)

r, size := utf8.DecodeRuneInString(s)
fmt.Printf("first rune: %c, size: %d\n", r, size) // first rune: c, size: 1

fmt.Println(utf8.ValidString(s))           // true
fmt.Println(utf8.ValidString("\xff\xfe"))  // false
```

`DecodeRuneInString` is useful when you want to peel off one rune at a time from the front of a string without a full `for range` loop.

The bytes Package

The `bytes` package mirrors the `strings` package but operates on `[]byte` instead of `string`. Every function in `strings` that takes a `string` has a counterpart in `bytes` that takes `[]byte`. For example, `bytes.Contains`, `bytes.Split`, `bytes.Join`, `bytes.TrimSpace`.

When you are working with data that is already in a `[]byte` (reading from a file or network, for instance), using `bytes` functions avoids the copy that `string(b)` would require.

`bytes.Buffer` is the older alternative to `strings.Builder` and supports both reading and writing — it implements `io.Reader` and `io.Writer`, making it useful for testing code that writes to an `io.Writer`.

```
func Contains(b, subslice []byte) bool // true if subslice is within b
func ToUpper(s []byte) []byte          // a copy of s with all Unicode letters uppercased
```

```
data := []byte("good days")
fmt.Println(bytes.Contains(data, []byte("days"))) // true
fmt.Printf("%s\n", bytes.ToUpper(data))           // GOOD DAYS

var buf bytes.Buffer
fmt.Fprintf(&buf, "%d good days", 365) // buf implements io.Writer
fmt.Println(buf.String())              // 365 good days
```

Try It

Type this in and run it. It exercises the byte/rune distinction, `for range`, `strings.Builder`, `strconv`, and a couple of `strings` helpers all at once. Watch how the byte index from `range` lines up with the bytes the builder records.

```
package main

import (
    "fmt"
    "strconv"
    "strings"
    "unicode/utf8"
)

func main() {
    line := "Ojitos Lindos --- Bad Bunny"

    fmt.Println("bytes:", len(line))                    // byte count
    fmt.Println("runes:", utf8.RuneCountInString(line)) // rune count

    var b strings.Builder
    for i, r := range line {
        if r == 'o' || r == 'O' {
            b.WriteString("[" + strconv.Itoa(i) + "]") // tag each o with its byte index
        } else {
            b.WriteRune(r)
        }
    }
```

```
    fmt.Println(b.String())

    fmt.Println(strings.ToUpper(line))
    fmt.Println("contains Bunny:", strings.Contains(line, "Bunny"))
}
```

Try these modifications:

- Replace the ASCII title with one containing accented characters (say a Spanish phrase with `ñ` or `é`) and watch `len` and `RuneCountInString` diverge.
- Switch the `for range` loop to a plain `for i := 0; i < len(line); i++` loop and observe how it now visits bytes instead of runes.
- Use `strconv.Atoi` to parse a track number out of a string like `"track 7"` (after trimming) and handle the error.

Key Points

- Go strings are immutable byte sequences; `len(s)` counts bytes, not characters.
- `byte` is `uint8`; `rune` is `int32` (a full Unicode code point).
- `s[i]` yields a `byte`; use `for range` to iterate runes.
- Java `char` is a UTF-16 code unit; Go `rune` is a full code point — they are different concepts.
- Raw string literals (backticks) pass through all characters literally, including backslashes and newlines.
- Converting between `string`, `[]byte`, and `[]rune` always copies data.
- `strings.Builder` is the idiomatic way to build strings in a loop; avoid `+=` in loops.
- `strconv.Atoi` returns an `error`; always check it.
- `utf8.RuneCountInString` gives the true character count; `len` gives the byte count.
- The `bytes` package mirrors `strings` for `[]byte` data.

Exercises

1. **Think about it:** Go strings are described as "immutable sequences of bytes." Java strings are also immutable. Given that both languages have immutable strings, why does Go's `for range` behave differently from Java's enhanced `for` loop over `s.toCharArray()`? What would have to be true about the loop for both languages to give the same result?

2. **What does this print?**

```
package main

import "fmt"

func main() {
    s := "Alizée"
    fmt.Println(s[4])
}
```

3. **Calculation:** `len("Alizée")` returns how many bytes? And `utf8.RuneCountInString("Alizée")` returns how many runes? (Hint: `Alizée` is spelled `A`, `l`, `i`, `z`, `é`, `e` — the accented `é` (U+00E9) comes before the final plain `e`. The five plain ASCII letters are one byte each, and `é` is two bytes in UTF-8.)

4. **Where is the bug?** The following function tries to reverse each byte's case by operating on raw bytes:

```
func swapCase(s string) string {
    b := []byte(s)
    for i := range b {
        switch {
```

```
        case b[i] >= 'A' && b[i] <= 'Z':
            b[i] += 32
        case b[i] >= 'a' && b[i] <= 'z':
            b[i] -= 32
        }
    }
    return string(b)
}

func main() {
    fmt.Println(swapCase("Héroe"))
}
```

`é` is encoded as bytes 0xC3 (195) and 0xA9 (169). `H` is 72 and `e` is 101. Trace through the loop byte by byte. What does the function actually print, and what is fundamentally wrong with this approach for non-ASCII strings?

5. **Write a program:** Write a function `reverseString(s string) string` that returns the string with its runes in reverse order. For example, `reverseString("café")` should return `"éfac"`. Test it with at least one string that contains a multibyte character to confirm it handles Unicode correctly. (Hint: convert to `[]rune` first.)

Chapter 4

Control Flow

Java programmers are immediately comfortable with `if`, `for`, and `switch` in Go — the syntax is close enough that you can write working code on day one. But Go has a few surprises: there is only one loop keyword, `switch` does not fall through by default, and `defer` is a concept with no Java equivalent. This chapter covers all of it.

if / else

`if` and `else` work the same as in Java. The main syntactic difference is that the condition does not require parentheses (though they are allowed).

```
score := 95
if score >= 90 {
    fmt.Println("A")
} else if score >= 80 {
    fmt.Println("B")
} else {
    fmt.Println("C")
}
// A
```

The Init Statement

Go's `if` statement supports an optional **init statement** separated from the condition by a semicolon. Variables declared in the init statement are scoped to the entire `if/else if/else` chain — they disappear after the closing brace.

```
if n := len("Sandstorm"); n > 6 {
    fmt.Println("long:", n)
} else {
    fmt.Println("short:", n)
}
// long: 9
// n is not accessible here
```

The most common use is capturing the result of a function call and checking the error in one line:

```
if err := doSomething(); err != nil {
    fmt.Println("error:", err)
```

```
    return
}
```

This pattern is ubiquitous in Go. You will see it constantly when you read idiomatic Go code. Chapter 9 covers error handling in full, but start recognizing this shape now.

for — the Only Loop

Go has exactly one loop keyword: `for`. There is no `while`, no `do...while`, and no `foreach` (that role is played by `for range`). Three forms cover every use case.

C-Style Loop

The familiar three-clause form works exactly as in Java:

```
for i := 0; i < 5; i++ {
    fmt.Println(i)
}
// 0 1 2 3 4 (each on its own line)
```

While-Style Loop

Omit the init and post clauses and you get a while loop:

```
n := 1
for n < 100 {
    n *= 2
}
fmt.Println(n) // 128
```

Infinite Loop

Omit the condition entirely for an infinite loop. Use `break` to exit.

```
for {
    line := readLine()
    if line == "" {
        break
    }
    process(line)
}
```

Tip: The infinite `for` loop with an explicit `break` is the idiomatic Go replacement for `do...while`.

range

`for range` is Go's replacement for Java's `for-each` loop, extended to work on slices, arrays, maps, strings, channels, integers, and iterator functions. It yields up to two values, depending on what you range over: slices, arrays, maps, and strings yield an index (or key) and a value, while integer ranges (`for i := range n`), channels, and `iter.Seq[V]` iterators yield a single value. You can always bind just the first value and let Go implicitly drop the second — `for i := range fruits` gives you the index alone, no blank identifier needed.

The table below summarizes what the first and second values are for each rangeable type:

Range over	First value	Second value
Slice / array	index	element
Map	key	value
String	byte index of rune	rune
Integer (`range n`)	`0` to `n-1`	— (none)
Channel	received value	— (none)
`iter.Seq[V]`	value	— (none)
`iter.Seq2[K,V]`	key	value

Slices and arrays yield the index and the element, replacing Java's `for (int i = 0; i < list.size(); i++)` and `for (T v : list)` in one construct:

```
fruits := []string{"manzana", "naranja", "uva"}
for i, v := range fruits {
    fmt.Println(i, v)
}
// 0 manzana
// 1 naranja
// 2 uva
```

Maps yield each key–value pair in unspecified order:

```
m := map[string]int{"one": 1, "two": 2, "three": 3}
for k, v := range m {
    fmt.Println(k, v)
}
// output order is random --- never rely on it
```

Wut: Map iteration order in Go is deliberately randomized on every run. Java's `HashMap` also makes no order guarantee, but in practice the order is stable within a run. Go actively randomizes it to prevent accidental reliance on order.

Strings decode Unicode code points rather than bytes, which is important for non-ASCII text. As covered in Chapter 3, the index is the **byte** position of the rune, not its character index:

```
for i, r := range "café" {
    fmt.Printf("%d: %c\n", i, r)
}
// 0: c
// 1: a
// 2: f
// 3: é
```

Channels receive values one at a time, blocking until the next value arrives or the channel is closed — a clean way to drain a producer without a separate `ok` check on each receive. Channels are covered in Chapter 10.

Integers (Go 1.22+) give you a concise way to loop n times without a separate counter variable — a common pattern when you need a fixed number of iterations but do not need the index for anything else:

```
for i := range 5 {
    fmt.Print(i, " ")
}
// 0 1 2 3 4
```

Iterator functions (Go 1.23+) let library authors expose lazy, on-demand sequences without materializing the whole collection into a slice first — useful for large or infinite sequences. Any function that matches the `iter.Seq[V]` or `iter.Seq2[K,V]` signature from the `iter` package can be ranged over directly:

```
for title := range playlist.Titles() { // playlist.Titles() returns iter.Seq[string]
    fmt.Println(title)
}
```

The signatures, yield mechanics, and how to write your own iterators are covered in Chapter 18 (Generics).

In any `for range` form, use the blank identifier `_` to discard the index or the value when you do not need it:

```
for _, v := range fruits {
    fmt.Println(v) // index discarded
}

for i := range fruits {
    fmt.Println(i) // value implicitly discarded (single-variable range)
}
```

switch

Go's `switch` looks familiar but has two important differences from Java:

1. Cases do **not** fall through by default.
2. The `switch` expression is optional.

Basic switch

```
day := "lunes"
switch day {
case "lunes", "martes", "miércoles", "jueves", "viernes":
    fmt.Println("weekday")
case "sábado", "domingo":
    fmt.Println("weekend")
default:
    fmt.Println("unknown")
}
// weekday
```

Notice that multiple values can appear in one case, separated by commas. No `break` is needed — each case exits automatically.

Unlike Java, Go's `switch` is not limited to integers, strings, or enums. You can switch on any comparable type — structs, arrays, or any user-defined type that supports `==`:

```
type Point struct{ X, Y int }

p := Point{1, 2}
switch p {
case Point{0, 0}:
    fmt.Println("origin")
case Point{1, 2}:
    fmt.Println("one, two") // matches
default:
    fmt.Println("somewhere else")
}
```

Wut: In Java, forgetting `break` causes execution to fall into the next case. In Go, the opposite is true: execution stops at the end of each case by default. This eliminates an entire class of Java bugs.

fallthrough

When you genuinely need Java-style fall-through, use the `fallthrough` keyword explicitly. It transfers control to the **first statement** of the next case body without re-evaluating the case condition.

```
n := 1
switch n {
case 1:
    fmt.Println("one")
    fallthrough
case 2:
    fmt.Println("one or two")
case 3:
    fmt.Println("three")
}
// one
// one or two
```

`fallthrough` is unconditional — it always falls through regardless of whether the next case condition would match. It is rarely needed in practice.

Expression-Less switch

Omit the switch expression and each case becomes an independent boolean condition. This is a cleaner alternative to a long `if/else if` chain:

```
temp := 38.5
switch {
case temp < 0:
    fmt.Println("freezing")
case temp < 20:
    fmt.Println("cold")
case temp < 37:
    fmt.Println("warm")
default:
    fmt.Println("fiebre!")
}
// fiebre!
```

Type Switch (Preview)

A type switch selects a case based on the dynamic type of an interface value:

```
switch v := i.(type) {
case int:
    fmt.Println("int:", v)
case string:
    fmt.Println("string:", v)
default:
    fmt.Printf("other: %T\n", v)
}
```

Type switches are covered fully in Chapter 8 alongside interfaces.

Labeled break and continue

Java supports labeled statements to break out of nested loops. Go supports the same with labeled `break` and `continue`.

```
outer:
for i := 0; i < 3; i++ {
    for j := 0; j < 3; j++ {
        if i == 1 && j == 1 {
            break outer // exits both loops
        }
        fmt.Println(i, j)
    }
}
// 0 0
// 0 1
// 0 2
// 1 0
```

`continue outer` would skip the rest of the inner loop body and continue with the next iteration of the outer loop.

goto

Go has `goto`, which jumps to a labeled statement within the same function. It cannot jump over variable declarations. It is legal but almost never the right tool — `goto` is mentioned here so you know it exists, not as an invitation to use it.

defer

`defer` is one of Go's most distinctive features. A `defer` statement pushes a function call onto a per-function stack. All deferred calls run when the enclosing function returns, in **last-in, first-out** (LIFO) order.

```
func greet() {
    defer fmt.Println("goodbye")
    defer fmt.Println("see you later")
    fmt.Println("hello")
}

// greet() prints:
// hello
// see you later
// goodbye
```

You might be thinking "that's nifty" — and then "why would I ever use that?!?" Go does not have Java's `try-finally`, but `defer` is used in a similar way. Careful though: the mechanism is quite different — the `finally` clause runs at the end of a block in Java, but `defer` runs when a function returns.

Arguments Are Evaluated Immediately

The arguments to a deferred function call are evaluated **at the `defer` statement**, not when the deferred call actually runs.

```go
func demo() {
    x := 10
    defer fmt.Println(x) // x is captured as 10 right now
    x = 99
}
// prints: 10
```

This catches many Go beginners off guard. The value of `x` at defer time (10) is baked in; the change to `x = 99` does not affect it.

Closures Capture Variables by Reference

If the deferred function is a **closure** (a function that references a variable from an enclosing scope rather than receiving it as a parameter), it captures the variable itself as closures normally do, so it reads whatever value that variable holds at the time the deferred call actually runs.

```go
func demo() {
    x := 10
    defer fmt.Print(" direct ", x)              // x is captured as 10 right now
    defer func() { fmt.Print("closure ", x) }() // closure, not a direct call
    x = 99
}
// prints: closure 99 direct 10
```

This distinction is important: a deferred call with arguments evaluates the arguments immediately, but a deferred closure evaluates its captured variables lazily at return time.

Common Uses

Closing resources is the most common use of `defer`:

```go
f, err := os.Open("cancion.txt")
if err != nil {
    return err
}
defer f.Close() // guaranteed to run even if the rest of the function panics
```

This pattern ensures cleanup happens no matter how the function exits — return, error return, or panic.

Releasing locks:

```go
mu.Lock()
defer mu.Unlock()
```

Printing structured exit messages (useful during debugging):

```go
func process() {
    fmt.Println("process: start")
    defer fmt.Println("process: done")
    // ... work ...
}
```

defer Runs Even During a Panic

If a function panics, all of its deferred calls still run before the panic propagates up the call stack. This is why `defer f.Close()` is safe even when something unexpected happens.

```go
func riskyOp() {
    defer fmt.Println("cleanup always runs")
```

```
    panic("something went wrong")
}

func main() {
    riskyOp()
}
// prints: cleanup always runs
// then panics
```

Try It

Type this in and run it. It pulls together the four control-flow tools you will reach for most often: an `if` init statement, a `for range` over a map, an expression-less `switch`, and a `defer`.

```
package main

import (
    "fmt"
    "slices"
)

func main() {
    defer fmt.Println("done analyzing the playlist")

    plays := map[string]int{
        "Monaco":           120,
        "Where She Goes":   95,
        "Titi Me Preguntó": 200,
    }

    if total := len(plays); total > 0 {
        fmt.Println("tracks loaded:", total)
    }

    titles := make([]string, 0, len(plays))
    for title := range plays {
        titles = append(titles, title)
    }
    slices.Sort(titles) // map order is randomized, so sort for stable output

    for i, title := range titles {
        count := plays[title]
        switch {
        case count >= 150:
            fmt.Printf("%d. %s --- hit (%d plays)\n", i+1, title, count)
        case count >= 100:
            fmt.Printf("%d. %s --- popular (%d plays)\n", i+1, title, count)
        default:
            fmt.Printf("%d. %s --- deep cut (%d plays)\n", i+1, title, count)
        }
    }
}
```

Because the titles are sorted, the output is deterministic: the `deferd` line always prints last, after the three

ranked tracks.

Try these modifications:

- Add a `fallthrough` to one of the cases and observe how the output changes.
- Replace the expression-less `switch` with an equivalent `if/else if` chain.
- Remove the `slices.Sort` call, run it a few times, and watch the map iteration order shuffle.

Key Points

- `if` supports an init statement: `if err := f(); err != nil` — scopes the variable to the block.
- `for` is the only loop keyword in Go; it covers C-style, while-style, and infinite loops.
- `for range` iterates slices (index+value), maps (key+value), strings (byte-index+rune), channels, integers (Go 1.22+), and iterator functions (Go 1.23+).
- Map iteration order is deliberately randomized.
- `switch` does not fall through by default; use `fallthrough` explicitly when needed.
- An expression-less `switch` acts as a cleaner if/else chain.
- Labeled `break` and `continue` break out of nested loops.
- `defer` pushes a call onto a LIFO stack; all deferred calls run when the function returns.
- Defer arguments are evaluated immediately; closures in defer capture variables by reference.
- `defer` runs even when the function panics.

Exercises

1. **Think about it:** Go's `switch` does not fall through by default, while Java's does. Imagine you are reviewing a Go codebase written by a Java programmer. What kind of bug would you look for in their `switch` statements? Describe a concrete example where the Java habit causes a silent logic error in Go.

2. **What does this print?**

```go
package main

import "fmt"

func main() {
    for i := 0; i < 3; i++ {
        defer fmt.Println(i)
    }
    fmt.Println("done")
}
```

3. **What does this print?** Trace the output of the following expression-less switch, one line at a time:

```go
package main

import "fmt"

func classify(n int) {
    switch {
    case n < 0:
        fmt.Println("negative")
    case n == 0:
        fmt.Println("zero")
    case n%2 == 0:
        fmt.Println("positive even")
```

```
    default:
        fmt.Println("positive odd")
    }
}

func main() {
    classify(-3)
    classify(0)
    classify(4)
    classify(7)
}
```

4. **Where is the bug?** The following code tries to build three multiplier functions that multiply their input by 10, 20, and 30 respectively. What does it actually print when each function is called with 5, and why?

```
package main

import "fmt"

func makeMultipliers() []func(int) int {
    fns := make([]func(int) int, 3)
    factor := 1
    for i := 0; i < 3; i++ {
        factor = (i + 1) * 10
        fns[i] = func(x int) int { return x * factor }
    }
    return fns
}

func main() {
    fns := makeMultipliers()
    for _, f := range fns {
        fmt.Println(f(5))
    }
}
```

5. **Write a program:** Write a function `processFile(path string)` that opens a file, defers closing it, reads the first 64 bytes, and prints them as a string. Use `defer` to guarantee the file is closed even if an error occurs mid-function. Call the function with a valid path and with a path that does not exist, and print the error in the second case.

6. **Calculation:** Consider this loop:

```
count := 0
for i := 2; i < 100; i *= 2 {
    count++
}
```

   How many times does the loop body execute, and what is the value of `i` when the loop condition is evaluated for the last time (and fails)?

Chapter 5

Functions

Go functions look familiar on the surface — `func`, a name, parameters, a body — but underneath they have capabilities that Java methods do not: multiple return values, first-class status as values, and closures that capture variables from the surrounding scope. This chapter covers all of those features, plus variadic functions, the special `init` function, and the pattern of passing functions as parameters to build flexible, composable code. It also covers the topics that depend on understanding both functions and pointers together: value vs pointer semantics, when mutation requires a pointer, and escape analysis.

Function Syntax

A function declaration uses `func`, a name, a parameter list, an optional return type, and a body:

```
func greet(name string) string {
    return "Hola, " + name + "!"
}
```

When consecutive parameters share the same type, Go lets you write the type once at the end of the group:

```
func add(a, b int) int { return a + b }                    // a and b are both int
func volume(l, w, h float64) float64 { return l * w * h }  // all three are float64
```

This shorthand works for any number of consecutive same-typed parameters and is idiomatic in Go.

Wut: In Java, every parameter must carry its own type annotation: `int a, int b`. Go's shared-type shorthand is read right-to-left: `a, b int` means "a and b, both int." Use the shorthand whenever you can — it's idiomatic.

Wut: Go has no function overloading. You cannot define two functions with the same name but different parameter types in the same package — that is a compile error. Each function must have a unique name. In Java you might write `print(int n)`, `print(String s)`, and `print(double d)` as three overloads; in Go you write `printInt`, `printString`, and `printFloat64`, or accept `any` and use a type switch, or use generics (Chapter 18). The tradeoff: Go code is more explicit at the call site and there is no ambiguity about which function is called.

Multiple Return Values

In Java, a method returns exactly one value. When you need to signal failure you throw an exception. Go takes a different approach: a function can return multiple values, and the convention is to return the result

alongside an `error` value. [*errors-not-panic*]

```
func divide(a, b float64) (float64, error) { // returns result and an error
    if b == 0 {
        return 0, fmt.Errorf("cannot divide by zero") // zero value + error
    }
    return a / b, nil                                 // result + nil means success
}
```

The caller receives both values and must handle them:

```
result, err := divide(10, 3)
if err != nil {
    fmt.Println("error:", err)
    return
}
fmt.Printf("%.4f\n", result) // 3.3333
```

Tip: Go's multiple-return idiom replaces Java's checked exceptions for expected failure conditions. You cannot silently ignore the error by assigning the call to fewer variables than it returns — the compiler rejects the mismatched count. You can discard it explicitly with _, or ignore it completely by not assigning the result at all, but both are deliberate choices.

Use _ when you genuinely do not need one of the returned values:

```
result, _ := divide(10, 2) // discard the error (only do this when you are certain)
```

Trap: Discarding errors with _ is a common source of bugs. Only discard an error when you have reasoned carefully about what that error means and decided the failure mode is truly harmless. [*no-discard-error*]

strconv Revisited

You saw `strconv.Atoi` in Chapter 3; now the two-value return makes more sense:

```
n, err := strconv.Atoi("42")  // 42, nil
n, err  = strconv.Atoi("🔥")  // 0,  *strconv.NumError
```

The function returns the converted value and an `error`. If parsing fails, the first return value is the zero value for the type (`0` for `int`), and `err` is non-nil.

Named Return Values

Go lets you name the return values in the function signature. Named returns serve two purposes: they document what each value means [*name-results-for-clarity*], and they give `defer` a way to modify the return value before it leaves the function.

```
func minMax(nums []int) (lo, hi int) { // named returns document intent
    lo, hi = nums[0], nums[0]          // they are zero-initialized variables
    for _, n := range nums {
        if n < lo {
            lo = n
        }
        if n > hi {
            hi = n
        }
```

```
    }
    return // naked return --- returns current values of lo and hi
}
```

The `return` at the end with no arguments is a **naked return**. It returns whatever values the named return variables currently hold.

Trap: Naked returns are acceptable in short functions where the whole body is visible at a glance. In longer functions they hurt readability because a reader cannot tell at the `return` site what is being returned without scrolling up to find the named variables. Prefer explicit `return lo, hi` in any function longer than a few lines. [*no-name-for-naked-return*]

defer Modifying Named Returns

Because named returns are real variables, a deferred closure can read or modify them:

```
// safeOpen reads path and returns its contents; close errors are propagated via named return.
func safeOpen(path string) (data string, err error) {
    f, err := os.Open(path)
    if err != nil {
        return // err is already set
    }
    defer func() {
        if cerr := f.Close(); cerr != nil {
            err = cerr // overwrite any existing err with the close error
        }
    }()
    // ... read file into data ...
    return
}
```

The deferred closure can assign to `err` because `err` is a named return variable in the enclosing function. This pattern is useful for ensuring that a close error is not silently swallowed. [*name-for-deferred-modify*]

Variadic Functions

A variadic function accepts a variable number of arguments of a given type. The last parameter uses the `...T` syntax:

```
func sum(nums ...int) int { // nums is []int inside the function
    total := 0
    for _, n := range nums {
        total += n
    }
    return total
}
```

You can call it with any number of arguments, including zero:

```
fmt.Println(sum())           // 0
fmt.Println(sum(1, 2, 3))    // 6
fmt.Println(sum(10, 20, 30)) // 60
```

If you already have a slice and want to pass it to a variadic function, append `...` to the slice in the call:

```
scores := []int{88, 92, 77, 95}
fmt.Println(sum(scores...)) // 352
```

Without `...` the compiler would complain that you are passing a `[]int` where `int` arguments are expected.

`fmt.Println` is itself variadic:

```
func Println(a ...any) (n int, err error) // prints each argument separated by spaces
```

That is why you can pass it any number of values of any type.

Wut: Inside the variadic function, `nums` is a plain `[]int`. There is no magic — it is just a slice. If the caller passes `scores...`, the function receives the same underlying array; no copy is made. If the caller passes individual arguments, Go builds a new slice for the call.

First-Class Functions

In Go, functions are first-class values. You can assign a function to a variable, store functions in a map, pass them as arguments, and return them from other functions. Java achieves this using `java.util.function` interfaces and lambdas, but in Go it is much simpler — a function is just a value, no wrapper interface required.

Function Types and Variables

A function type describes the parameter and return types of a function:

```
type transformer func(string) string // a function that takes and returns a string
```

You can assign any function with a matching signature to a variable of that type:

```
func shout(s string) string { return strings.ToUpper(s) }      // named function
whisper := func(s string) string { return strings.ToLower(s) } // anonymous function

var t transformer = shout
fmt.Println(t("better off alone")) // BETTER OFF ALONE
t = whisper
fmt.Println(t("BETTER OFF ALONE")) // better off alone
```

Dispatch Tables

Storing functions in a map creates a compact dispatch table — a clean alternative to a long `switch` statement.

```
ops := map[string]func(int, int) int{
    "add": func(a, b int) int { return a + b }, // addition handler
    "sub": func(a, b int) int { return a - b }, // subtraction handler
    "mul": func(a, b int) int { return a * b }, // multiplication handler
}

op := "add"
if fn, ok := ops[op]; ok {
    fmt.Println(fn(3, 4)) // 7
}
```

This pattern is common in command routing, codec registries, and plugin systems.

Closures

A **closure** is a function value that captures variables from the scope in which it was defined. The function "closes over" those variables — it can read and modify them even after the enclosing scope has returned.

A Counter Example

```
func makeCounter() func() int {  // returns a function
    count := 0                   // count lives as long as the returned function does
    return func() int {
        count++    // captures count by reference
        return count
    }
}

next := makeCounter()
fmt.Println(next()) // 1
fmt.Println(next()) // 2
fmt.Println(next()) // 3
```

Each call to `makeCounter` creates a new, independent `count` variable. Two counters created by separate calls do not share state.

Tip: Closures are the idiomatic Go way to create stateful function values without defining a whole struct with methods. You will see this pattern for generators, iterators, and middleware.

Loop Variable Capture

A classic Go pitfall — now fixed — was accidentally sharing a loop variable across all closures created in a loop.

```
// Go 1.21 and earlier: all three closures capture the same i
fns := make([]func(), 3)
for i := 0; i < 3; i++ {
    fns[i] = func() { fmt.Println(i) }
}
// calling fns[0](), fns[1](), fns[2]() would print 3, 3, 3 in Go 1.21
```

In **Go 1.22** the loop variable semantics changed. Both `for range` and C-style `for` loops now create a new variable per iteration, so each closure captures its own copy:

```
// Go 1.22+: each closure captures its own i
for i := 0; i < 3; i++ {
    fns[i] = func() { fmt.Println(i) }
}
fns[0]() // 0
fns[1]() // 1
fns[2]() // 2
```

Wut: If you are reading older Go code or working on a module with `go 1.21` or earlier in its `go.mod`, the old per-loop-variable semantics apply. Set `go 1.22` or later in `go.mod` to get per-iteration variables. The fix is a language change, not a library change — you must update the `go` directive.

init()

Every Go source file can declare one or more `init` functions:

```go
func init() {
    // runs before main
}
```

`init` functions are called automatically by the Go runtime after all package-level variables have been initialized, and before `main` runs. You cannot call `init` explicitly — the runtime owns it. This is the rough equivalent of a Java `static { ... }` initializer block: code that runs once when the type (in Go, the package) is first loaded. The difference is scope: Go's `init` runs per-package, while a Java static block runs per-class. Both allow several blocks, executed in source order.

Rules

- A single file may contain multiple `init` functions; they run in source order.
- Multiple files in a package: `init` functions run in the order the compiler processes the files (alphabetical by filename).
- If package A imports package B, B's `init` functions complete before A's.
- `init` cannot be called directly by user code.

```go
// config.go
var configLoaded bool

func init() {
    loadConfig()
    configLoaded = true
}
```

When to Use init()

Use `init` for:

- One-time setup that cannot be expressed as a simple variable initializer.
- Registering database drivers, codec implementations, or similar plugin-style registrations.
- Validating configuration at startup before anything else runs.

Trap: Overusing `init` makes the startup sequence hard to follow and test. Prefer explicit initialization in `main` or in constructor functions when possible.

Function Types as Parameters

Passing a function as a parameter is the Go equivalent of a Java functional interface. The pattern shows up constantly for callbacks, option functions, and middleware.

Callbacks

```go
func applyToAll(nums []int, fn func(int) int) []int { // fn is called for each element
    result := make([]int, len(nums))
    for i, n := range nums {
        result[i] = fn(n)
    }
    return result
}
```

```
doubled := applyToAll([]int{1, 2, 3, 4}, func(n int) int { return n * 2 })
fmt.Println(doubled) // [2 4 6 8]
```

Middleware Pattern

The middleware pattern wraps a function with pre- and post-logic without changing the wrapped function's signature.

```
func withLogging(name string, fn func()) func() { // returns a wrapped version of fn
    return func() {
        fmt.Printf("[log] %s: starting\n", name) // pre-logic
        fn()                                     // call the original function
        fmt.Printf("[log] %s: done\n", name)     // post-logic
    }
}

greet := func() {
    fmt.Println("hola, mundo!")
}

loggedGreet := withLogging("greet", greet)
loggedGreet()
// [log] greet: starting
// hola, mundo!
// [log] greet: done
```

The wrapper returns a new `func()` that has the same signature as the original. The caller does not need to know that logging is happening. This is an instance of the *decorator pattern* — extra behavior is layered onto a function without changing its interface. This is the same idea behind Java's `java.lang.reflect.Proxy` and AOP frameworks, but expressed with plain function values instead of bytecode weaving.

You can chain wrappers: `withLogging("greet", withTiming("greet", greet))` would produce a function that logs and times the greeting. Building a `withTiming` wrapper follows the same shape as `withLogging`; the `time` package that makes it useful is covered in Chapter 14.

Tip: The middleware pattern is the foundation of HTTP handler wrappers in Go's standard library. `net/http` handlers are functions, and middleware is just a function that takes a handler and returns a new handler — you will see this in Chapter 15.

Pointer vs Value Semantics

Go structs are **value types**: assigning one struct to another copies all the fields. Java objects are always **reference types**: variables hold pointers to the object. Assigning a variable to another copies the pointer, not the object, thus both variables end up referring to the same object.

Consider a `Point` struct (covered fully in Chapter 2):

```
type Point struct {
    X, Y int
}
```

Value copy behavior:

```
a := Point{X: 3, Y: 4}
b := a              // b is a full copy
b.X = 99
```

```
fmt.Println(a.X)    // 3 --- a is unchanged
fmt.Println(b.X)    // 99
```

The same assignment in Java would have `b` and `a` pointing at the same object, so setting `b.x = 99` would also change `a.x`.

The Swap That Doesn't Work

The classic demonstration of why value semantics matters is a swap function.

```
// This does NOT work.
func swapBad(a, b int) {
    a, b = b, a // swaps the local copies only
}

func main() {
    x, y := 10, 20
    swapBad(x, y)
    fmt.Println(x, y) // 10 20 --- unchanged
}
```

`swapBad` receives copies of `x` and `y`. Swapping `a` and `b` inside the function has no effect on the caller's variables.

The Swap That Works

Pass pointers and write through them:

```
// This works.
func swap(a, b *int) {    // a and b are pointers to the caller's ints
    *a, *b = *b, *a       // dereference both and swap the values in-place
}

func main() {
    x, y := 10, 20
    swap(&x, &y)           // pass the addresses of x and y
    fmt.Println(x, y)      // 20 10 --- swapped
}
```

Tip: Go has a built-in swap idiom that makes `swap` functions rare in practice: `x, y = y, x` works directly without any function call. The pointer swap above is a teaching example; in real code, just write the one-liner.

When Mutation Requires a Pointer

The most common mistake for Java programmers moving to Go is writing a function that is supposed to modify a variable — and being surprised when nothing changes.

A struct is a value type: passing one to a function copies every field.

```
type Track struct {
    Title string
    Plays int
}

// Intended to record a play. Does not work.
func playBad(t Track) {
```

```
    t.Plays++ // modifies the local copy
}

func main() {
    song := Track{Title: "Crazy Train", Plays: 0}
    playBad(song)
    fmt.Println(song.Plays) // 0 --- still 0
}
```

The fix is to pass a pointer so the function can reach the caller's variable:

```
func play(t *Track) {
    t.Plays++ // modifies the caller's Track
}

func main() {
    song := Track{Title: "Crazy Train", Plays: 0}
    play(&song)
    fmt.Println(song.Plays) // 1
}
```

Tip: Go automatically handles the dereference when you write `t.Plays` through a pointer — you do not need to write `(*t).Plays`.

Trap: Java programmers are used to mutating object fields through a parameter because Java objects are always references. In Go, a struct parameter is a copy; mutating its fields inside the function has no effect on the caller. Always pass `*T` when the function needs to modify a struct.

Reference-Like Types

Several built-in types carry an internal pointer, so passing them by value still allows the function to mutate the underlying data — but only the data the pointer reaches, not the variable itself.

Type	What the value contains	Contents mutable without pointer?	Variable reassignable without pointer?
`map[K]V`	pointer to hash table	yes — add, delete, update entries	no
`[]T` (slice)	pointer + length + capacity	yes — modify elements	no — caller's len/cap unchanged
`chan T`	pointer to channel runtime	yes — send and receive	no
`func(...)`	pointer to code + closure env	n/a	no

The pattern is always the same: the *header* (the variable itself) is copied on every call, but it contains a pointer to shared data. Mutating through that pointer affects the original; replacing the header does not.

```
func addPlay(s []int, n int) {
    s = append(s, n) // appends to a local copy of the header --- caller sees nothing
}

func addPlayPtr(s *[]int, n int) {
    *s = append(*s, n) // replaces the caller's header
```

```
}

func main() {
    plays := []int{1, 2, 3}
    addPlay(plays, 4)
    fmt.Println(plays) // [1 2 3] --- unchanged

    addPlayPtr(&plays, 4)
    fmt.Println(plays) // [1 2 3 4]
}
```

Wut: Because a map's header is already a pointer to the hash table, you can insert and delete entries through a plain map parameter — no `*map[K]V` needed. Slices are different: a plain slice parameter lets you modify existing elements, but `append` (which may grow the backing array and update len/cap) requires a pointer to the slice header or a returned value.

Contrast with Java

In Java you can mutate the fields of an object passed as a parameter, but you cannot reassign which object the caller's variable points to:

```
// Java: mutating a field works; reassigning does not affect the caller
void bump(Counter c) {
    c.value++;       // caller sees this change
    c = new Counter(); // caller does NOT see this -- reassigns local ref only
}
```

Go's model is more consistent: nothing you do inside a function can affect a caller's variable unless the function received a pointer to it.

Escape Analysis

In C, returning a pointer to a local variable is undefined behavior — the stack frame is gone by the time the caller uses the pointer. In Go this is perfectly safe:

```
func newPoint(x, y int) *Point {
    p := Point{X: x, Y: y}   // local variable
    return &p                 // safe --- compiler handles it
}
```

The Go compiler performs **escape analysis**: it determines at compile time whether a variable's lifetime can be bounded to the current stack frame, or whether it must be allocated on the heap so that it outlives the function. If you take the address of a local variable and return it (or store it somewhere that outlives the function), the compiler silently moves the variable to the heap. You never have to make this decision yourself.

`new(T)` and `&T{}`

Two ways to allocate a pointer to a zeroed value are equivalent:

```
p1 := new(Point)      // allocates a zeroed Point; returns *Point
p2 := &Point{}        // composite literal with zero values; also returns *Point

fmt.Println(*p1 == *p2) // true --- both are zeroed Points
```

`new(T)` is the older form; `&T{}` is more idiomatic in modern Go because it lets you initialize fields at the same time:

```
p3 := &Point{X: 5, Y: 7}   // initialized and allocated in one expression
```

Both forms trigger escape analysis; neither forces the allocation onto the heap unless the pointer actually escapes.

Inspecting Escape Decisions

You can ask the compiler to show its escape analysis decisions with:

```
go build -gcflags=-m ./...
```

The output contains lines like:

```
./main.go:6:2: moved to heap: p
./main.go:9:16: &x does not escape
```

The first line tells you the variable was promoted to the heap because it escaped the function (someone kept a pointer to it). The second tells you the address was taken but the pointer never outlived the function, so the variable stayed on the stack.

Tip: You do not need to read escape analysis output in daily work. It becomes useful when profiling shows unexpected heap allocations, or when you are writing a hot inner loop and want to confirm that short-lived values are staying on the stack.

Wut: A common misconception is that `new(T)` always allocates on the heap and `:=` always allocates on the stack. Neither is true — the compiler decides based on escape analysis, not on the syntax you used. `var x int; p := &x; return p` will promote `x` to the heap even though you never wrote `new`.

Try It

Type this program in and run it. It exercises the chapter's core ideas in one place: a closure that captures `tag`, a variadic `totalPlays`, a multiple-return `average` that signals failure with an `error`, and a first-class function stored in a variable.

```
package main

import (
    "fmt"
    "strings"
)

// makeTagger returns a closure that prefixes every title with a fixed tag.
func makeTagger(tag string) func(string) string {
    return func(title string) string {
        return "[" + tag + "] " + title
    }
}

// totalPlays is variadic: sum any number of play counts.
func totalPlays(counts ...int) int {
    sum := 0
    for _, c := range counts {
        sum += c
    }
```

```
    return sum
}

// average returns the mean and an error when there is nothing to average.
func average(counts ...int) (float64, error) {
    if len(counts) == 0 {
        return 0, fmt.Errorf("no plays to average")
    }
    return float64(totalPlays(counts...)) / float64(len(counts)), nil
}

func main() {
    shout := func(s string) string { return strings.ToUpper(s) } // first-class value

    tag := makeTagger("fav")
    fmt.Println(tag(shout("Bad Bunny --- Monaco"))) // [fav] BAD BUNNY --- MONACO

    plays := []int{120, 80, 200}
    fmt.Println("total:", totalPlays(plays...)) // total: 400

    avg, err := average(plays...)
    if err != nil {
        fmt.Println("error:", err)
        return
    }
    fmt.Printf("promedio: %.1f\n", avg) // promedio: 133.3

    if _, err := average(); err != nil {
        fmt.Println("error:", err) // error: no plays to average
    }
}
```

Try these modifications:

- Add a `withCount` middleware that wraps `makeTagger`'s closure and prints how many times the tagger has been called.
- Change `average` to also return the highest play count, and update the caller to print it.
- Store several taggers (`"fav"`, `"new"`, `"skip"`) in a `map[string]func(string) string` and look one up by key.

Key Points

- Consecutive parameters of the same type can share a single type annotation: `a, b int` means both `a` and `b` are `int`.
- Go functions can return multiple values; the convention is to return `(result, error)` rather than throwing exceptions.
- Named return values act as pre-declared variables; a naked `return` returns their current values.
- Named returns allow `defer` closures to inspect or modify the return values — useful for wrapping close errors.
- Variadic functions use `...T` for the last parameter; inside the function it is a `[]T`.
- Pass an existing slice to a variadic function with `slice...`.
- Functions are first-class values: assign them to variables, store them in maps, pass them as arguments.
- A closure captures variables from its enclosing scope by reference.
- Since Go 1.22, loop variables are per-iteration, eliminating the classic closure capture bug.

- `init()` runs before `main`, after package-level variables are initialized; multiple `init` functions per file are allowed; they cannot be called explicitly.
- The Java analogue of `init` is a static initializer block.
- Passing functions as parameters enables callbacks and the middleware pattern.
- Go structs are value types — assignment copies all fields; Java objects are reference types — assignment copies only the reference.
- A function that receives a value parameter cannot modify the caller's variable; pass a pointer to allow mutation.
- The Go compiler uses escape analysis to decide whether a variable lives on the stack or the heap; you do not manage this manually.
- `new(T)` and `&T{}` are equivalent ways to allocate a zeroed value; `&T{}` is more idiomatic.
- `go build -gcflags=-m` shows escape analysis decisions.

Exercises

1. **Think about it:** Go returns errors as values rather than throwing exceptions. A Java `checked exception` forces the caller to handle it — the compiler will not let you ignore it. Go's multi-return error is also explicit, but you can discard it with _ or simply not assign the second return value. Does Go's approach give you the same safety guarantee as Java's checked exceptions? What is gained and what is lost by each approach?

2. **What does this print?**

```go
package main

import "fmt"

func makeAdder(n int) (func() int, func() int) {
    inc := func() int { n++; return n }
    dec := func() int { n--; return n }
    return inc, dec
}

func main() {
    inc, dec := makeAdder(5)
    fmt.Println(inc())
    fmt.Println(inc())
    fmt.Println(dec())
    fmt.Println(dec())
}
```

3. **Calculation:** Given the function below, what values are printed by the three `fmt.Println` calls? Trace the value of `total` at each step.

```go
package main

import "fmt"

func running(start int) func(int) int {
    total := start
    return func(n int) int {
        total += n
        return total
    }
}
```

```
func main() {
    acc := running(100)
    fmt.Println(acc(10))
    fmt.Println(acc(20))
    fmt.Println(acc(-5))
}
```

4. **Where is the bug?** The following code tries to build a slice of greeting functions, one for each name in a list, using a Go 1.21 module (i.e. the `go` directive in `go.mod` is `go 1.21`).

```
package main

import "fmt"

func main() {
    names := []string{"benson", "amara", "priya"}
    greets := make([]func(), len(names))
    for i, name := range names {
        greets[i] = func() { fmt.Println("hola,", name) }
    }
    for _, g := range greets {
        g()
    }
}
```

5. **Write a program:** Write a function `pipeline(fns ...func(int) int) func(int) int` that takes any number of `func(int) int` functions and returns a new function that applies them in order. For example, given a double function and an add-ten function, `pipeline(double, addTen)(3)` should return `16`. Write the function, define at least two simple transforms, and demonstrate the pipeline with a few calls.

Chapter 6

Objects using Methods and Embedding

Java bundles data, behavior, and code reuse together in a single class construct. Go separates them into three distinct mechanisms: structs hold data, methods attach behavior to structs, and embedding provides composition-based code reuse. This chapter covers all three, explains how to write constructors and handle resource cleanup without the language features Java provides for those tasks, and shows why Go's "no inheritance" stance is a strength rather than a limitation.

Receivers and Methods

Like Java, Go lets you attach methods to a type, but it declares them quite differently. Consider the following class:

```
class Point {
    int x;
    int y;
    void scale(int factor) { x *= factor; y *= factor; }
}
```

In Java you define the member variables of a class in the same place you define all the methods of a class. Go does it differently. You define the member variables with `struct`, and the methods are declared later to operate on that structure using *receivers* that have the following forms:

```
func (t T) MethodName(params) returnType    // value receiver --- gets a copy of T
func (t *T) MethodName(params) returnType   // pointer receiver --- gets a pointer to T
```

Method declarations look just like normal function declarations except for the receiver variables, shown as `t` in the above examples. These receiver variables act like the `this` pointer in Java with two big differences. First, you can pick the name. Go best practices say that it should be a one- or two-letter abbreviation of the type name. [***receiver-name-abbreviation***] Second, a receiver can act on a pointer to an object — like `this` in Java — or it can act on a copy of the object. The former is called a pointer receiver, the latter a value receiver.

Using `Point` again as a minimal example:

```
type Point struct {
    X, Y int
}

// Value receiver --- Scale gets a copy of the Point.
func (p Point) ScaleBad(factor int) {
    p.X *= factor   // modifies the copy only
    p.Y *= factor
```

```
}

// Pointer receiver --- Scale gets a pointer to the original Point.
func (p *Point) Scale(factor int) {
    p.X *= factor   // modifies the original
    p.Y *= factor
}

pt := Point{X: 3, Y: 4}

pt.ScaleBad(10)
fmt.Println(pt) // {3 4} --- unchanged; ScaleBad worked on a copy

pt.Scale(10)
fmt.Println(pt) // {30 40} --- changed; Scale worked on the original
```

Go is convenient here: even though `pt` is a plain `Point` value (not a `*Point`), you can call `pt.Scale(10)` and Go automatically takes the address for you — it is equivalent to `(&pt).Scale(10)`.

Trap: The `Point` example above deliberately mixes a value receiver (`ScaleBad`) with a pointer receiver (`Scale`) to show the behavioral difference. In real code, `Point` would have `Scale` as its only method (or all methods would use `*Point`). The name `ScaleBad` is the hint: drop the value-receiver `ScaleBad` and use only pointer receivers.

Tip: If any method on a type needs a pointer receiver, make all methods on that type pointer receivers. Mixing value and pointer receivers on the same type confuses the method set rules and makes interface satisfaction harder to reason about. This is a firm Go idiom worth following from day one. [***no-mixed-receivers***]

Wut: If you call a pointer receiver method through a value that is not addressable — for example, the return value of a function call used directly — Go cannot take the address and the compiler reports an error. Chapter 2 introduced & and pointers; a value is addressable when you could legally apply & to it, which is why a bare function return value does not qualify.

The same rules apply to any type. Here is a `Track` type with two pointer-receiver methods:

```
func (t *Track) String() string {              // pointer receiver: consistent with ScaleBPM
    return fmt.Sprintf("%s by %s (%d BPM)", t.Title, t.Artist, t.BPM)
}

func (t *Track) ScaleBPM(factor float64) {     // pointer receiver: mutates BPM
    t.BPM = int(float64(t.BPM) * factor)
}
```

`ScaleBPM` mutates `BPM`, so it needs `*Track`. [***pointer-receiver-for-mutation***] `String` uses `*Track` too, even though it only reads, to keep receivers consistent.

Unlike Java, where all methods of a class must live inside that class's file, Go methods can be declared in any file within the same package — which means the same directory. A single file can also define structs and methods for several different types. There is no rule that says `track.go` must contain only `Track`-related code. The constraint is the package boundary: a type and its methods must all be in the same package, but they can be spread across as many files as you like.

How you split code across files is a style question. The Google Go Style Guide recommends keeping files focused and reasonably short: "files should be focused enough that a maintainer can tell which file contains

something, and the files should be small enough that it will be easy to find once there" (Google 2024). A common pattern is one file per major type, with its closely related helpers alongside it.

Method Sets

Every type has a **method set** — the set of methods you can call on a value of that type.

Value type	Methods callable
`T`	only value-receiver methods of `T`
`*T`	all methods of `T` (both value and pointer receivers)

A `*T` pointer can call everything: pointer-receiver methods (directly) and value-receiver methods (Go automatically dereferences). A plain `T` value can only call value-receiver methods.

Wut: You might expect that `T` can call pointer-receiver methods too, because Go auto-takes addresses for you when you write `t.ScaleBPM(1.1)` on an addressable variable. It can — but only for *calls*, not for interface satisfaction. When you store a `T` (not `*T`) in an interface, only the value-receiver methods are in the method set. This distinction matters in Chapter 8.

Calling Methods

Go is flexible about how you call methods on values vs pointers:

```
t := Track{Title: "Flaming June", Artist: "BT", BPM: 118}

// Calling a pointer-receiver method on an addressable value --- Go auto-takes address.
fmt.Println(t.String())    // equivalent to (&t).String() --- Flaming June by BT (118 BPM)
t.ScaleBPM(1.1)            // equivalent to (&t).ScaleBPM(1.1)
fmt.Println(t.BPM)         // 129

// Calling a pointer-receiver method on a pointer --- straightforward.
p := &t
fmt.Println(p.String())    // Flaming June by BT (129 BPM)
```

Go's auto-dereference and auto-address rules mean you rarely write `(&t).Method()` or `(*p).Method()` yourself.

Attaching Methods to the Playlist

Using the `Track` type from Chapter 2 plus a new `Playlist` type, here is a complete example of both receiver kinds in one type:

```
type Track struct {
    Title    string
    Artist   string
    BPM      int
    Duration float64 // seconds
}

type Playlist struct {
    Name   string
    tracks []Track
}
```

```go
func (p *Playlist) Add(t Track) {          // pointer receiver: mutates the slice
    p.tracks = append(p.tracks, t)
}

func (p *Playlist) Len() int {             // pointer receiver: consistent with Add
    return len(p.tracks)
}

func (p *Playlist) AverageBPM() float64 { // pointer receiver: consistent with Add
    if len(p.tracks) == 0 {
        return 0
    }
    total := 0
    for _, t := range p.tracks {
        total += t.BPM
    }
    return float64(total) / float64(len(p.tracks))
}

pl := Playlist{Name: "Late Night Vibes"}
pl.Add(Track{Title: "Flaming June", Artist: "BT", BPM: 118})
pl.Add(Track{Title: "Emerald Triangle 2012", Artist: "Angoscia", BPM: 127})
pl.Add(Track{Title: "Gamemaster", Artist: "Matt Darey & Lost Tribe", BPM: 97})
fmt.Println(pl.Len())                       // 3
fmt.Printf("%.1f BPM\n", pl.AverageBPM())  // 114.0 BPM
```

Tip: All three methods use pointer receivers because `Add` needs one, and the rule is: if any method on a type needs a pointer receiver, make them all pointer receivers. `tracks` is also unexported (lowercase), so callers outside this package can only grow the playlist through `Add`, which lets you add validation later without breaking the public API.

Methods on Non-Struct Types

This is where Go and Java part ways. In Java, methods can only live inside a class. A bare `int` or `String` can never grow a method of your own. Go has no such rule: you can attach a method to *any* named type, not just a struct. A named numeric type, a named string type, even a named slice or map type can have methods.

The only requirement is that you define the named type yourself, in the same package as the method. Here is a `BPM` type whose underlying type is `int`, with a method that classifies the tempo:

```go
type BPM int

// Value receiver --- Genre just reads the number.
func (b BPM) Genre() string {
    switch {
    case b < 100:
        return "downtempo"
    case b < 125:
        return "house"
    default:
        return "trance"
    }
}
```

```
b := BPM(118)
fmt.Println(b.Genre()) // house
fmt.Println(b + 10)    // 128 --- still behaves like an int
```

`BPM` keeps all the arithmetic behavior of its underlying `int` — you can add, compare, and print it like a number — but it also carries the `Genre` method you defined. A value receiver is fine here because `Genre` only reads `b`; reach for a pointer receiver only when a method mutates the value.

Named slice and map types work the same way, and this is how you give a collection its own behavior without wrapping it in a struct:

```
type Tracklist []Track

func (tl Tracklist) TotalSeconds() float64 {
    var total float64
    for _, t := range tl {
        total += t.Duration
    }
    return total
}
```

Even a *function* type can have methods. This sounds exotic, but it is the trick behind one of Go's most common idioms: adapting a plain function so it satisfies an interface — the **adapter** or **strategy** pattern from Java, with no class boilerplate. Define a named function type, give it the method the interface requires, and the method body simply calls the function value (its own receiver):

```
type TrackFilter func(Track) bool

// Negate returns a filter that accepts exactly the tracks f rejects.
func (f TrackFilter) Negate() TrackFilter {
    return func(t Track) bool { return !f(t) }
}

var isLong TrackFilter = func(t Track) bool { return t.Duration > 300 }
isShort := isLong.Negate()
fmt.Println(isShort(Track{Duration: 120})) // true
```

The standard library uses exactly this pattern: `http.HandlerFunc` is a named function type with a `ServeHTTP` method, which lets an ordinary function stand in wherever an `http.Handler` interface value is expected (Chapter 15).

Trap: You can only attach a method to a type that is defined in the current package. Writing `func (i int) Double() int` is a compile error (*cannot define new methods on non-local type int*), and so is attaching a method directly to `time.Duration` or any other type from another package. The fix is the same in both cases: define your own named type (`type BPM int`) and hang the method on that.

Constructors

Go has **no constructor syntax**. There is no `new SomeClass(...)` keyword, no `__init__` method, and no special function that runs automatically when a struct is created.

A zero-value struct is valid on its own — that is by design. When you need a struct that starts in a specific non-zero state, or when you want to validate inputs at creation time, the idiomatic Go replacement is a **`New*` factory function**.

The New* Pattern

By convention, a factory function is named `New` followed by the type name. `go doc` and IDE tools recognize this pattern and display it alongside the type.

```
func NewPlaylist(name string) *Playlist {
    return &Playlist{Name: name}
}
```

The return type is `*Playlist` rather than `Playlist` so the caller gets a pointer and can immediately call pointer-receiver methods like `Add`. [*return-concrete-types*]

```
pl := NewPlaylist("Late Night Vibes")
pl.Add(Track{Title: "Flaming June", Artist: "BT", BPM: 118})
```

Compare with Java:

```
// Java
Playlist pl = new Playlist("Late Night Vibes");

// Go
pl := NewPlaylist("Late Night Vibes")
```

The call sites look nearly identical. The Go version is a plain function call with no special language support — but that is all you need.

Constructors That Validate

Factory functions can return an `error` when the inputs are invalid. A Java constructor throws an exception; a Go factory function returns the error as a second value. Error strings should be lowercase and not end with punctuation [*lowercase-error-strings*], as the `NewTrack` example below demonstrates.

```
func NewTrack(title, artist string, bpm int) (Track, error) {
    if title == "" {
        return Track{}, fmt.Errorf("newTrack: title must not be empty")
    }
    if artist == "" {
        return Track{}, fmt.Errorf("newTrack: artist must not be empty")
    }
    if bpm <= 0 || bpm > 300 {
        return Track{}, fmt.Errorf("newTrack: BPM %d is out of range [1, 300]", bpm)
    }
    return Track{Title: title, Artist: artist, BPM: bpm}, nil
}

t, err := NewTrack("Emerald Triangle 2012", "Angoscia", 127)
if err != nil {
    log.Fatal(err)
}
fmt.Println(t.Title) // Emerald Triangle 2012
```

Tip: Return `(T, error)` — a value, not a pointer — when the zero value of `T` is harmless and `T` is small. Return `(*T, error)` when the caller needs to call pointer-receiver methods immediately after construction, or when `T` is large enough that you want to avoid copying it.

Wut: In Java, throwing in a constructor is the only way to signal a construction failure. In Go, a factory function returns `(nil, err)` or `(T{}, err)`. There is no special "failed construction" state — the error is just a value you check like any other.

Destructors

Go has **no destructor syntax**. There is no `finalize()`, no `__del__`, and no `~ClassName()`. The garbage collector reclaims heap memory automatically; you do not manage object lifetimes.

For resources that need explicit cleanup — open files, network connections, mutex locks, database transactions — Go's answer is `defer`.

defer for Cleanup

`defer` was introduced in Chapter 4. The key rule: a deferred call runs when the surrounding function returns, in LIFO order. This makes it the idiomatic replacement for Java's try-with-resources.

The standard open/close pattern:

```
f, err := os.Open(path)
if err != nil {
    return err
}
defer f.Close()  // runs when the enclosing function returns, no matter how
// ... use f ...
```

The `defer f.Close()` line is written immediately after the successful open, before any logic that might return early on error. That placement ensures `f.Close()` is called on every path out of the function.

Compare the patterns side by side:

```
// Java: try-with-resources
try (FileReader f = new FileReader(path)) {
    // use f --- close is called automatically on exit
} catch (IOException e) {
    // handle error
}
```

```
// Go: defer
f, err := os.Open(path)
if err != nil {
    return err
}
defer f.Close()
// use f --- Close is called automatically on return
```

Both guarantee that the resource is released even if the body returns early. The Go form scales naturally to multiple resources:

```
db, err := sql.Open("postgres", dsn)
if err != nil {
    return err
}
defer db.Close()

tx, err := db.Begin()
if err != nil {
```

```
    return err
}
defer tx.Rollback() // rolls back only if Commit has not been called
```

Trap: The arguments to a deferred call — including the receiver — are evaluated at the moment the `defer` statement executes, not when the deferred function actually runs. Wrapping the call in a closure defers the evaluation of any captured variables until the function returns, which matters when you reassign one of those variables afterward:

```
f, _ := os.Open("a.txt")
defer f.Close()          // captures THIS f right now

f, _ = os.Open("b.txt")   // reassigns f
// the deferred call still closes a.txt, not b.txt
```

The closure form reads the variable at return time instead:

```
f, _ := os.Open("a.txt")
defer func() { f.Close() }() // reads f when main returns

f, _ = os.Open("b.txt")      // reassigns f
// now the deferred call closes b.txt
```

Wut: The Go documentation writes `defer f.Close()` everywhere, yet that very pattern violates the *Errors are values* proverb — it throws away the error that `Close` returns. For a file you only read from, ignoring it is usually fine: you already have the bytes you wanted. But some errors surface *only* on `Close`. A buffered writer can hold data that is not flushed until `Close` runs, and even for a plain `os.File` some filesystems report a failed write only when the file is closed — `defer f.Close()` would silently swallow it. When the close can lose data, capture the error instead of deferring it blindly: assign it to a named return value from a deferred closure, or call `Close` explicitly on the happy path.

```
func save(path string, data []byte) (err error) {
    f, err := os.Create(path)
    if err != nil {
        return err
    }
    defer func() {
        if cerr := f.Close(); cerr != nil && err == nil {
            err = cerr // surface a close failure the caller would miss
        }
    }()
    _, err = f.Write(data)
    return err
}
```

runtime.SetFinalizer

The `runtime` package has `SetFinalizer(obj, finalizer)`, which registers a function to run when the GC is about to collect `obj`. It exists for interoperability with C libraries that require explicit deallocation, and for similar low-level needs.

In almost every other situation, `SetFinalizer` is the wrong tool. Finalizers run at an unpredictable time, may never run at all if the program exits normally, and interact poorly with GC tuning.

Trap: Do not use `runtime.SetFinalizer` to release file handles, locks, or network connections. Use `defer` instead — it is deterministic, runs immediately on function exit, and is far easier to reason about.

Embedding

Go has no inheritance. The mechanism for code reuse between types is **embedding**: you include one struct inside another by naming only the type, without a field name.

The fields and methods of the embedded type are **promoted** to the outer type — you can access them directly as if they were declared on the outer type itself.

Field and Method Promotion

```
type Artist struct {
    Name    string
    Country string
}

func (a Artist) Label() string {
    return fmt.Sprintf("%s (%s)", a.Name, a.Country)
}

type Song struct {
    Artist        // embedded --- no field name
    Title  string
    BPM    int
}

s := Song{
    Artist: Artist{Name: "Angoscia", Country: "Italy"},
    Title:  "Emerald Triangle 2012",
    BPM:    127,
}

fmt.Println(s.Name)    // Angoscia  --- promoted from Artist
fmt.Println(s.Country) // Italy    --- promoted from Artist
fmt.Println(s.Label()) // Angoscia (Italy) --- promoted method
fmt.Println(s.Title)   // Emerald Triangle 2012 --- own field
```

You can still reach the embedded struct directly by its type name when you need to:

```
fmt.Println(s.Artist.Name) // Angoscia --- explicit path
```

The explicit path is also how you distinguish between an outer field and an embedded field when there is a name collision — covered below.

Embedded Value vs Embedded Pointer

You can embed either a value or a pointer to a type:

```
type Song struct {
    Artist        // embedded value --- Song owns the Artist data
    Title  string
}
```

```
type SongRef struct {
    *Artist       // embedded pointer --- SongRef borrows Artist data
    Title  string
}
```

Use an **embedded value** when the outer struct fully owns the embedded data and you want simple value-copy semantics.

Use an **embedded pointer** when multiple outer structs share the same embedded object, or when the embedded type is large and you want to avoid copying it.

Trap: A zero-value `SongRef` has a `nil` `*Artist` pointer. Accessing any promoted field or method on a nil embedded pointer causes a runtime panic. Always initialize the embedded pointer before use:

```
sr := SongRef{
    Artist: &Artist{Name: "Angoscia", Country: "Italy"},
    Title:  "Emerald Triangle 2012",
}
```

Name Collisions

If two embedded types define a field or method with the same name, the compiler reports an ambiguity error the moment you try to access the name without qualification.

```
type Meta struct {
    Title string
}

type Track struct {
    Meta          // has Title
    Title  string // also has Title
}

t := Track{Meta: Meta{Title: "metadata title"}, Title: "track title"}
fmt.Println(t.Title)       // "track title" --- outer field wins (no ambiguity here)
fmt.Println(t.Meta.Title)  // "metadata title" --- must qualify to reach embedded field
```

When the outer type itself declares `Title`, that field shadows the promoted one — no ambiguity, outer wins. When two embedded types both promote the same name and the outer type does not declare it, the compiler errors on any unqualified access:

```
type A struct{ Val int }
type B struct{ Val int }

type C struct {
    A
    B
}

c := C{}
fmt.Println(c.Val)   // compile error: ambiguous selector c.Val
fmt.Println(c.A.Val) // OK
fmt.Println(c.B.Val) // OK
```

Embedding vs Inheritance

Java's class inheritance is an *is-a* relationship: a `FeaturedTrack` that extends `Track` is substitutable wherever a `Track` is expected. Go embedding is a *has-a* relationship: a `FeaturedTrack` that embeds `Track` gets `Track`'s promoted fields and methods, but is not substitutable for a `Track`.

Side-by-Side Comparison

```java
// Java: inheritance
class FeaturedTrack extends Track {
    String feature;

    FeaturedTrack(String title, String artist, int bpm, String feature) {
        super(title, artist, bpm);
        this.feature = feature;
    }

    @Override
    public String toString() {
        return super.toString() + " ft. " + feature;
    }
}

// A FeaturedTrack IS-A Track --- substitutable.
Track t = new FeaturedTrack("Gamemaster", "Matt Darey & Lost Tribe", 97, "Alizée");
```

```go
// Go: embedding
type FeaturedTrack struct {
    Track              // has-a Track, not is-a Track
    Feature string
}

func (ft *FeaturedTrack) String() string {
    return ft.Track.String() + " ft. " + ft.Feature
}
```

`FeaturedTrack.String` uses a pointer receiver because the embedded `Track.String` does, and the all-pointer-receiver idiom applies to the outer type too.

```go
ft := FeaturedTrack{
    Track:   Track{Title: "Gamemaster", Artist: "Matt Darey & Lost Tribe", BPM: 97},
    Feature: "Alizée",
}

fmt.Println(ft.Title)    // Gamemaster --- promoted
fmt.Println(ft.String()) // Gamemaster by Matt Darey & Lost Tribe (97 BPM) ft. Alizée
```

But this does **not** compile:

```go
var t Track = ft  // compile error: cannot use FeaturedTrack as Track
```

A `FeaturedTrack` is not a `Track`. It has a `Track` inside, but Go's type system does not consider that inheritance.

What You Get and What You Don't

What embedding gives you:

- Promoted fields: `ft.Title`, `ft.Artist`, `ft.BPM` work without qualification.

- Promoted methods: `ft.ScaleBPM(1.05)` delegates to the embedded `Track`'s method.
- Less boilerplate: no need to write forwarding methods by hand.

What embedding does not give you:

- Substitutability: a `FeaturedTrack` cannot be passed where a `Track` is expected.
- Virtual dispatch: there is no override mechanism; the outer type's method simply shadows the inner one.

How Interfaces Fill the Gap

Go fills the polymorphism gap with interfaces (covered in Chapter 8). If both `Track` and `FeaturedTrack` implement the same interface, they can be used interchangeably through that interface — regardless of their struct relationship. Method promotion has a powerful consequence here: because an embedded type's methods are promoted to the outer type, the outer type automatically satisfies any interface the embedded type satisfies (see Chapter 8) — you can embed a type purely to inherit its interface implementation for free.

```
type Playable interface {
    String() string
}

// Both *Track and *FeaturedTrack provide String() ---
// both satisfy Playable, independently.
func announce(p Playable) {
    fmt.Println("Now playing:", p.String())
}

announce(&t)   // *Track satisfies Playable
announce(&ft)  // *FeaturedTrack also satisfies Playable
```

Tip: The Go proverb is "favor composition over inheritance." Go takes that further: composition via embedding is the *only* option. Once you internalize it, you will find the explicit has-a relationship easier to reason about than deep inheritance hierarchies.

A Realistic Embedding Example: LoggedPlaylist

Embedding shines when you want to extend the behavior of an existing type without modifying it. A common pattern is to wrap a type with one that adds cross-cutting concerns — logging, metrics, caching — by embedding the original and selectively overriding methods.

```
import (
    "fmt"
    "log"
)

type LoggedPlaylist struct {
    Playlist                    // embeds all of Playlist's fields and methods
}

func NewLoggedPlaylist(name string) *LoggedPlaylist {
    return &LoggedPlaylist{Playlist: Playlist{Name: name}}
}

// Add wraps Playlist.Add with a log line.
func (lp *LoggedPlaylist) Add(t Track) {
```

```
    log.Printf("adding %q to %q", t.Title, lp.Name)
    lp.Playlist.Add(t) // delegate to the embedded method
}

lp := NewLoggedPlaylist("Late Night Vibes")
lp.Add(Track{Title: "Flaming June", Artist: "BT", BPM: 118})
// 2026/05/28 00:00:00 adding "Flaming June" to "Late Night Vibes"

fmt.Println(lp.Len())          // 1  --- promoted from Playlist
fmt.Println(lp.AverageBPM())   // 118 --- promoted from Playlist
```

`LoggedPlaylist` inherits `Len` and `AverageBPM` for free via promotion. It only needs to define `Add` — the one method it wants to wrap.

Trap: When `LoggedPlaylist.Add` calls `lp.Playlist.Add(t)`, it must use the explicit qualified path `lp.Playlist.Add`. Writing `lp.Add(t)` inside the method would call `LoggedPlaylist.Add` recursively and loop forever.

Try It

Type this in and run it: it pulls together every idea in this chapter — pointer receivers, a `New*` factory, embedding via promotion, a wrapped method, and `defer` for end-of-function cleanup. The `LoggedPlaylist` embeds a `*Playlist`, gets `Len` for free, and overrides only `Add`.

```
package main

import (
    "fmt"
    "log"
)

type Track struct {
    Title  string
    Artist string
    BPM    int
}

func (t *Track) String() string { // pointer receiver, consistent across the type
    return fmt.Sprintf("%s by %s (%d BPM)", t.Title, t.Artist, t.BPM)
}

type Playlist struct {
    Name   string
    tracks []*Track
}

func NewPlaylist(name string) *Playlist { // New* factory in place of a constructor
    return &Playlist{Name: name}
}

func (p *Playlist) Add(t *Track) { // pointer receiver: mutates the slice
    p.tracks = append(p.tracks, t)
}
```

```
func (p *Playlist) Len() int { // pointer receiver for consistency
    return len(p.tracks)
}

type LoggedPlaylist struct {
    *Playlist // embedded pointer: promotes Len and friends
}

func (lp *LoggedPlaylist) Add(t *Track) { // wraps the embedded Add
    log.Printf("queueing %q", t.Title)
    lp.Playlist.Add(t) // explicit path avoids infinite recursion
}

func main() {
    defer fmt.Println("done") // runs last, like a cleanup step

    lp := &LoggedPlaylist{Playlist: NewPlaylist("Fiesta")}
    lp.Add(&Track{Title: "Emerald Triangle 2012", Artist: "Angoscia", BPM: 127})
    lp.Add(&Track{Title: "Gamemaster", Artist: "Matt Darey & Lost Tribe", BPM: 97})

    fmt.Println("tracks:", lp.Len()) // promoted from Playlist
    for _, t := range lp.tracks {
        fmt.Println(t) // calls (*Track).String automatically
    }
}
```

The `log` lines carry a timestamp, but the rest is deterministic: a `tracks: 2` count, the two tracks formatted through `String`, and `done` printed last by the deferred call.

Try these modifications:

- Add an `AverageBPM` method on `*Playlist` and call it through `lp` — notice it is promoted for free.
- Change `LoggedPlaylist.Add` to mistakenly call `lp.Add(t)` instead of `lp.Playlist.Add(t)` and watch the stack overflow.
- Swap the embedded `*Playlist` for a value `Playlist` and see which calls still compile.

Key Points

- A method is a function with a receiver declared between `func` and the method name.
- A value receiver `(t T)` gets a copy; a pointer receiver `(t *T)` gets a pointer to the original and can mutate it.
- If any method on a type uses a pointer receiver, use pointer receivers for all methods on that type.
- The method set of `*T` includes all methods of `T`; the method set of `T` includes only value-receiver methods.
- Go auto-takes the address for pointer-receiver calls on addressable values and auto-dereferences for value-receiver calls on pointers.
- Go has no constructor syntax; the idiomatic replacement is a `New*` factory function.
- Factory functions return `*T` so the caller can immediately use pointer-receiver methods.
- Factory functions can return `(T, error)` or `(*T, error)` to signal construction failures.
- `go doc` and IDE tooling recognize the `New*` naming convention.
- Go has no destructor syntax; the GC reclaims heap memory automatically.
- Use `defer resource.Close()` immediately after a successful resource acquisition for deterministic cleanup.
- `defer` is Go's equivalent of Java's try-with-resources; multiple defers form a LIFO cleanup stack.
- `runtime.SetFinalizer` exists but is rarely correct; prefer `defer`.

- Embedding includes one struct type inside another by naming only the type, with no field name.
- Embedded fields and methods are promoted to the outer type; you can access them without qualification.
- An embedded value gives full ownership; an embedded pointer shares the embedded data.
- A zero-value struct with an embedded pointer has a nil embedded pointer; accessing promoted members panics.
- When two embedded types have the same field or method name and the outer type does not shadow it, any unqualified access is a compile error; resolve it with an explicit path.
- Go embedding is a has-a relationship, not an is-a relationship; a `FeaturedTrack` cannot be used where a `Track` is expected.
- Interfaces fill the polymorphism gap: any type that provides the required methods satisfies the interface, regardless of its embedding structure.
- "Favor composition over inheritance" is the Go proverb; embedding is the language's only code-reuse mechanism.

Exercises

1. **Think about it:** In Java, a class bundles data and behavior together and inheritance lets you share both across a type hierarchy. Go separates data (struct), behavior (methods), and code reuse (embedding) into three distinct mechanisms, and interfaces handle polymorphism independently of all three. What advantages does Go's separated approach offer over Java's unified class model? Can you think of a scenario where Java's approach is simpler or more convenient?

2. **What does this print?**

```go
package main

import "fmt"

type Base struct {
    ID int
}

func (b Base) Describe() string {
    return fmt.Sprintf("Base ID=%d", b.ID)
}

type Widget struct {
    Base
    Color string
}

func main() {
    w := Widget{
        Base:  Base{ID: 42},
        Color: "blue",
    }
    fmt.Println(w.ID)
    fmt.Println(w.Color)
    fmt.Println(w.Describe())
    fmt.Println(w.Base.Describe())
}
```

3. **Calculation:** Given the types below, count the methods in each method set.

```
type Track struct {
    Title  string
    Artist string
}

func (t *Track) String() string     { /* ... */ }
func (t *Track) ScaleBPM(f float64) { /* ... */ }
func (t Track) IsLong() bool        { /* ... */ }

type FeaturedTrack struct {
    Track
    Feature string
}

func (ft *FeaturedTrack) String() string { /* ... */ }
```

a. How many methods are in the method set of `Track` (the value type)?
b. How many methods are in the method set of `*Track`?
c. How many methods are in the method set of `FeaturedTrack` (the value type), counting promoted methods?
d. How many methods are in the method set of `*FeaturedTrack`, counting promoted methods?

4. **Where is the bug?** The following program panics at runtime. Identify the exact line that panics, explain why, and describe how to fix it.

```
package main

import "fmt"

type Artist struct {
    Name string
}

func (a Artist) Label() string {
    return "Artist: " + a.Name
}

type Song struct {
    *Artist
    Title string
}

func main() {
    s := Song{Title: "Out Of The Blue"}
    fmt.Println(s.Title)
    fmt.Println(s.Label()) // line A
}
```

5. **Write a program:** Define a struct `Counter` with a single `int` field `Value`. Write a `New*` constructor that accepts a starting value and returns a `*Counter`. Add three pointer-receiver methods: `Increment()` that adds 1, `Reset()` that sets `Value` to zero, and `String() string` that returns the current value formatted as `"count: N"`. In `main`, create a `Counter` with `NewCounter(10)`, increment it three times, print it, reset it, and print it again. Use `defer` to print `"done"` at the end of `main`, so that "done" appears as the very last line of output.

6. **Where is the bug?** The following program does not compile. Explain the exact reason the compiler

rejects it, and describe the smallest change that makes it work.

```
package main

import "fmt"

func (n int) IsFast() bool {
    return n >= 125
}

func main() {
    bpm := 128
    fmt.Println(bpm.IsFast())
}
```

Chapter 7

Maps and Slices

If you are coming from Java, two data structures will carry the weight of almost every program you write in Go: maps and slices. Maps replace `HashMap<K,V>` — same O(1) lookup, far less ceremony. Slices replace `ArrayList<E>` — same dynamic growth, but backed by a plain array and built directly into the language. Without a solid grasp of these two, simple things turn painful: you cannot iterate a collection in deterministic order, you write a value into a map and the program panics, or you slice off "just the first two elements" and silently corrupt the original. A Java programmer expects `ArrayList` to be a self-contained object and `HashMap` to hide its storage; Go exposes the seams, and the aliasing behavior that follows from those exposed seams is the single biggest source of surprise bugs in early Go code. Getting maps and slices right up front saves you from chasing those bugs in every later chapter. This chapter covers maps first (key–value lookup), then slices (dynamic sequences), because maps tend to surprise Java programmers more and are worth getting right before the subtleties of slice aliasing land on top.

Maps

A map in Go is an unordered collection of key–value pairs. The Java equivalent is `HashMap<K,V>`. Like `HashMap`, Go maps offer O(1) average-case lookup, insertion, and deletion. Unlike `HashMap`, Go maps are a built-in type with dedicated syntax.

Map Type and Declaration

The type of a map with keys of type `K` and values of type `V` is written `map[K]V`.

```
var plays map[string]int // nil map --- not yet initialized
```

`K` must be **comparable**: any type that supports `==` and `!=`. Strings, integers, floats, booleans, pointers, channels, arrays of comparable elements, and structs whose fields are all comparable are valid key types. Slices, maps, and functions are **not** comparable and cannot be used as keys.

Wut: Java lets you use any object as a `HashMap` key as long as it implements `hashCode()` and `equals()`. Go requires the key type to support `==` at the language level. You cannot use a `[]byte` as a Go map key; convert it to `string` first.

There are two ways to create a non-nil map.

Map literal:

```
streams := map[string]int{
    "Saltwater":       1_200_000_000, // Chicane
    "Out Of The Blue": 980_000_000,   // System F
```

```
    "Gamemaster":      750_000_000,   // Matt Darey & Lost Tribe
}
```

make:

```
plays := make(map[string]int) // empty, ready to use
```

`make(map[K]V)` returns an initialized, empty map. You can also pass a size hint — `make(map[K]V, 100)` — which pre-allocates internal buckets but does not set a maximum size.

Use `make` when you need an empty map, possibly pre-sized. Use a literal when you have initial values — literals also compose naturally for nested maps:

```
grades := map[string]map[string]int{
    "CMPE 30": {
        "Ben":  22,
        "Amy":  88,
        "Fred": 32,
    },
    "CMPE 50": {
        "Qi":  90,
        "Cal": 102,
    },
}
```

Trap: A `var` declaration without an initializer gives you a **nil map**. Reading from a nil map is safe and returns the zero value. **Writing to a nil map panics at runtime.**

```
var m map[string]int
fmt.Println(m["key"]) // fine --- prints 0
m["key"] = 1          // panic: assignment to entry in nil map
```

Always initialize with a literal or `make` before writing.

Map Operations

The four basic operations on a map are read, write, delete, and count.

```
m := map[string]int{
    "cumbia":    92,
    "reggaeton": 98,
    "afrobeats": 120,
}

fmt.Println(m["cumbia"])   // 92       --- read
m["drill"] = 140           //          --- write
delete(m, "reggaeton")     //          --- delete
fmt.Println(len(m))        // 3        --- count: cumbia, afrobeats, drill
```

The signatures of the relevant built-in functions are:

```
func delete(m map[K]V, k K)  // remove the entry with key k; no-op if k is absent
func len(v Type) int         // number of entries in a map (or length of slice/string/channel)
```

Reading a key that is not in the map does **not** panic or throw. It returns the zero value for the value type:

```
bpm := m["classical"] // 0 --- key not present, zero value returned
```

This is safe but can hide bugs: you cannot tell whether `bpm` is `0` because classical music has no tempo entry, or because someone explicitly stored `0`. The comma-ok idiom (next section) resolves that ambiguity. [*no-in-band-errors*]

The Comma-ok Idiom

To test whether a key is actually present in a map, use the two-value assignment form:

```
v, ok := m[k] // v is the value (or zero), ok is true only if k exists
```

`ok` is a `bool`. It is `true` if the key was found, `false` if it was not. This is idiomatic Go; you will see it everywhere.

```
catalog := map[string]string{
    "Emerald Triangle 2012": "Angoscia",
    "Better Off Alone":      "DJ Cobra",
    "Children":              "Robert Dream House",
}

if artist, ok := catalog["Better Off Alone"]; ok {
    fmt.Println("Found:", artist) // Found: DJ Cobra
} else {
    fmt.Println("Not in catalog")
}
```

Tip: The Java idiom is `if (map.containsKey(k)) { v = map.get(k); }` — two lookups. The Go comma-ok form is a single lookup that returns both the value and the presence flag. Prefer the comma-ok form over comparing the value to its zero value; the zero value might be a legitimate stored value.

Iteration Order

Ranging over a map with `for range` visits every key–value pair, but **in a random order that changes on every iteration** — even two range loops over the same map in the same run can visit keys in different orders.

```
bpm := map[string]int{
    "amapiano": 112,
    "hyperpop": 160,
    "lo-fi":    85,
}

for genre, b := range bpm {
    fmt.Printf("%s: %d\n", genre, b) // order varies every run
}
```

This is a deliberate design choice. Go randomizes map iteration to make it immediately obvious if your code accidentally depends on order. Java's `HashMap` also makes no ordering guarantee, but it tends to be stable in practice within a single run, which can mask order dependencies.

Trap: Never depend on map iteration order. If you need sorted output, collect the keys into a slice and sort it first. Add `"sort"` to your imports at the top of the file:

```
import "sort"
```

Then collect and sort the keys:

```
keys := make([]string, 0, len(bpm))
for k := range bpm {
    keys = append(keys, k)
}
sort.Strings(keys)

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

Clearing a Map

Since Go 1.21, the `clear` built-in deletes all entries from a map:

```
func clear(m map[K]V) // delete all entries; map remains non-nil and usable
```

```
m := map[string]int{"a": 1, "b": 2, "c": 3}
clear(m)
fmt.Println(len(m)) // 0
m["d"] = 4          // fine --- map is still usable
```

After `clear`, the map is empty but not nil. This is different from assigning `m = make(map[K]V)`, which allocates a new map and abandons the old one. `clear` reuses the existing map, which can be useful when the map is shared or when you want to avoid a reallocation.

Sets

Go has no built-in set type — there is no `HashSet` waiting in the standard library the way there is in Java's `java.util`. The idiomatic substitute is a map whose keys are the set's elements and whose values carry no information. Two encodings are common: `map[T]bool` and `map[T]struct{}`.

The empty struct, `struct{}`, is the idiomatic value type because it occupies **zero bytes** — you are using the map purely for its keys, so the value should cost nothing:

```
seen := map[string]struct{}{}  // a set of strings

seen["reggaeton"] = struct{}{} // add an element
seen["dembow"] = struct{}{}

_, ok := seen["reggaeton"]     // membership test (the comma-ok idiom)
fmt.Println(ok)                // true

delete(seen, "dembow")         // remove an element
fmt.Println(len(seen))         // 1 --- the cardinality of the set

for genre := range seen {      // iterate the elements (random order)
    fmt.Println(genre)
}
```

Every set operation is just a map operation you have already seen: add with `set[x] = struct{}{}`, test membership with the comma-ok idiom, remove with `delete`, and count with `len`. The doubled-up `struct{}{}`

reads as "a value of the empty-struct type" — the inner `struct{}` is the type, and the trailing `{}` is a literal of that type.

Tip: `map[T]bool` is the friendlier-looking alternative: you add with `set[x] = true` and test with `if set[x]`, leaning on the `false` zero value for absent keys. It costs one byte per entry instead of zero, which rarely matters. Reach for `map[T]struct{}` when the set is large or memory-sensitive, or when you want the type itself to announce "the values here are meaningless." Use `map[T]bool` when readability wins.

Trap: With the `map[T]bool` encoding, do not confuse "absent" with "present but `false`." Testing `set[x]` returns `false` both when `x` was never added and when you deliberately stored `false`. If `false` is ever a value you might store, use the comma-ok form (`_, ok := set[x]`) or switch to `map[T]struct{}`, where the only way a key can exist is that you put it there.

Arrays

Go has arrays, but you will rarely use them directly. An array in Go is a **value type** with a **fixed size that is part of the type itself**.

```
var a [3]int            // [0 0 0]    --- zero-initialized
b := [3]int{10, 20, 30} // [10 20 30]
c := [...]int{1, 2, 3}  // [1 2 3]    --- compiler counts the elements
```

The `[...]` form lets the compiler infer the length from the literal. After that point the length is still fixed — `c` is type `[3]int`.

Array Length Is Part of the Type

`[3]int` and `[4]int` are **different types**. You cannot pass a `[3]int` to a function that expects a `[4]int`, and you cannot assign one to the other.

```
var x [3]int
var y [4]int
// x = y  // compile error: cannot use [4]int as [3]int
```

Wut: In Java, `int[]` is a reference type regardless of length. In Go, `[3]int` and `[4]int` are as distinct as `int` and `string`. This is why arrays are rarely used as function parameters — you would have to hard-code the length into every function signature. Slices solve this problem.

Arrays Are Value Types

Assigning an array copies every element. Passing an array to a function passes a full copy.

```
a := [3]int{1, 2, 3}
b := a      // b is an independent copy
b[0] = 99
fmt.Println(a[0]) // 1  --- a is unchanged
fmt.Println(b[0]) // 99
```

Java arrays are reference types: assigning `int[] b = a` makes both variables point at the same array. Go arrays have value semantics: every assignment is a copy.

Slices

A slice is a **three-field descriptor** that points into a contiguous region of an underlying array:

Field	Meaning
pointer	address of the first element visible through the slice
length	number of elements currently accessible
capacity	number of elements from the pointer to the end of the backing array

You can think of it as:

```
// conceptual --- not real Go syntax
type sliceHeader struct {
    ptr *T   // pointer to backing array
    len int  // number of accessible elements
    cap int  // elements from ptr to end of backing array
}
```

The type `[]int` (square brackets with no number) is a slice of `int`. `[3]int` (with a number) is an array of three `int` values. The presence or absence of the number is everything.

`len` and `cap` are built-in functions:

```
func len(v Type) int // number of elements in v (slice, array, map, string, channel)
func cap(v Type) int // capacity of v (slice or array; array cap == array len)
```

```
s := []int{10, 20, 30, 40, 50}
fmt.Println(len(s)) // 5
fmt.Println(cap(s)) // 5
```

Slice Literals and `make`

Slice Literal

A slice literal is written just like an array literal, but without a length:

```
s := []int{1, 2, 3}  // len=3, cap=3
```

make

`make` allocates a backing array and returns a slice header pointing to it:

```
func make(t Type, size ...int) Type // allocate and initialize a slice, map, or channel
```

For slices, the two common forms are:

```
s1 := make([]int, 5)     // len=5, cap=5    --- five zeros
s2 := make([]int, 0, 10) // len=0, cap=10   --- empty but room for 10 elements
```

`make([]int, 0, 10)` is the idiomatic way to pre-allocate when you know roughly how many elements you will append. It avoids repeated reallocation as the slice grows. This mirrors Java's `new ArrayList<>(initialCapacity)`: you are not filling the slice, you are reserving room so that `append` calls don't trigger repeated reallocations.

```
s := make([]int, 0, 100) // empty slice, room for 100 elements
for i := range 100 {
    s = append(s, i)      // no reallocation needed
}
```

Tip: `make([]T, 0, n)` is the idiomatic way to pre-allocate a slice you will fill with `append`. `make([]T, n)` is for when you want `n` zero-valued elements you will assign by index.

`nil` Slice vs Empty Slice

```
var s []int          // nil slice   --- pointer is nil, len=0, cap=0
e := []int{}         // empty slice --- pointer is non-nil, len=0, cap=0
m := make([]int, 0)  // empty slice --- same as above
```

Both a nil slice and an empty slice have `len` of zero and behave identically with `range`, `append`, and `len`. The only difference is `s == nil`, which is `true` for a nil slice and `false` for an empty slice. APIs should not distinguish between the two; callers should not need to care whether they received a nil slice or an empty one. [*no-nil-vs-empty-api*]

Tip: A nil slice is a perfectly valid starting point. You can `append` to a nil slice without any initialization. Prefer `var s []int` over `s := []int{}` when you are building a slice with `append`; it is slightly more idiomatic and avoids an unnecessary allocation. [*nil-slice-preferred*]

append

`append` is the built-in function for adding elements to a slice:

```
func append[T any](s []T, elems ...T) []T // append elems to s and return the new slice
```

`append` always returns a new slice header. You must use the returned value — ignoring it discards the length update.

```
s := []int{1, 2, 3}
s = append(s, 4)       // s is now [1 2 3 4]
s = append(s, 5, 6, 7) // append multiple elements at once
```

The Backing Array May Be Replaced

When there is room in the backing array (i.e., `len(s) < cap(s)`), `append` extends the slice in place and no new allocation happens. When the backing array is full, `append` allocates a larger array, copies the existing elements, and returns a slice pointing at the new array.

```
s := make([]int, 0, 3) // len=0, cap=3 --- room for 3 before reallocation
s = append(s, 1)       // len=1, cap=3 --- no reallocation
s = append(s, 2)       // len=2, cap=3 --- no reallocation
s = append(s, 3)       // len=3, cap=3 --- no reallocation
s = append(s, 4)       // len=4, cap>=6 --- new backing array allocated and old contents copied
```

The growth factor is not guaranteed by the specification, but the standard library currently roughly doubles capacity for small slices, then grows more slowly for large ones. The exact strategy can change between Go releases.

Trap: Because `append` may return a slice backed by a **new** array, any other slice that was sharing the old backing array is **not updated**. Always reassign: `s = append(s, elem)`. Never write `append(s, elem)` without capturing the return value — the compiler will reject it anyway with "evaluated but not used".

Appending a Slice

To append all elements from one slice onto another, use the `...` spread operator:

```
a := []int{1, 2, 3}
b := []int{4, 5, 6}
a = append(a, b...) // a is now [1 2 3 4 5 6]
```

The `b...` syntax unpacks `b` into individual arguments. Without it, `append(a, b)` would be a type error because `b` is a `[]int`, not an `int`.

`copy`

`copy` copies elements from one slice to another:

```
func copy(dst, src []Type) int // copy min(len(dst),len(src)) elements; return count copied
```

`copy` copies exactly `min(len(dst), len(src))` elements — it never grows `dst`. It does not append; the destination must already have enough length.

```
src := []int{1, 2, 3, 4, 5}
dst := make([]int, 3)   // len=3
n := copy(dst, src)     // copies 3 elements (limited by dst length)
fmt.Println(dst)        // [1 2 3]
fmt.Println(n)          // 3
```

The primary use of `copy` is to **break aliasing**: when you need an independent slice that will not affect the original if you modify it.

```
original := []int{10, 20, 30}
clone := make([]int, len(original))
copy(clone, original)
clone[0] = 99
fmt.Println(original[0]) // 10 --- original is untouched
fmt.Println(clone[0])    // 99
```

Slicing Expressions

You can derive a new slice from an existing slice (or array) using a **slicing expression**. The result shares the backing array with the original.

Two-Index Form

```
s[low:high] // elements from index low up to (not including) high
```

```
s := []string{"a", "b", "c", "d", "e"}
t := s[1:3] // ["b" "c"]   --- len=2, cap=4 (from index 1 to end of backing array)
```

Either index may be omitted; the defaults are `0` and `len(s)`:

```
s[:3]  // same as s[0:3]
s[2:]  // same as s[2:len(s)]
s[:]   // same as s[0:len(s)]  --- a slice of the whole thing
```

Three-Index Form

The three-index form `s[low:high:max]` sets the **capacity** of the new slice as well as its length.

```
s := []int{10, 20, 30, 40, 50}
t := s[1:3:3] // len=2, cap=2  --- elements [20 30]; cannot reach [40 50]
```

Without the third index, `t` would have `cap=4` and an `append` to `t` could overwrite `s[3]`. With `s[1:3:3]`, the capacity equals the length, so any `append` to `t` forces a new backing array and never touches `s`.

Tip: Use the three-index form when you return a sub-slice from a function and want to guarantee that the caller cannot accidentally modify the original slice by appending to the returned sub-slice.

Slice Aliasing

Re-slicing does **not** copy data. Both the original and the new slice point into the same backing array. Modifying elements through one slice modifies what the other sees.

```
track := []string{"Flaming June", "Sandstorm", "Gouryella", "The Sound of Silence"}
top2 := track[:2]         // shares backing array
top2[0] = "Crazy Train"   // modifies the backing array
fmt.Println(track[0])     // Crazy Train  --- track sees the change too
```

This is efficient (no copying) but surprising when you forget about it.

```
a := []int{1, 2, 3, 4, 5}
b := a[1:3] // b is [2 3], cap=4
b = append(b, 99)
fmt.Println(a) // [1 2 3 99 5]  --- append overwrote a[3]!
```

Here `b` had capacity for two more elements before reaching the end of `a`'s backing array. The `append` wrote `99` into index 3 of the backing array, which is `a[3]`.

Trap: Appending to a sub-slice can silently overwrite elements in the parent slice if the sub-slice has remaining capacity. When in doubt, use `copy` to get a fully independent slice, or use the three-index form to cap the sub-slice's capacity.

When to Use `copy`

Use `copy` when you need a slice that is fully independent of its source:

- You are returning a sub-slice from a function and the caller should not be able to affect the internal state of the original.
- You are modifying a slice inside a goroutine and need to avoid data races with other goroutines that hold the original.
- You want to snapshot the current contents and continue appending to the original without affecting the snapshot.

Passing Slices to Functions

Passing a slice to a function passes the **slice header** by value: a copy of the pointer, length, and capacity. The function receives its own copy of the header, but both copies point at the same backing array.

This means a function **can** modify the elements in the backing array:

```
func doubleAll(s []int) {
    for i := range s {
        s[i] *= 2
```

```
    }
}

nums := []int{1, 2, 3}
doubleAll(nums)
fmt.Println(nums) // [2 4 6]  --- elements were modified through the shared backing array
```

But a function **cannot** change the caller's slice header (length or capacity) without returning the new slice:

```
func tryAppend(s []int) {
    s = append(s, 99) // modifies the function's local copy of the header
}

nums := []int{1, 2, 3}
tryAppend(nums)
fmt.Println(nums)      // [1 2 3]  --- caller's slice header was not changed
fmt.Println(len(nums)) // 3
```

If a function needs to grow a slice, it must return the new slice and the caller must reassign:

```
func addTrack(playlist []string, track string) []string {
    return append(playlist, track)
}

p := []string{"Sandstorm"}
p = addTrack(p, "Gouryella")
fmt.Println(p) // [Sandstorm Gouryella]
```

Tip: The convention in Go is to accept a slice and return the modified slice, just as `append` does. If your function may grow a slice, return it.

Multidimensional Slices

Go has no built-in 2D slice type. A "matrix" is a slice of slices: `[][]int`. Each row is an independent slice and may have a different length.

```
matrix := [][]int{
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9},
}
fmt.Println(matrix[1][2]) // 6
```

To allocate an m×n matrix dynamically:

```
func newMatrix(rows, cols int) [][]int {
    m := make([][]int, rows)
    for i := range m {
        m[i] = make([]int, cols)
    }
    return m
}
```

Because each row is independent, modifying `m[0]` never affects `m[1]`. There is no guarantee that consecutive rows share a single contiguous backing array — each call to `make` allocates independently. If contiguous

memory matters (for performance or interoperability with C), allocate one large `[]int` and slice it into rows manually.

The `slices` Package (Go 1.21+)

Go 1.21 introduced the `slices` package in the standard library, providing the generic utility functions that previously required hand-rolling or third-party libraries. Import it with `import "slices"`.

Sorting

```
func Sort[S ~[]E, E cmp.Ordered](x S)                    // sort x in ascending order in place
func SortFunc[S ~[]E, E any](x S, cmp func(a, b E) int) // sort x using a custom comparison
```

```
tracks := []string{"The Sound of Silence", "Bad Apple!!", "Sandstorm", "Flaming June"}
slices.Sort(tracks)
fmt.Println(tracks) // [Bad Apple!! Flaming June Sandstorm The Sound of Silence]
```

`Sort` works with any `cmp.Ordered` type: integers, floats, and strings. For custom types or reverse order, use `SortFunc`, which takes a comparison function returning a negative number, zero, or a positive number (the same three-way contract as Java's `Comparator.compare`).

The `cmp` package (Go 1.21) supplies a ready-made three-way comparison so you do not have to hand-roll one (and risk overflow from `a - b`). Import it with `import "cmp"`.

```
func Compare[T cmp.Ordered](x, y T) int // -1 if x<y, 0 if x==y, +1 if x>y
```

```
scores := []int{42, 7, 99, 13}
slices.SortFunc(scores, func(a, b int) int {
    return cmp.Compare(b, a) // cmp.Compare is safe; b-a can overflow
})
fmt.Println(scores) // [99 42 13 7]
```

Searching and Testing

```
func Contains[S ~[]E, E comparable](s S, v E) bool // true if v is in s
func Index[S ~[]E, E comparable](s S, v E) int     // index of first occurrence of v, or -1
```

```
genres := []string{"pop", "hip-hop", "indie", "R&B"}
fmt.Println(slices.Contains(genres, "indie"))  // true
fmt.Println(slices.Contains(genres, "metal"))  // false
fmt.Println(slices.Index(genres, "hip-hop"))   // 1
fmt.Println(slices.Index(genres, "metal"))     // -1
```

Other Useful Functions

```
func Compact[S ~[]E, E comparable](s S) S   // remove consecutive duplicate elements
func Collect[E any](seq iter.Seq[E]) []E    // collect an iterator into a slice (Go 1.23)
```

`Compact` is the equivalent of removing consecutive duplicates (like the Unix `uniq` command). `Collect` pairs with the `iter` package covered in Chapter 14.

Tip: Before Go 1.21, sorting a slice required implementing `sort.Interface` (three methods) or using `sort.Slice` with a comparison closure. `slices.Sort` and `slices.SortFunc` are cleaner and type-safe. Prefer the `slices` package for any new code targeting Go 1.21 or later.

`clear` on a Slice (Go 1.21)

The built-in `clear` function was added in Go 1.21. Applied to a slice, it **zeroes all elements** without changing the length or capacity.

```
func clear[T ~[]E, E any](s T)                  // zero all elements; len and cap unchanged
func clear[T ~map[K]V, K comparable, V any](m T) // delete all entries from the map
```

```
s := []int{1, 2, 3, 4, 5}
clear(s)
fmt.Println(s)      // [0 0 0 0 0]
fmt.Println(len(s)) // 5  --- length unchanged
```

`clear` is useful for reusing a slice buffer without releasing the backing array back to the garbage collector. It is **not** the same as `s = s[:0]`, which also keeps the capacity but does not zero the elements (and for slices of pointers or interfaces, would leave stale references that prevent garbage collection).

Wut: `s = s[:0]` resets the length to zero but leaves the old values in the backing array. For slices of integers or floats that does not matter much, but for slices of pointers, interfaces, or structs containing pointers, the stale values keep referenced objects alive longer than you might expect. `clear(s)` followed by `s = s[:0]` properly zeros the elements before shrinking the length.

Try It

Type this in and run it a few times. It builds a play-count map, looks a key up with the comma-ok idiom, sorts the keys for deterministic output, and uses `copy` to take a snapshot that `append` cannot disturb. Watch how the map iteration would jump around if you printed it raw, while the sorted slice stays put.

```
package main

import (
    "fmt"
    "slices"
)

func main() {
    // Build a play-count map, then report it in sorted order.
    plays := map[string]int{
        "Houdini":            910_000_000,   // Dua Lipa
        "Espresso":           1_300_000_000, // Sabrina Carpenter
        "Birds of a Feather": 1_500_000_000, // Billie Eilish
    }

    plays["Houdini"] += 1_000 // maps mutate in place, no reassignment needed

    // comma-ok: distinguish "absent" from "stored zero".
    if n, ok := plays["Texas Hold 'Em"]; ok {
        fmt.Println("found:", n)
    } else {
        fmt.Println("Texas Hold 'Em: not tracked yet")
    }

    // Collect keys into a slice and sort for deterministic output.
    titles := make([]string, 0, len(plays))
    for title := range plays {
```

```
        titles = append(titles, title)
    }
    slices.Sort(titles)

    for _, title := range titles {
        fmt.Printf("%-20s %d\n", title, plays[title])
    }

    // copy breaks aliasing so appends never touch the original.
    snapshot := make([]string, len(titles))
    copy(snapshot, titles)
    snapshot = append(snapshot, "(new arrival)")
    fmt.Println("snapshot:", snapshot)
    fmt.Println("originals intact:", titles)
}
```

The sorted output is deterministic; the snapshot ends with `(new arrival)` while `titles` is unchanged.

Try these modifications:

- Print `plays` directly with `fmt.Println(plays)` and run it several times to watch the iteration order shift.
- Drop the `copy` and append straight into a re-slice of `titles` to see the aliasing trap bite.
- Swap the key-collecting `for range` loop for a `slices.SortFunc` call that orders titles by play count instead of alphabetically.

Key Points

- A map's type is `map[K]V`; `K` must be comparable (supports `==`).
- Declare a map with a literal or `make`; a nil map panics on write but is safe to read.
- Reading a missing key returns the zero value; use `v, ok := m[k]` to distinguish "absent" from "stored zero."
- Map iteration order is randomized by design; sort keys explicitly for deterministic output.
- `clear(m)` (Go 1.21) removes all entries but leaves the map non-nil and reusable.
- Go has no built-in set; use `map[T]struct{}` (zero-byte values) or `map[T]bool` and rely on the keys.
- Arrays are value types; the length is part of the type (`[3]int` and `[4]int` are different types).
- A slice is a three-field header: pointer to backing array, length, and capacity.
- `[]int` is a slice; `[3]int` is an array — the presence of a number is the difference.
- A nil slice and an empty slice both have length zero; a nil slice can be appended to directly.
- `append` always returns a new slice header; always assign the result back.
- When `append` exceeds capacity it allocates a new backing array — slices that shared the old array are not updated.
- `copy` copies `min(len(dst), len(src))` elements and never grows the destination.
- Slicing expressions share the backing array; use `copy` or the three-index form to avoid accidental aliasing.
- Passing a slice to a function passes the header by value; the function can modify elements but cannot change the caller's length or capacity.
- The `slices` package (Go 1.21) provides `Sort`, `SortFunc`, `Contains`, `Index`, and `Compact`; the iterator-bridging `Collect` was added in Go 1.23.
- `clear` on a slice zeroes all elements without changing the length (Go 1.21).

Exercises

1. **Think about it:** In Java, `HashMap<K,V>` requires keys to implement `hashCode()` and `equals()`, and `ArrayList<E>` stores references to boxed objects on the heap. Go's `map[K]V` requires `K` to be comparable

at the language level, and a `[]E` slice stores values directly in the backing array. What are the trade-offs of Go's approach for each collection type? Give one example of a Java key type you cannot use directly as a Go map key, and explain one scenario where storing values directly in a slice (rather than as heap references) matters for performance.

2. **What does this print?**

```go
package main

import "fmt"

func main() {
    catalog := map[string]int{
        "Saltwater":       1_200_000_000,
        "Out Of The Blue": 980_000_000,
    }
    hits := []string{"Out Of The Blue", "Watermelon Sugar", "Saltwater"}
    for _, title := range hits {
        if plays, ok := catalog[title]; ok {
            fmt.Printf("%s: %d\n", title, plays)
        } else {
            fmt.Printf("%s: not found\n", title)
        }
    }
}
```

3. **Calculation:** Given the following code, trace the value of `len(s)` and `cap(s)` after each line. Does any line cause a new backing array to be allocated?

```go
s := make([]int, 2, 5)
s = append(s, 10)
s = append(s, 20)
s = append(s, 30)
s = append(s, 40)
```

4. **Where is the bug?** The following program tries to build a word-frequency map and then print only the words that appear more than once. It compiles but panics at runtime, even though `"Gamemaster"` appears twice. What is wrong, and how do you fix it?

```go
package main

import "fmt"

func main() {
    words := []string{"Gouryella", "Gamemaster", "Flaming June", "Gamemaster", "Sandstorm"}
    var freq map[string]int
    for _, w := range words {
        freq[w]++
    }
    for word, count := range freq {
        if count > 1 {
            fmt.Println(word, count)
        }
    }
}
```

5. **Write a program:** Write a program that reads a slice of song titles and builds a map from the first letter

(as a `string`) to a slice of titles starting with that letter. Use the input `[]string{"Sandstorm", "Bad Apple!!", "Gouryella", "Better Off Alone", "Flaming June", "Sandstorm"}`. Print each letter and its titles in sorted order (sort both the letters and the titles within each group). Collect the map's keys into a slice with a `for range` loop, then use `slices.Sort` for both the keys and the titles within each group.

Chapter 8

Interfaces

You have been writing Go code for several chapters now, and you have already brushed against interfaces — `fmt.Stringer`, `io.Reader`, `error`. This chapter is where they become first-class citizens. Go interfaces are the primary tool for abstraction, and they work very differently from Java's. Java uses **nominal** interfaces: a class is a `Comparable` only if its source says `implements Comparable`, so retrofitting a type you do not own means editing (or wrapping) its source. Go uses **structural** typing: a type satisfies an interface the moment it has the right methods, no declaration required, which lets you define a tiny interface in your own package and have third-party types satisfy it for free. That decoupling is what makes Go code so easy to mock in tests and extend without inheritance. Understanding interfaces unlocks idiomatic Go: clean, testable, composable code that does not require a class hierarchy.

Implicit Interface Satisfaction

In Java, a class declares that it implements an interface:

```
class Song implements Stringer { ... }
```

If you forget `implements`, the class is not a `Stringer`, even if it has every required method. The compiler checks the declaration, not the methods.

Go flips this completely. There is no `implements` keyword. Any type that has the required methods **automatically** satisfies the interface. No declaration needed.

Here is the `fmt.Stringer` interface from the standard library:

```
type Stringer interface {
    String() string  // returns a human-readable representation of the value
}
```

Any type with a `String() string` method is automatically a `fmt.Stringer`.

```
type Track struct {
    Title  string
    Artist string
    BPM    int
}

func (t Track) String() string {
    return fmt.Sprintf("%s by %s (%d BPM)", t.Title, t.Artist, t.BPM)
}
```

`Track` is now a `fmt.Stringer`. No annotation, no registration. `fmt.Println` calls `String()` automatically when it sees a `Stringer`:

```
t := Track{Title: "Sounds of Slashdot", Artist: "San Mehat", BPM: 144}
fmt.Println(t) // Sounds of Slashdot by San Mehat (144 BPM)
```

Tip: This is called **structural typing** or **duck typing** with compile-time checking. If it has the right methods, it satisfies the interface — the compiler verifies this at the point of use, not at the point of definition. Java's approach is **nominal typing**: the name of the interface in the declaration is what matters, not the shape of the type.

Checking Satisfaction Explicitly

You will sometimes want to assert at compile time that a type satisfies an interface, without actually using the interface in a function call. The idiom is:

```
var _ fmt.Stringer = Track{}  // compile error if Track does not satisfy fmt.Stringer
```

The blank identifier discards the value; the assignment only exists to force a compile-time check. This is useful at the top of a file as documentation and a safety net.

Pointer Receivers and the Method Set

There is a wrinkle that trips up Java programmers, who are used to every method being dispatched on a reference. If a method has a **pointer receiver** (Chapter 6), only `*T` satisfies the interface, not `T`. The value type `T` is missing that method from its method set, so it does not satisfy the interface.

```
type Counter struct{ n int }

func (c *Counter) String() string { return fmt.Sprintf("count=%d", c.n) }

var _ fmt.Stringer = &Counter{}  // OK: *Counter has String()
var _ fmt.Stringer = Counter{}   // compile error: Counter does not implement fmt.Stringer
                                 // (method String has pointer receiver)
```

The fix is to pass the address (`&Counter{}`) or to give the method a value receiver if it does not need to mutate. See Chapter 6 for the full rules on value versus pointer receivers and method sets.

Interface Composition

Go interfaces can embed other interfaces, combining their method sets. The standard library uses this pervasively.

```
// io.Reader requires one method
type Reader interface {
    Read(p []byte) (n int, err error)  // reads up to len(p) bytes into p
}

// io.Writer requires one method
type Writer interface {
    Write(p []byte) (n int, err error)  // writes len(p) bytes from p
}

// io.ReadWriter is both
type ReadWriter interface {
    Reader  // embeds io.Reader
```

```
    Writer  // embeds io.Writer
}
```

A type that implements both `Read` and `Write` automatically satisfies `io.ReadWriter`.

You can define your own composed interfaces the same way:

```
type ReadWriteCloser interface {
    io.Reader   // Read(p []byte) (n int, err error)
    io.Writer   // Write(p []byte) (n int, err error)
    io.Closer   // Close() error
}
```

In Java, you would write `interface ReadWriteCloser extends Reader, Writer, Closeable`. Go's embedding syntax is a bit cleaner — you list the interfaces you want to include, and the compiler assembles the combined method set.

Tip: Prefer small, single-method interfaces over large ones. The `io` package is the gold standard: `Reader`, `Writer`, `Closer`, `Seeker` are each one method. Larger interfaces emerge from composing small ones. Do not define an interface until you have a concrete use case that requires it — premature abstraction adds indirection without benefit. [*no-premature-interface*]

any — The Top Type

Sometimes you want a variable that can hold any type. In Java, you do this with `Object` — `Object v` means any type of object can be assigned to `v`. You can do the same in Go with `var v interface{}`, but that is a rather verbose way to express that `v` can be assigned anything. So, Go 1.18 introduced `any` as an alias for `interface{}`. They are identical; `any` is just friendlier to read.

```
var x any = 42      // x holds an int
x = "sabor a mí"    // now x holds a string
x = []int{1, 2, 3}  // now x holds a slice
```

`any` is Go's counterpart to Java's `Object` — every type satisfies the empty interface because there are no methods to implement.

Wut: `any` is not a magic box that avoids copies. Assigning a value to an `any` variable wraps it in an interface value, which holds the concrete type plus a pointer to the data. Despite folklore to the contrary, modern Go does **not** store small values like `int` or `bool` inline in the interface; that inline optimization was removed back in Go 1.4, so a non-pointer value is boxed. The runtime may avoid the allocation in some cases (for example, small integers it keeps cached), but it never stores the value itself inline in the interface word.

Older Go code uses `interface{}` everywhere. When you read code that predates Go 1.18, `interface{}` and `any` mean exactly the same thing. New code should use `any`.

Tip: Use `any` sparingly. Code that traffics in `any` values gives up compile-time type safety and often requires type assertions (see below) to get the value back out. Generics (Chapter 18) are usually the better choice when you want a function that works with multiple types.

Type Assertions

An **interface value** holds two things: the concrete type and the concrete value. A type assertion extracts the concrete value.

The Panicking Form

```
var i any = "Saltwater"

s := i.(string)      // assert that i holds a string; assign it to s
fmt.Println(s)       // Saltwater
```

If the assertion is wrong, the program panics immediately:

```
n := i.(int)  // panic: interface conversion: interface {} is string, not int
```

Use this form when you are certain of the type — for example, immediately after a type switch case.

The Safe Form

```
s, ok := i.(string)  // ok is true if i holds a string
if ok {
    fmt.Println("got string:", s)
} else {
    fmt.Println("not a string")
}
```

The safe form never panics. If the type does not match, `ok` is `false` and `s` is the zero value of the asserted type.

Trap: Always use the two-value form (`v, ok := i.(T)`) when you are not certain of the type. The single-value form panics on a wrong guess, which is a runtime crash, not a compile error.

Type Switches

A **type switch** is a `switch` statement that dispatches on the dynamic type of an interface value. Chapter 4 showed a preview; here is the full picture.

```
func describe(i any) string {
    switch v := i.(type) {
    case int:
        return fmt.Sprintf("int: %d", v)      // v is int here
    case string:
        return fmt.Sprintf("string: %q", v)   // v is string here
    case bool:
        return fmt.Sprintf("bool: %t", v)     // v is bool here
    case []int:
        return fmt.Sprintf("[]int of length %d", len(v))
    case nil:
        return "nil"
    default:
        return fmt.Sprintf("unknown type: %T", v)
    }
}
```

The expression `i.(type)` is only valid inside a type switch — you cannot write it anywhere else. In each case, `v` is automatically converted to the matched concrete type. In the `default` case, `v` retains the type of `i` (i.e., `any`).

You can match multiple types in one case:

```go
switch v := i.(type) {
case int, int64:
    fmt.Println("some integer:", v)  // v is any here because the types differ
case string:
    fmt.Println("string:", v)
}
```

When a case lists more than one type, `v` takes the type of the switch expression (here `any`) because the compiler cannot assign a single concrete type to `v`.

Tip: Type switches are the idiomatic Go replacement for Java's `instanceof` chains. In Java you write:
`if (obj instanceof String s) { ... }`
`else if (obj instanceof Integer n) { ... }`
In Go you write a type switch. It is cleaner and exhaustive — the `default` case catches everything else.

Key Standard Library Interfaces

Go's standard library defines a small set of interfaces that appear everywhere. Knowing them lets you understand most Go code at a glance.

`io.Reader`

```go
type Reader interface {
    Read(p []byte) (n int, err error)  // reads into p; returns bytes read and error
}
```

`Read` fills the slice `p` with up to `len(p)` bytes. It returns the number of bytes actually read and any error. When the underlying data is exhausted, it returns `0, io.EOF`. A reader may also return the final `n > 0` bytes together with `io.EOF` in the same call, so always process the `n` bytes you got before acting on the error.

Here is a concrete type that implements `io.Reader`:

```go
// CountReader wraps an io.Reader and counts bytes as they are read.
type CountReader struct {
    r     io.Reader
    count int
}

func (cr *CountReader) Read(p []byte) (int, error) {
    n, err := cr.r.Read(p)  // delegate to the underlying reader
    cr.count += n           // tally the bytes
    return n, err
}
```

Any function that accepts an `io.Reader` will accept `*CountReader` without modification.

`io.Writer`

```go
type Writer interface {
    Write(p []byte) (n int, err error)  // writes all of p; returns count and error
}
```

`Write` must write exactly `len(p)` bytes or return an error. `os.Stdout`, `*os.File`, `*bytes.Buffer`, and `*strings.Builder` all satisfy `io.Writer`.

A minimal implementation:

```
// UpperWriter wraps an io.Writer and converts all bytes to upper case.
type UpperWriter struct {
    w io.Writer
}

func (uw *UpperWriter) Write(p []byte) (int, error) {
    upper := bytes.ToUpper(p)  // convert to upper case
    return uw.w.Write(upper)   // delegate to the underlying writer
}
```

fmt.Stringer

```
type Stringer interface {
    String() string  // returns a human-readable string representation
}
```

`fmt.Println`, `fmt.Sprintf` with `%v` or `%s`, and most other `fmt` functions check whether a value satisfies `Stringer` and call `String()` if so. You saw this at the start of the chapter with `Track`.

error

```
type error interface {
    Error() string  // returns the error message
}
```

`error` is a predeclared interface, not a type in any package — it is part of the language itself. Any type with an `Error() string` method is an `error`. Error handling is covered in full in Chapter 9; for now, just note the shape.

```
type ValidationError struct {
    Field   string
    Message string
}

func (e *ValidationError) Error() string {
    return fmt.Sprintf("validation failed on %s: %s", e.Field, e.Message)
}
```

`*ValidationError` is now an `error` and can be returned anywhere an `error` is expected.

sort.Interface

```
type Interface interface {
    Len() int              // returns the number of elements in the collection
    Less(i, j int) bool    // reports whether element i should sort before element j
    Swap(i, j int)         // swaps the elements at positions i and j
}
```

Any type that implements all three methods can be sorted by `sort.Sort`. The standard library does not know or care what you are sorting — it only calls these three methods.

Here is a concrete type that sorts a slice of songs by title:

```
type Song struct {
    Title  string
    Artist string
```

```
}

// ByTitle wraps a []Song and sorts it alphabetically by title.
type ByTitle []Song

func (b ByTitle) Len() int           { return len(b) }
func (b ByTitle) Less(i, j int) bool { return b[i].Title < b[j].Title }
func (b ByTitle) Swap(i, j int)      { b[i], b[j] = b[j], b[i] }
```

Use it with `sort.Sort`:

```
songs := []Song{
    {Title: "The Sound of Silence", Artist: "Disturbed"},
    {Title: "Sandstorm",            Artist: "Darude"},
    {Title: "Better Off Alone",     Artist: "DJ Cobra"},
}

sort.Sort(ByTitle(songs))

for _, s := range songs {
    fmt.Printf("%s --- %s\n", s.Title, s.Artist)
}
// Better Off Alone --- DJ Cobra
// Sandstorm --- Darude
// The Sound of Silence --- Disturbed
```

`ByTitle(songs)` is a type conversion that reinterprets the `[]Song` slice as a `ByTitle` — no copying occurs.

Tip: `sort.Interface` is best suited to types that need to be sortable as a **first-class capability** — for example, a domain type that your package exposes and that users will sort repeatedly or pass to generic sort utilities. For a one-off sort inside a single function, Go 1.21's `slices.SortFunc` is simpler:

```
slices.SortFunc(songs, func(a, b Song) int {
    return cmp.Compare(a.Title, b.Title)  // cmp package, Chapter 7
})
```

You saw `slices.SortFunc` and `cmp.Compare` in Chapter 7 (remember to `import "cmp"`); use `sort.Interface` when the sortable behavior belongs to the type itself.

Accept Interfaces, Return Structs

One of Go's most important design idioms is: **accept interfaces, return concrete types**.

When a function's parameter is an interface, the caller can pass any type that satisfies it — including a mock in a test. When a function returns a concrete type, callers get access to all the methods of that type, not just the interface subset. Define the interface in the package that consumes it, not in the package that provides the implementation. [*interface-in-consumer*]

```
// Good: accepts io.Writer so any writer will do --- os.Stdout, a file, a buffer
func WriteJSON(w io.Writer, v any) error {
    enc := json.NewEncoder(w)
    return enc.Encode(v)
}

// Less flexible: only accepts *os.File
func WriteJSONToFile(f *os.File, v any) error {
```

```
    enc := json.NewEncoder(f)
    return enc.Encode(v)
}
```

Returning interfaces makes things harder:

```
// Avoid: callers cannot use *bytes.Buffer methods --- only io.Writer methods
func NewBuffer() io.Writer {
    return &bytes.Buffer{}
}

// Better: callers get the full *bytes.Buffer API
func NewBuffer() *bytes.Buffer {
    return &bytes.Buffer{}
}
```

Tip: Accept the narrowest interface that meets your needs. Return the most specific concrete type you can. This maximizes caller flexibility and testability without hiding useful API. [*return-concrete-types*]

The rule has one well-known exception: the `error` interface. Functions return `error` (an interface) rather than a concrete error type so that callers are not coupled to the specific error implementation. Chapter 9 explains why.

The Interface Nil Trap

This is one of Go's most notorious gotchas. Read this section carefully.

An interface value has two components: a **type** and a **value**. An interface is `nil` only when **both** the type and the value are `nil`. A typed nil — a nil pointer stored in an interface — is **not** nil.

```
type MyError struct{ msg string }

func (e *MyError) Error() string { return e.msg }

func mayFail(fail bool) error {
    var e *MyError  // e is a typed nil: type=*MyError, value=nil
    if fail {
        e = &MyError{msg: "something went wrong"}
    }
    return e        // BUG: always returns a non-nil interface!
}

func main() {
    err := mayFail(false)
    if err != nil {
        fmt.Println("got error:", err)  // this line executes even when fail=false!
    }
}
```

When `fail` is `false`, `e` is `nil` (a nil `*MyError`), but `return e` wraps it in an `error` interface value with type `*MyError` and value `nil`. That interface value is **not** nil because the type field is populated. `err != nil` is `true`.

Trap: Never return a typed nil pointer in an interface. If you want to return "no error," return the untyped `nil` literal directly:

```
func mayFail(fail bool) error {
    if fail {
        return &MyError{msg: "something went wrong"}
    }
    return nil  // untyped nil: type=nil, value=nil --- this is a nil error
}
```

The fix is to return `nil` (untyped) rather than a variable of the concrete error type.

You can inspect the components of an interface value using reflection, but in practice the fix is always the same: return `nil` directly, not a typed nil pointer.

This trap generalizes beyond pointers. Any **typed nil** boxed in an interface is non-nil — a nil map, a nil slice, a nil function value, or a nil channel all carry a populated type field once stored in an interface, so the interface compares unequal to `nil`.

```
var s []int            // s is a nil slice
var i any = s          // but i is NOT nil: type=[]int, value=nil
fmt.Println(i == nil)  // false
```

The lesson is the same regardless of the underlying kind: to mean "no value," assign or return the untyped `nil` literal, never a typed nil that has been wrapped in an interface.

Trap: Comparing two interface values with `==` compares their dynamic types and then their dynamic values, and it **panics at run time** if the dynamic type is not comparable (a slice, map, or function). This bites hardest when you use interface values as map keys, since inserting a key with a non-comparable dynamic type panics on the spot.

```
var a, b any = []int{1}, []int{1}
fmt.Println(a == b)  // panic: runtime error: comparing uncomparable type []int
```

If you need to compare interface values whose dynamic type might be uncomparable, reach for `reflect.DeepEqual` instead.

Try It

Type this in and run it to watch structural typing, interface composition, a type switch, and `sort.Interface` all working together. Notice that `Track` never says it implements `Playable` — it just has the right methods.

```
package main

import (
    "fmt"
    "sort"
)

// Playable is a tiny interface: anything that can describe itself and report a
// duration satisfies it --- no "implements" needed.
type Playable interface {
    fmt.Stringer
    Seconds() int
}

type Track struct {
    Title  string
```

```
    Artist string
    Length int // seconds
}

func (t Track) String() string { return fmt.Sprintf("%s --- %s", t.Title, t.Artist) }
func (t Track) Seconds() int   { return t.Length }

type byLength []Track

func (b byLength) Len() int           { return len(b) }
func (b byLength) Less(i, j int) bool { return b[i].Length < b[j].Length }
func (b byLength) Swap(i, j int)      { b[i], b[j] = b[j], b[i] }

func describe(p Playable) {
    fmt.Printf("%s (%d s)\n", p, p.Seconds())
}

func main() {
    tracks := []Track{
        {"Espresso", "Sabrina Carpenter", 175},
        {"Good Luck, Babe!", "Chappell Roan", 218},
        {"Birds of a Feather", "Billie Eilish", 210},
    }

    sort.Sort(byLength(tracks))
    for _, t := range tracks {
        describe(t) // Track satisfies Playable structurally
    }

    var v any = tracks[0]
    switch p := v.(type) {
    case Playable:
        fmt.Println("playable:", p)
    default:
        fmt.Println("not playable")
    }
}
```

Try these modifications:

- Add a `Podcast` type with `String()` and `Seconds()` and pass it to `describe` — it satisfies `Playable` for free.
- Replace `sort.Sort(byLength(tracks))` with `slices.SortFunc` and `cmp.Compare` (Chapter 7) to sort by title instead.
- Add a `var _ Playable = Track{}` line at the top of the file, then delete the `Seconds()` method and watch the compile-time check fire.

Key Points

- Go uses **structural typing**: a type satisfies an interface by having the right methods, with no `implements` declaration required.
- Interface composition embeds interfaces inside interfaces; `io.ReadWriter` is `io.Reader` + `io.Writer`.
- `any` (alias for `interface{}`) is the top type; use it sparingly and prefer generics when possible.
- Type assertions come in two forms: `v := i.(T)` panics on failure; `v, ok := i.(T)` is safe.

- Type switches dispatch on dynamic type: `switch v := i.(type) { case int: ... }`.
- The five interfaces to know first: `io.Reader`, `io.Writer`, `fmt.Stringer`, `error`, and `sort.Interface`.
- `sort.Interface` (`Len`, `Less`, `Swap`) lets any type be sorted by `sort.Sort`; prefer `slices.SortFunc` for one-off sorts.
- Accept interfaces, return concrete types — this maximizes flexibility and testability.
- A typed nil pointer stored in an interface is **not** nil; always return the untyped `nil` to signal "no error."

Exercises

1. **Think about it:** Go's structural typing means any package can retroactively make its types satisfy an interface defined in any other package. In Java, if you want your `Song` class to satisfy a new interface `Playable` defined in a library you do not control, you must modify `Song`'s source. Explain how Go's approach changes the relationship between library authors and library users. What does this mean for extending types from packages you cannot modify?

2. **What does this print?**

```go
package main

import "fmt"

type Celsius float64
type Fahrenheit float64

func (c Celsius) String() string {
    return fmt.Sprintf("%.1f°C", float64(c))
}

func printTemp(v fmt.Stringer) {
    fmt.Println(v.String())
}

func main() {
    c := Celsius(37.5)
    f := Fahrenheit(99.5)
    printTemp(c)
    fmt.Println(f)
}
```

3. **Calculation:** An interface value in Go stores two fields: a pointer to type information and a pointer to (or copy of) the data. Given a variable declared as `var r io.Reader = &bytes.Buffer{}`, how many distinct type/value components does `r` hold? If `r` is then assigned `nil`, describe the type and value components of the resulting interface value.

4. **Where is the bug?**

```go
package main

import "fmt"

type DBError struct{ code int }

func (e *DBError) Error() string { return fmt.Sprintf("db error %d", e.code) }

func connect(bad bool) error {
    var err *DBError
```

```
    if bad {
        err = &DBError{code: 500}
    }
    return err
}

func main() {
    e := connect(false)
    if e == nil {
        fmt.Println("connected OK")
    } else {
        fmt.Println("failed:", e)
    }
}
```

5. **Write a program:** Define an interface `Shape` with two methods: `Area() float64` and `Perimeter() float64`. Implement `Shape` for two concrete types: `Rectangle` (with fields `Width` and `Height float64`) and `Circle` (with field `Radius float64`; use `math.Pi`). Write a function `printShapeInfo(s Shape)` that prints the area and perimeter. In `main`, create one `Rectangle` and one `Circle` and call `printShapeInfo` on each.

Chapter 9

Error Handling

Go's approach to errors will feel alien at first if you come from Java — there are no checked exceptions, no `try/catch`, and no exception hierarchy. Instead, errors are plain values that functions return alongside their results, and the caller is expected to inspect them immediately. Java's checked exceptions force the compiler to nag you until every failure path is caught or re-declared; Go makes the opposite bet, trusting you to check the `error` value the function hands back. That trust has teeth: nothing stops you from ignoring a returned error, and when you do, a failed write looks like a successful one, a missing record reads as an empty result, and the bug surfaces three layers away from where it started. Treating errors as explicit values is what keeps Go's control flow readable, but only if you actually look at them. This chapter explains the `error` interface, the conventions around creating and returning errors, how to wrap and unwrap error chains, sentinel errors, custom error types, and the narrow use of `panic/recover`. By the end you will understand why "errors are values" is one of Go's defining proverbs and how to apply it in production code.

The `error` Interface

You saw `error` briefly in Chapter 8 when the standard library interfaces were introduced. Here it gets the full treatment.

`error` is a predeclared interface built into the language itself — not in any package:

```
type error interface {
    Error() string  // returns a human-readable description of the error
}
```

Any type with an `Error() string` method automatically satisfies `error`. That is the entire contract.

Return `error` as the Last Value

The universal Go convention is to return `error` as the **last** return value of a function. On success, return `nil`; on failure, return a non-nil error. Chapter 5 introduced this pattern; here is a refresher:

```
func parseTrackNumber(s string) (int, error) {
    n, err := strconv.Atoi(s)  // convert string to int
    if err != nil {
        return 0, fmt.Errorf("invalid track number %q: %w", s, err)
    }
    return n, nil
}
```

The caller handles it immediately:

```go
n, err := parseTrackNumber("3")
if err != nil {
    log.Fatal(err)
}
fmt.Println("Track:", n)
```

Trap: Java programmers sometimes assign the error to a variable and check it later, or silently ignore it with `_`. In Go, the convention is to check the error **right after the call** and return early. Delaying error checks makes the control flow hard to follow and is not idiomatic. Handle errors first so the success path stays unindented and easy to read. [*error-first-return-early*] and [*no-else-after-error*]

Wut: Go has no checked exceptions. The compiler does not force you to handle an `error` return value. You can write `n, _ := parseTrackNumber(s)` and discard the error entirely — the program will compile. Using `_` is occasionally appropriate (e.g., closing a read-only file), but doing it carelessly is a common source of bugs. `go vet` and linters like `errcheck` can flag unchecked errors. [*no-discard-error*]

Creating Errors

The standard library provides two basic ways to create an error value.

`errors.New`

```go
import "errors"

var ErrEmptyPlaylist = errors.New("playlist is empty")
```

`errors.New` returns an opaque error value whose message is the string you provide. Two calls to `errors.New` with the same string return distinct values — equality is by identity, not by message content.

`fmt.Errorf`

```go
import "fmt"

func loadTrack(id int) (*Track, error) {
    if id <= 0 {
        return nil, fmt.Errorf("loadTrack: invalid id %d", id)
    }
    // ...
    return nil, fmt.Errorf("loadTrack: id %d not found", id)
}
```

`fmt.Errorf` creates an error with a formatted message. It is the workhorse of error creation in Go.

Tip: Prefix error messages with the function or package name: `"loadTrack: ..."`. When errors bubble up through several layers, the prefix chain reads like a stack trace in plain text: `"server: loadTrack: id 42 not found"`. Error strings must be lowercase and must not end with punctuation so they compose cleanly when prefixed. [*lowercase-error-strings*]

Wrapping Errors

Go 1.13 added error wrapping: you can attach a **cause** to an error so that callers can inspect the original error without depending on the message string. The `%w` verb in `fmt.Errorf` is how you wrap:

```
func fetchSong(id int) (*Song, error) {
    row, err := db.LookupSong(id)
    if err != nil {
        return nil, fmt.Errorf("fetchSong %d: %w", id, err)  // wrap the database error
    }
    // ...
}
```

The wrapping error remembers the original `err` and exposes it through `errors.Unwrap`:

```
wrapped := fmt.Errorf("outer: %w", inner)
fmt.Println(errors.Unwrap(wrapped)) // prints inner's message
```

You can wrap multiple levels deep, forming an **error chain**. `errors.Is` and `errors.As` (next section) walk the entire chain, so callers do not need to manually call `errors.Unwrap`.

Tip: Use `%w` whenever you add context to an error and want callers to be able to test for the original cause. Use `%v` (not `%w`) when you want to log the cause for humans but do **not** want callers to programmatically inspect it — for example, when the original error is an implementation detail.

`errors.Is` and `errors.As`

These two functions replace the Java pattern of `catch (SpecificException e)` or `e instanceof SomeException`.

`errors.Is` — Matching Sentinel Values

`errors.Is(err, target)` reports whether any error in `err`'s chain equals `target`.

```
import (
    "errors"
    "io"
)

_, err := reader.Read(buf)
if errors.Is(err, io.EOF) {
    fmt.Println("reached end of stream")
}
```

Trap: Never compare errors with `==` when they might be wrapped. `wrappedErr == io.EOF` is `false` if `wrappedErr` was produced by `fmt.Errorf("read: %w", io.EOF)`. `errors.Is(wrappedErr, io.EOF)` correctly walks the chain and returns `true`.

`errors.As` — Extracting a Concrete Type

`errors.As(err, &target)` walks the chain and, if it finds an error assignable to `*target`, sets `target` to that error and returns `true`. This is Go's replacement for `instanceof` with a cast.

```go
type TrackError struct {
    TrackID int
    Reason  string
}

func (e *TrackError) Error() string {
    return fmt.Sprintf("track %d: %s", e.TrackID, e.Reason)
}

func process(err error) {
    var te *TrackError
    if errors.As(err, &te) {
        fmt.Printf("problem with track %d: %s\n", te.TrackID, te.Reason)
    }
}
```

In Java you would write:

```java
if (err instanceof TrackException te) {
    System.out.println("problem with track " + te.trackId);
}
```

The Go form is slightly more verbose but it works through any number of wrapping layers.

Tip: Use `errors.Is` when you want to test for a **specific value** (a sentinel like `io.EOF`). Use `errors.As` when you want to test for a **specific type** and access its fields.

`errors.Join` — Combining Multiple Errors

Go 1.20 added `errors.Join`, which combines multiple errors into one. The combined error's `Error()` method returns each non-nil message on its own line, and `errors.Is`/`errors.As` check all of them.

```go
import "errors"

func validatePlaylist(name, owner string, trackCount int) error {
    var errs []error

    if name == "" {
        errs = append(errs, errors.New("name is required"))
    }
    if owner == "" {
        errs = append(errs, errors.New("owner is required"))
    }
    if trackCount < 0 {
        errs = append(errs, errors.New("track count cannot be negative"))
    }

    return errors.Join(errs...)  // nil if errs is empty
}
```

`errors.Join(nil, nil)` returns `nil`, so you can always pass the slice with `...` and get a nil result when there are no errors.

Tip: `errors.Join` is the idiomatic tool for **validation** patterns where you want to report all problems at once rather than stopping at the first one. In Java you might collect exceptions in a list and throw a composite exception at the end; `errors.Join` is the Go equivalent.

A practical example with Andrew Spencer's discography validator:

```
package main

import (
    "errors"
    "fmt"
)

type Album struct {
    Title    string
    Artist   string
    Year     int
    TrackCnt int
}

func validateAlbum(a Album) error {
    var errs []error
    if a.Title == "" {
        errs = append(errs, errors.New("title is required"))
    }
    if a.Artist == "" {
        errs = append(errs, errors.New("artist is required"))
    }
    if a.Year < 1900 || a.Year > 2100 {
        errs = append(errs, fmt.Errorf("year %d is out of range", a.Year))
    }
    if a.TrackCnt <= 0 {
        errs = append(errs, errors.New("track count must be positive"))
    }
    return errors.Join(errs...)
}

func main() {
    a := Album{
        Title:    "Zombie",
        Artist:   "Andrew Spencer",
        Year:     2022,
        TrackCnt: 23,
    }
    if err := validateAlbum(a); err != nil {
        fmt.Println("invalid:", err)
    } else {
        fmt.Println("album OK:", a.Title)
    }

    bad := Album{Title: "", Artist: "", Year: 1800, TrackCnt: -1}
    if err := validateAlbum(bad); err != nil {
        fmt.Println("invalid album:")
        fmt.Println(err)
```

```
    }
}
```

Output:

```
album OK: Zombie
invalid album:
title is required
artist is required
year 1800 is out of range
track count must be positive
```

Sentinel Errors

A **sentinel error** is a package-level error variable used as a well-known signal. The caller compares against it with `errors.Is`.

The most famous sentinel in Go is `io.EOF`:

```
// in package io
var EOF = errors.New("EOF")
```

`io.EOF` signals that there is no more data to read. It is not a failure — it is expected behavior at the end of a stream. Functions like `bufio.Scanner.Scan` and `io.Copy` consume it internally as a termination signal — a successful `io.Copy` returns `nil`, not `io.EOF`.

Other common sentinels you will encounter:

```
// database/sql
var ErrNoRows = errors.New("sql: no rows in result set")

// os (aliases of the io/fs sentinels, tested via errors.Is)
var ErrNotExist = fs.ErrNotExist     // "file does not exist"
var ErrPermission = fs.ErrPermission // "permission denied"
```

Tip: By convention, sentinel error variable names start with `Err`. Define your own sentinels as package-level `var` declarations using `errors.New`. Export them (capitalize them) when callers outside your package need to test for them.

Trap: Do not test sentinel errors with `==`. Wrap-aware callers must use `errors.Is`:

```
// Wrong: misses wrapped errors
if err == io.EOF { ... }

// Right: walks the entire chain
if errors.Is(err, io.EOF) { ... }
```

Here is a realistic read loop using `io.EOF` as a sentinel:

```
package main

import (
    "bufio"
    "errors"
    "fmt"
    "io"
    "strings"
)
```

```
func main() {
    // Robert Dream House, Darude, Chicane --- trance classics
    src := strings.NewReader("Children\nSandstorm\nSaltwater\n")
    r := bufio.NewReader(src)

    for {
        line, err := r.ReadString('\n')
        if len(line) > 0 {
            fmt.Print("track: ", line)
        }
        if errors.Is(err, io.EOF) {
            break
        }
        if err != nil {
            fmt.Println("unexpected error:", err)
            break
        }
    }
}
```

Output:

```
track: Children
track: Sandstorm
track: Saltwater
```

Custom Error Types

When an error needs to carry structured data — a status code, a field name, a record ID — define a custom error type. Implement the `error` interface by adding an `Error() string` method.

```
// StreamError is returned when a streaming request fails.
type StreamError struct {
    TrackID  string  // ID of the track that failed
    HTTPCode int     // HTTP status code from the streaming service
    Msg      string  // human-readable reason
}

func (e *StreamError) Error() string {
    return fmt.Sprintf("stream error %d for track %q: %s", e.HTTPCode, e.TrackID, e.Msg)
}
```

Return it as the `error` interface:

```
func streamTrack(id string) error {
    if id == "" {
        return &StreamError{TrackID: id, HTTPCode: 400, Msg: "empty track ID"}
    }
    // ... call streaming API ...
    return nil
}
```

The caller uses `errors.As` to recover the structured data:

```
err := streamTrack("")
var se *StreamError
```

```
if errors.As(err, &se) {
    fmt.Printf("HTTP %d: %s\n", se.HTTPCode, se.Msg)
}
// HTTP 400: empty track ID
```

Tip: `errors.As` matches the **concrete type** stored in the error chain — it has nothing to do with receiver style by itself. The receiver you choose decides *which* concrete type satisfies `error`: a pointer receiver makes `*StreamError` the error type, while a value receiver makes `StreamError` the error type. Whatever you return is what ends up in the chain, and the `target` you pass to `errors.As` must point to that same type (`var se *StreamError` paired with returning `&StreamError{...}`). The standard practice for custom error types is a pointer receiver plus returning `*StreamError` values; mixing a value receiver with `var se *StreamError` is the classic mismatch that makes `errors.As` quietly fail to match.

Adding Context with `Unwrap`

If your custom error wraps another error, implement `Unwrap() error` so that `errors.Is` and `errors.As` can traverse your type:

```
type FetchError struct {
    URL string  // URL that was fetched
    Err error   // the underlying error
}

func (e *FetchError) Error() string {
    return fmt.Sprintf("fetch %s: %s", e.URL, e.Err)
}

func (e *FetchError) Unwrap() error {
    return e.Err  // expose the wrapped error for errors.Is / errors.As
}
```

Now `errors.Is(fetchErr, io.EOF)` will correctly check whether the underlying error is `io.EOF`, even though the outer type is `*FetchError`.

A fuller example using DJ Analyzer's "Insomnia":

```
package main

import (
    "errors"
    "fmt"
    "io"
)

// PlaybackError wraps an underlying error with playback context.
type PlaybackError struct {
    Track string  // track that failed to play
    Err   error   // underlying cause
}

func (e *PlaybackError) Error() string {
    return fmt.Sprintf("playback failed for %q: %s", e.Track, e.Err)
}
```

```go
func (e *PlaybackError) Unwrap() error {
    return e.Err  // allows errors.Is / errors.As to walk the chain
}

func play(track string) error {
    // simulate an EOF from a truncated audio stream
    return &PlaybackError{Track: track, Err: io.EOF}
}

func main() {
    err := play("Insomnia")
    fmt.Println(err)

    if errors.Is(err, io.EOF) {
        fmt.Println("stream ended unexpectedly --- retry or skip")
    }

    var pe *PlaybackError
    if errors.As(err, &pe) {
        fmt.Println("affected track:", pe.Track)
    }
}
```

Output:

```
playback failed for "Insomnia": EOF
stream ended unexpectedly --- retry or skip
affected track: Insomnia
```

`panic` and `recover`

Go has `panic` and `recover`, but they are **not** the normal error-handling mechanism. Java programmers sometimes reach for `panic` as a `RuntimeException` substitute — that is the wrong model.

`panic`

`panic` stops the normal execution of the current goroutine, unwinds the call stack running any deferred functions, and terminates the program with a message (unless a deferred function recovers, as described below).

```go
func mustPositive(n int) int {
    if n <= 0 {
        panic(fmt.Sprintf("mustPositive: got %d, want > 0", n))
    }
    return n
}
```

Use `panic` only for **truly unrecoverable situations**:

- A programming error that should have been caught at compile time (e.g., misused internal API).
- Initialization failures so severe the program cannot run at all (e.g., a required config file is missing at startup).
- Invariant violations that indicate a bug, not a runtime condition.

Never use `panic` for expected failure conditions like "file not found" or "invalid user input." Return an `error` instead.

Trap: A common Java habit is to use unchecked exceptions for "should never happen" cases and let them propagate freely. In Go, `panic` is the analog, but the bar is much higher. If callers can reasonably be expected to handle the failure, return an `error`. `panic` is a last resort, not a shortcut. [*errors-not-panic*]

`recover`

`recover` catches a panic in a deferred function and returns the value that was passed to `panic`. Outside of a deferred function, `recover` returns `nil` and has no effect.

```
func safePlay(track string) (err error) {
    defer func() {
        if r := recover(); r != nil {
            err = fmt.Errorf("safePlay: recovered panic: %v", r)
        }
    }()

    // pretend this panics on a bad track
    if track == "" {
        panic("empty track name")
    }
    fmt.Println("playing:", track)
    return nil
}

err := safePlay("")
fmt.Println(err)
// safePlay: recovered panic: empty track name

err = safePlay("Sandstorm")
// playing: Sandstorm
```

Tip: The primary legitimate use of `recover` is inside a **library boundary** — converting an unexpected internal panic into an error that the library returns to its caller. This prevents library bugs from crashing the entire program. The standard `encoding/json` package uses this pattern internally. Do not use `recover` as a general exception handler the way you might use a `catch (Exception e)` block in Java.

Wut: `panic` and `recover` are not the same as Java exceptions. Java exceptions propagate up the call stack and can be caught at any frame with a `try/catch`. Go's `recover` only works in **deferred functions** in the **same goroutine** that panicked. You cannot `recover` a panic from a different goroutine.

The `must` Idiom

Some initialization errors are programmer mistakes, not runtime conditions — a bad regex pattern or a malformed template literal is a bug, not something you should silently swallow or handle per-request. For these cases the standard library provides `Must` wrappers that panic on error:

```
// regexp.MustCompile compiles a pattern and panics if the syntax is invalid.
var trackTitleRE = regexp.MustCompile(`^[A-Za-z0-9 ,'!?&()\-]+$`)

func validTitle(s string) bool {
```

```
    return trackTitleRE.MatchString(s)
}
```

The standard library ships only a handful of these wrappers, one bolted onto each package that happened to need it: `regexp.MustCompile`, `template.Must`, `netip.MustParseAddr`, and a few others. They all do the same thing — take a `(value, error)` pair, panic if the error is non-nil, and otherwise return the value — so writing a new one for every type that returns `(T, error)` is tedious. With generics (Chapter 18) you can write that pattern **once** and reuse it for any such function:

```
// Must unwraps a (value, error) pair, panicking if err is non-nil.
func Must[T any](v T, err error) T {
    if err != nil {
        panic(err)
    }
    return v
}
```

Because `Must` returns the unwrapped value directly, it drops into any package-level `var` that would otherwise need a dedicated `Must` wrapper — no matter the type:

```
var trackTmpl = Must(template.New("track").Parse(
    `{{.Title}} by {{.Artist}} ({{.BPM}} BPM)`,
))

var nyc = Must(time.LoadLocation("America/New_York")) // *time.Location or panic
```

These patterns are appropriate because the panic fires at program startup (package-level `var` blocks and `init()` run before `main`), so a bug in the literal is caught immediately rather than surfacing as a mysterious runtime failure on the first request.

Tip: This little generic `Must` is one of the most-copied helpers in Go; many teams keep a one-line `must` package for it. It exists because Go 1.18 added generics *after* the standard library had already hand-written `MustCompile`, `template.Must`, and the rest — a single generic function now subsumes them all.

Trap: Never call `regexp.MustCompile` or `Must` inside a function that runs per-request or in a loop. They are for compile-time-constant literals only. If the pattern or template text comes from user input or configuration, use `regexp.Compile` or `template.New(...).Parse(...)` and handle the error normally.

Go Error-Handling Proverbs

Two of Go's most quoted proverbs apply directly to this chapter.

"Errors are values"

Rob Pike's essay of the same name (Pike 2015) makes a key point: because errors are just values, you can write functions that operate on them, store them in structs, and compose them with the same tools you use for any other value. This is why `errors.Join`, custom error types, and wrapping all feel natural in Go. Java's exceptions, by contrast, are objects thrown out of the normal return path — they are harder to treat as first-class data.

"Don't just check errors, handle them gracefully"

Checking `if err != nil` and immediately calling `log.Fatal` is not handling an error — it is panicking politely. Handling an error means deciding what to do: retry, return a friendlier message, fall back to a default, or accumulate it with `errors.Join` and report all problems at once. The goal is code that is predictable and informative when things go wrong.

Try It

Type this in and run it. It pulls together the chapter's main moves — a sentinel error, a custom error type with `Unwrap`, and the `errors.Is`/`errors.As` pair — in one small lookup. Watch how the same returned `error` answers two different questions: "is this *kind* of failure?" (`errors.Is`) and "what *value* failed?" (`errors.As`).

```
package main

import (
    "errors"
    "fmt"
)

var ErrTrackNotFound = errors.New("track not found")

// LookupError carries the requested title alongside the underlying cause.
type LookupError struct {
    Title string  // title the caller asked for
    Err   error   // underlying cause (often ErrTrackNotFound)
}

func (e *LookupError) Error() string {
    return fmt.Sprintf("lookup %q: %s", e.Title, e.Err)
}

func (e *LookupError) Unwrap() error {
    return e.Err  // lets errors.Is / errors.As walk the chain
}

func findBPM(title string) (int, error) {
    catalog := map[string]int{
        "Quédate":   108,  // Bizarrap & Quevedo, BZRP Music Sessions #52
        "Sandstorm": 136,
    }
    bpm, ok := catalog[title]
    if !ok {
        return 0, &LookupError{Title: title, Err: ErrTrackNotFound}
    }
    return bpm, nil
}

func main() {
    for _, title := range []string{"Quédate", "Inexistente"} {
        bpm, err := findBPM(title)
        if errors.Is(err, ErrTrackNotFound) {
            var le *LookupError
            if errors.As(err, &le) {
```

```
                fmt.Printf("no encontrado: %q (%v)\n", le.Title, err)
            }
            continue
        }
        if err != nil {
            fmt.Println("unexpected:", err)
            continue
        }
        fmt.Printf("%s is %d BPM\n", title, bpm)
    }
}
```

Then try these modifications:

- Switch the `Error()` and `Unwrap()` methods to value receivers and change `var le *LookupError` to `var le LookupError`; confirm `errors.As` still matches when you return `LookupError{...}` (no `&`), then deliberately mismatch them and watch it fail.
- Wrap the returned error one more level with `fmt.Errorf("findBPM: %w", err)` and confirm both `errors.Is` and `errors.As` still see through the extra layer.
- Replace `%w` with `%v` in that extra wrap and observe how `errors.Is` now returns `false`.

Key Points

- `error` is a predeclared interface: `Error() string`; any type with that method satisfies it.
- Return `error` as the last value; return `nil` for success; check the error immediately after the call.
- `errors.New` creates a simple error; `fmt.Errorf` creates a formatted one.
- `fmt.Errorf` with `%w` wraps an error, forming a chain; use `%v` when you do not want callers to inspect the cause.
- `errors.Is(err, target)` walks the chain to match a sentinel value; never use `==` for this.
- `errors.As(err, &target)` walks the chain to match a concrete type; it is Go's replacement for `instanceof` + cast.
- `errors.Join` (Go 1.20) combines multiple errors; ideal for validation that should report all failures at once.
- Sentinel errors (e.g., `io.EOF`, `sql.ErrNoRows`) are package-level `var` values; name them `ErrFoo`.
- Custom error types implement `error` with a pointer receiver; add `Unwrap() error` if they wrap another error.
- `panic` is for unrecoverable programming errors, not for expected runtime conditions — return an `error` instead.
- `recover` only works in deferred functions in the same goroutine; its main use is converting internal panics into errors at a library boundary.
- "Errors are values": handle them with the same composability you apply to any other data.

Exercises

1. **Think about it:** Java uses checked exceptions to force callers to handle failures. Go returns `error` values that the compiler does not require you to inspect. What are the trade-offs of each approach? In what situations does Go's approach lead to more reliable code, and in what situations might it lead to less reliable code compared to Java's checked exceptions?

2. **What does this print?**

```
package main

import (
```

```
    "errors"
    "fmt"
)

var ErrNotFound = errors.New("not found")

type CatalogError struct {
    Track string
    Err   error
}

func (e *CatalogError) Error() string {
    return fmt.Sprintf("catalog: %s: %s", e.Track, e.Err)
}

func (e *CatalogError) Unwrap() error {
    return e.Err
}

func lookup(track string) error {
    return &CatalogError{Track: track, Err: ErrNotFound}
}

func main() {
    err := lookup("Insomnia")
    fmt.Println(err)
    fmt.Println(errors.Is(err, ErrNotFound))

    var ce *CatalogError
    if errors.As(err, &ce) {
        fmt.Println(ce.Track)
    }
}
```

3. **Calculation:** Consider the following code. For the input `Song{Title: "", Artist: "DJ Analyzer", Year: 2021, BPM: -1}`, how many sub-errors does the joined error returned by `validateSong` contain? What is the output of `fmt.Println(err)` for that input?

```
package main

import (
    "errors"
    "fmt"
)

type Song struct {
    Title  string
    Artist string
    Year   int
    BPM    int
}

func validateSong(s Song) error {
    var errs []error
    if s.Title == "" {
```

```
        errs = append(errs, errors.New("title required"))
    }
    if s.Year < 2000 || s.Year > 2030 {
        errs = append(errs, fmt.Errorf("year %d out of range", s.Year))
    }
    if s.BPM <= 0 {
        errs = append(errs, errors.New("BPM must be positive"))
    }
    return errors.Join(errs...)
}

func main() {
    s := Song{Title: "", Artist: "DJ Analyzer", Year: 2021, BPM: -1}
    err := validateSong(s)
    fmt.Println(err)
}
```

4. **Where is the bug?**

```
package main

import (
    "fmt"
    "io"
    "strings"
)

func readAll(r io.Reader) ([]byte, error) {
    buf := make([]byte, 4)
    var result []byte
    for {
        n, err := r.Read(buf)
        result = append(result, buf[:n]...)
        if err == io.EOF {
            break
        }
        if err != nil {
            return nil, fmt.Errorf("readAll: %w", err)
        }
    }
    return result, nil
}

func main() {
    r := strings.NewReader("Children")
    data, err := readAll(r)
    if err != nil {
        fmt.Println("error:", err)
        return
    }
    fmt.Println(string(data))
}
```

5. **Write a program:** Write a function `parseTimecode(s string) (int, int, error)` that parses a string in the format `"MM:SS"` (e.g., `"03:45"`) and returns the minutes, seconds, and `nil`. Return a descriptive

error using `fmt.Errorf` if the string is not in the expected format, if either part is not a valid integer, or if minutes or seconds are out of range (minutes >= 0, seconds 0–59). Define a sentinel `var ErrInvalidTimecode = errors.New("invalid timecode")` and wrap it with `%w` in your error returns so that callers can use `errors.Is(err, ErrInvalidTimecode)`. In `main`, call `parseTimecode` with at least three inputs: one valid (`"03:45"`), one with a bad format (`"345"`), and one with an out-of-range second (`"01:61"`). Print the result or error for each.

Chapter 10

Goroutines and Channels

Java programmers think about concurrency in terms of threads, locks, and shared mutable state. Go takes a fundamentally different approach: lightweight goroutines communicate through typed channels instead of fighting over shared memory. In Java you reach for an `ExecutorService` or a thread pool to run work concurrently, and as soon as two threads touch the same data you are reasoning about locks, memory visibility, and race conditions — shared-memory threading is notoriously easy to get subtly wrong. Go flips the model: goroutines are so cheap you can launch one per task, and instead of guarding shared state you pass values over channels, letting the type system and the runtime handle the handoff. This makes a whole class of concurrency bugs harder to write in the first place. This chapter covers how to launch goroutines with `go f()`, how channels act as typed conduits with synchronization built in, how `select` coordinates multiple channels, and the Go proverb that ties it all together.

Goroutines

A **goroutine** is a function executing concurrently with the rest of your program. You launch one by placing the `go` keyword in front of a function call:

```
go f()           // launch f in a new goroutine
go pkg.Method()  // works with any callable
go func() {      // anonymous function launched immediately
    fmt.Println("Escape")
}()
```

That is the entire syntax. There is no `Thread` class, no `Runnable`, no `ExecutorService`. There are also no handles to goroutines! That means **you cannot query if a goroutine has completed**. If you need to track the status of goroutines, you have to do it yourself! **sync.WaitGroup**, a common way to do it, is explained in the next chapter.

Goroutines vs Java Threads

Java threads are OS threads under the hood — each one starts with a fixed stack (typically 512 KB to 1 MB) and is managed by the operating system scheduler. Creating tens of thousands of Java threads is impractical because each one reserves a large, fixed block of memory and requires an OS context switch to schedule.

Goroutines are very different:

	Java thread	Go goroutine
Starting stack	512 KB – 1 MB	~2 KB
Scheduler	OS kernel	Go runtime (user space)

	Java thread	Go goroutine
Context switch	kernel mode, microseconds	cooperative + preemptive, nanoseconds
Practical ceiling	thousands	millions

A Go program can run a million goroutines comfortably on a laptop. The runtime multiplexes them onto a small number of OS threads (`GOMAXPROCS`, default equal to CPU count, caps how many threads execute Go code at once — the runtime spins up extra OS threads for goroutines blocked in syscalls or cgo). This is called **M:N scheduling** — M goroutines mapped onto N OS threads.

Tip: `GOMAXPROCS` defaults to the number of CPU cores, but since Go 1.25 the runtime also respects the container's cgroup CPU limit. So inside a container with a CPU quota, `GOMAXPROCS` may be lower than the host's core count — which is usually what you want.

Tip: Because goroutines are cheap, Go code often launches a goroutine for every incoming request or every item in a work queue. Java code typically uses a thread pool to amortize the cost of thread creation; in Go that concern largely disappears.

Goroutine stacks start small (~2 KB) and **grow automatically** when needed. The runtime detects that the stack is about to overflow and copies the goroutine's stack to a larger allocation. You never set a stack size — it is invisible to your code.

Wut: Goroutine stacks can grow and shrink at runtime. This means a pointer into a goroutine's stack frame may become invalid after the stack is relocated. The Go compiler and runtime manage this transparently — you never see it — but it is one reason Go does not allow pointer arithmetic (see Chapter 2).

A Simple Goroutine Example

```go
package main

import (
    "fmt"
    "time"
)

func playTrack(title string) {
    fmt.Println("Playing:", title)
}

func main() {
    go playTrack("Escape")            // runs concurrently
    go playTrack("Legend")            // runs concurrently
    time.Sleep(10 * time.Millisecond) // give goroutines time to run
    fmt.Println("done")
}
```

Trap: When `main` returns, the program exits — even if goroutines are still running. The `time.Sleep` above is a toy fix; real programs use channels or `sync.WaitGroup` (Chapter 11) to wait for goroutines to finish. [*goroutine-must-exit*]

Channels

A **channel** is a typed conduit through which goroutines send and receive values. The zero value of a channel is `nil`; use `make` to create one.

Wut: A send or receive on a `nil` channel blocks forever. This sounds like a bug waiting to happen, but it is handy in a `select`: setting a case's channel variable to `nil` disables that case until you set it back to a real channel.

```
ch := make(chan string)    // unbuffered channel of strings
ch := make(chan int, 10)   // buffered channel of ints, capacity 10
```

The send operator `<-` puts a value into the channel. The receive operator `<-` takes a value out:

```
ch <- "Turn Me On"   // send "Turn Me On" into ch
msg := <-ch          // receive from ch; assign to msg
```

Both the send and receive operators block until the other side is ready, unless the channel is buffered (covered in the next section).

Tip: Channels are first-class values — you can pass them to functions, store them in structs, and return them. The `<-` operator is the same for send and receive; the position relative to the channel name determines the direction: `ch <- v` sends, `v := <-ch` receives.

A Channel-Coordinated Goroutine

```
package main

import "fmt"

func fetchLyrics(out chan<- string) {
    out <- "Do you think you're better off alone?"  // send into channel
}

func main() {
    ch := make(chan string) // unbuffered channel
    go fetchLyrics(ch)      // goroutine sends one value
    lyric := <-ch           // main goroutine receives it
    fmt.Println(lyric)
}
```

Output:

```
Do you think you're better off alone?
```

`main` blocks on `<-ch` until `fetchLyrics` sends. There is no mutex, no lock, no explicit signal — the channel synchronizes the two goroutines automatically.

Buffered vs Unbuffered Channels

An **unbuffered channel** (`make(chan T)`) has no internal queue. A send blocks until a receiver is waiting; a receive blocks until a sender is ready. They synchronize at the point of exchange. This is like a relay baton hand-off: both runners must be at the exchange point at the same time.

A **buffered channel** (`make(chan T, n)`) has an internal queue of capacity `n`. A send blocks only when the queue is full; a receive blocks only when the queue is empty.

```go
ch := make(chan string, 3)  // capacity 3

ch <- "Turn Me On"    // queued immediately, no receiver needed
ch <- "Legend"        // queued
ch <- "Escape"        // queued

fmt.Println(<-ch)  // Turn Me On
fmt.Println(<-ch)  // Legend
fmt.Println(<-ch)  // Escape
```

Tip: Use an unbuffered channel when you want a **synchronization point** — when you need to know that the receiver has received the value. Use a buffered channel when you want to decouple the sender's pace from the receiver's pace, up to a known limit.

Trap: Sending to a full buffered channel blocks just like sending to an unbuffered channel. A channel with capacity 1 is not the same as "fire and forget."

Directional Channel Types

You can restrict a channel to send-only or receive-only in a function signature:

```go
func produce(out chan<- string) { // out is send-only
    out <- "$100 Bills"
}

func consume(in <-chan string) {  // in is receive-only
    fmt.Println(<-in)
}
```

The full bidirectional `chan string` converts to either directional type automatically. Directional types document intent and prevent bugs: a function that only receives cannot accidentally send and vice versa.

```go
ch := make(chan string)  // bidirectional
go produce(ch)           // ch narrows to chan<- inside produce
consume(ch)              // ch narrows to <-chan inside consume
```

Tip: Always use directional channel types in function parameters. If a function only reads from a channel, declare the parameter as `<-chan T`. If it only writes, declare it as `chan<- T`. This is self-documenting and lets the compiler catch mistakes at compile time.

Closing a Channel

Calling `close(ch)` marks the channel as closed. After that:

- **Receiving** from a closed channel returns immediately. If there are buffered values, they are drained first. Once empty, receives return the zero value of the channel's element type.
- **Sending** to a closed channel panics.

The idiomatic way to receive from a channel until it is closed is `range`:

```go
ch := make(chan string, 3)
ch <- "Escape"
ch <- "$100 Bills"
ch <- "Legend"
```

```
close(ch)

for msg := range ch {  // drains until channel is closed and empty
    fmt.Println(msg)
}
```

Output:

```
Escape
$100 Bills
Legend
```

You can also use the comma-ok idiom (introduced in Chapter 7 for maps) to detect a closed channel:

```
msg, ok := <-ch  // ok is false when ch is closed and empty
if !ok {
    fmt.Println("channel closed")
}
```

Trap: Only the sender should close a channel. Closing a channel from the receiver's side is a data race if the sender is still running. Closing an already-closed channel also panics. The rule is simple: the goroutine that produces values owns the channel and is responsible for closing it.

Wut: You never *have* to close a channel. Unlike a file, an unclosed channel is not a resource leak — it will be garbage-collected once all goroutines holding a reference to it exit. Close a channel only when you need the receiver to know that no more values are coming. A goroutine that never exits, however, prevents GC of every value it has closed over. [***leaked-goroutine-grows-memory***]

A Producer-Consumer Pipeline

```
package main

import "fmt"

func generate(tracks []string, out chan<- string) {
    for _, t := range tracks {
        out <- t  // send each track
    }
    close(out)    // signal: no more tracks
}

func main() {
    playlist := []string{"Escape", "$100 Bills", "Legend", "Turn Me On"}
    ch := make(chan string)

    go generate(playlist, ch)

    for track := range ch {
        fmt.Println("Playing:", track)
    }
}
```

Output:

```
Playing: Escape
Playing: $100 Bills
```

```
Playing: Legend
Playing: Turn Me On
```

`generate` closes `ch` when the loop finishes, which causes the `range` in `main` to terminate.

The `select` Statement

`select` is to channels what `switch` is to values: it waits on multiple channel operations and runs the first one that is ready.

```
select {
case msg := <-ch1:
    fmt.Println("from ch1:", msg)
case msg := <-ch2:
    fmt.Println("from ch2:", msg)
}
```

If both `ch1` and `ch2` are ready simultaneously, Go picks one at **random**. If neither is ready, `select` blocks until one becomes ready.

Fan-In

Fan-in merges multiple channels into one. `select` makes this straightforward:

```
package main

import (
    "fmt"
    "time"
)

func source(name, msg string, out chan<- string, delay time.Duration) {
    time.Sleep(delay)
    out <- name + ": " + msg
}

func fanIn(ch1, ch2 <-chan string) <-chan string {
    merged := make(chan string, 2)
    go func() {
        for i := 0; i < 2; i++ {
            select {
            case msg := <-ch1:
                merged <- msg
            case msg := <-ch2:
                merged <- msg
            }
        }
        close(merged)
    }()
    return merged
}

func main() {
    ch1 := make(chan string, 1)
    ch2 := make(chan string, 1)
```

```
    go source("Jaroslav Beck", "Escape",     ch1, 10*time.Millisecond)
    go source("Jaroslav Beck", "Turn Me On", ch2, 5*time.Millisecond)

    for msg := range fanIn(ch1, ch2) {
        fmt.Println(msg)
    }
}
```

Whichever goroutine sends first will be received first, regardless of declaration order.

Timeouts

`time.After(d)` returns a `<-chan time.Time` that receives a value after duration `d`. Use it in a `select` to implement timeouts:

```
select {
case result := <-workCh:
    fmt.Println("got result:", result)
case <-time.After(500 * time.Millisecond):
    fmt.Println("timed out waiting for result")
}
```

Java programmers reach for `Future.get(timeout, unit)`; in Go you compose `select` with `time.After`.

Non-Blocking Send and Receive

Adding a `default` case makes a `select` non-blocking: if no channel is ready, the `default` case runs immediately.

```
select {
case msg := <-ch:
    fmt.Println("received:", msg)
default:
    fmt.Println("nothing ready, moving on")
}
```

Use `default` sparingly. A busy-polling loop around a `select` with `default` wastes CPU time; channels with a proper blocking `select` usually express the intent more clearly.

Tip: A common use of the non-blocking pattern is a **done channel** signal combined with a work channel:

```
select {
case work := <-workCh:
    process(work)
case <-doneCh:
    return  // shut down cleanly
default:
    // nothing to do right now
}
```

Share Memory by Communicating

The most famous Go concurrency proverb is:

> "Don't communicate by sharing memory; share memory by communicating."

In Java, you protect shared state with `synchronized`, `ReentrantLock`, `volatile`, and `AtomicInteger`. The data lives in a shared heap; the locks prevent conflicting access.

Go's channel model inverts this. Instead of several goroutines all reading and writing the same variable while holding a lock, you pass ownership of the data through a channel. At any instant, exactly one goroutine holds the value — the one that last received it from the channel. No lock is needed because the data is never shared simultaneously.

```
// Java style: shared mutable state with a lock
// mu.Lock(); count++; mu.Unlock()

// Go style: one goroutine owns the state; others send requests
type command struct {
    inc    bool
    result chan<- int
}

func counter(cmds <-chan command) {
    count := 0
    for cmd := range cmds {
        if cmd.inc {
            count++
        }
        if cmd.result != nil {
            cmd.result <- count
        }
    }
}
```

Only `counter` touches `count`; all other goroutines communicate with it through the `cmds` channel. There is no lock because there is no shared access.

This model does not replace mutexes entirely — Chapter 11 covers `sync.Mutex` and `sync.WaitGroup` for the cases where channels are the wrong tool. But for many concurrency problems, especially pipelines and fan-out/fan-in patterns, the channel model is cleaner and less error-prone.

Tip: A goroutine that owns a piece of state and exposes it only through a channel is called an **active object** or **actor**. The `counter` example above is one. This pattern eliminates data races on the owned state by construction — you cannot accidentally forget to take a lock. Keeping ownership simple also makes it obvious when the goroutine can exit, which is essential for avoiding leaks. [*obvious-goroutine-lifetimes*]

Try It

Type this in and run it a few times. It launches one goroutine per track, collects results over a channel, and uses `select` with `time.After` to bail out if the work takes too long. Notice that the order of the "fetched" lines changes between runs — that is the scheduler at work.

```
package main

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

```go
// fetch sends a result for each track after a short delay.
func fetch(track string, out chan<- string) {
    time.Sleep(5 * time.Millisecond)
    out <- "fetched: " + track
}

func main() {
    tracks := []string{"Padam Padam", "Houdini", "Espresso"}
    results := make(chan string)

    for _, t := range tracks {
        go fetch(t, results) // one goroutine per track
    }

    timeout := time.After(500 * time.Millisecond)
    got := 0
    for got < len(tracks) {
        select {
        case r := <-results:
            fmt.Println(r)
            got++
        case <-timeout:
            fmt.Println("timed out waiting for fetches")
            return
        }
    }
    fmt.Println("all", got, "tracks fetched")
}
```

The final `all 3 tracks fetched` line is deterministic; the lines above it are not.

Try these modifications:

- Change `results` to a buffered channel (`make(chan string, len(tracks))`) and confirm the program still works.
- Lower the timeout to `1 * time.Millisecond` and watch the `select` take the timeout case instead.
- Add a `done := make(chan struct{})` channel and a third `select` case that returns when something closes it.

Key Points

- A goroutine is launched with `go f()`. Goroutines are cheap (~2 KB initial stack) and multiplexed onto OS threads by the Go runtime.
- Java threads are OS threads with large fixed stacks; Go goroutines are user-space and grow dynamically. You can run millions of goroutines where Java would permit only thousands of threads.
- `make(chan T)` creates an unbuffered channel; `make(chan T, n)` creates a buffered channel with capacity `n`.
- An unbuffered send blocks until a receiver is ready; an unbuffered receive blocks until a sender is ready. A buffered send blocks only when the buffer is full.
- Use directional types (`chan<- T`, `<-chan T`) in function parameters to document intent and catch mistakes at compile time.
- Only the sender should close a channel. Receiving from a closed channel returns the zero value; sending to a closed channel panics.
- You do not have to close a channel; close it only to signal "no more values."
- `range` over a channel drains it until it is closed.

- `select` waits on multiple channel operations; the first ready case runs. A `default` case makes `select` non-blocking.
- Use `time.After` in a `select` case to implement timeouts.
- The Go proverb: "Don't communicate by sharing memory; share memory by communicating." Pass ownership of data through channels instead of locking shared variables.

Exercises

1. **Think about it:** Java's `Thread` and `Runnable` model requires you to think about thread pool sizing. Go's goroutine model mostly frees you from this. Explain the runtime mechanism that makes goroutines cheap enough to use one per task. What cost, if any, do goroutines impose that Java threads do not, and when might you still want to limit the number of running goroutines?

2. **What does this print?**

```go
package main

import "fmt"

func main() {
    ch := make(chan int, 3)

    ch <- 7
    ch <- 13
    ch <- 21
    close(ch)

    for v := range ch {
        fmt.Println(v)
    }

    v, ok := <-ch
    fmt.Println(v, ok)
}
```

3. **Calculation:** Consider the following program. Trace its execution and determine the exact output. How many goroutines are alive (other than `main`) when the final `fmt.Println` in `main` runs?

```go
package main

import "fmt"

func double(in <-chan int, out chan<- int) {
    for v := range in {
        out <- v * 2
    }
    close(out)
}

func main() {
    src := make(chan int, 3)
    dst := make(chan int, 3)

    src <- 3
    src <- 5
```

```
    src <- 8
    close(src)

    go double(src, dst)

    for result := range dst {
        fmt.Println(result)
    }
    fmt.Println("done")
}
```

4. **Where is the bug?**

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    var wg sync.WaitGroup
    results := make(chan string, 3)
    tracks := []string{"Turn Me On", "Legend", "Escape"}

    for _, t := range tracks {
        go func() {
            wg.Add(1)
            defer wg.Done()
            results <- "Playing: " + t
        }()
    }

    wg.Wait()
    close(results)

    for r := range results {
        fmt.Println(r)
    }
}
```

5. **Write a program:** Write a program that launches three goroutines. Each goroutine sleeps for a different duration (use `time.Sleep` with values like `10ms`, `20ms`, `30ms`) and then sends the string `"Jaroslav Beck & Crispin: Legend"`, `"Jaroslav Beck: $100 Bills"`, or `"Jaroslav Beck: Turn Me On"` on its own channel. In `main`, use a `select` loop to receive from all three channels and print each message as it arrives. Also add a `time.After(100 * time.Millisecond)` case that prints `"timeout"` and exits the loop if no message arrives within 100 ms of the last received one. Print the messages in the order they actually arrive.

Chapter 11

Synchronization

Chapter 10 introduced goroutines and channels — Go's preferred way to share work. But channels are not always the right tool: sometimes you need to protect a shared data structure, initialize something exactly once, or coordinate goroutines that don't exchange messages. The `sync` package and `sync/atomic` cover those cases, offering primitives that Java programmers will recognize under new names.

The Go Memory Model

Before reaching for a mutex, you need to understand what the Go memory model guarantees. The model defines **happens-before** relationships: when a write in one goroutine is guaranteed to be visible to a read in another.

Without synchronization, goroutines are allowed to observe memory in any order. The compiler and CPU can reorder instructions as long as the reordering is invisible within a single goroutine — but that reordering **is** visible across goroutines.

The key rules:

- A send on a channel happens-before the corresponding receive from that channel.
- A `sync.Mutex` `Unlock` happens-before any subsequent `Lock` that succeeds.
- `sync.Once.Do` completion happens-before any call to `Do` returns.
- Program initialization (all `init` functions) happens-before `main`.

Wut: Java programmers expect that writing a variable in one thread and reading it in another is safe as long as no two threads write at the same time. In Go (and in Java under the JMM) that is **not** safe unless there is a synchronization action between the write and the read. Without one, the reading goroutine may see a stale or partially written value.

The practical rule: any time two goroutines access the same memory and at least one of them is writing, you must use a channel, a mutex, or an atomic to establish a happens-before relationship. The race detector (`go test -race`) enforces this mechanically — see the end of this chapter.

`sync.Mutex` and `sync.RWMutex`

`sync.Mutex` is Go's equivalent of Java's `synchronized` block. It provides exclusive access to a critical section.

```
import "sync"

type Playlist struct {
```

```
    mu     sync.Mutex // protects tracks
    tracks []string
}

func (p *Playlist) Add(track string) {
    p.mu.Lock()
    defer p.mu.Unlock()
    p.tracks = append(p.tracks, track)
}

func (p *Playlist) Tracks() []string {
    p.mu.Lock()
    defer p.mu.Unlock()
    result := make([]string, len(p.tracks))
    copy(result, p.tracks)
    return result
}
```

`Lock` blocks until the mutex is available. `Unlock` releases it. Using `defer p.mu.Unlock()` immediately after `Lock` ensures the mutex is always released, even if the function panics.

Tip: Always use `defer mu.Unlock()` on the very next line after `mu.Lock()`. This eliminates the risk of forgetting to unlock on every return path.

Trap: A `sync.Mutex` must not be copied after first use. If you embed one in a struct, always pass the struct by pointer (`*Playlist`), never by value. Copying a locked mutex is undefined behavior. [*pointer-receiver-for-mutex*]

Comparison with Java `synchronized`

Java's `synchronized` keyword takes an **object monitor** as its lock:

```
synchronized (this) {
    tracks.add(track);
}
```

Go has no per-object monitor. You declare an explicit `sync.Mutex` field and lock it by name. This is more verbose but also more precise: you can have multiple independent mutexes protecting different fields in the same struct.

`sync.RWMutex`

`sync.RWMutex` is an independent type (not a subtype of `sync.Mutex`) that offers a reader/writer lock with separate read and write modes. Many goroutines can hold the read lock at once, **or** a single goroutine can hold the write lock exclusively — never both. Readers do not block each other; a writer blocks everyone. Use it when reads far outnumber writes.

```
type Catalog struct {
    mu     sync.RWMutex
    songs  map[string]string // title -> artist
}

func (c *Catalog) Lookup(title string) (string, bool) {
    c.mu.RLock()         // multiple goroutines can hold RLock at once
```

```
    defer c.mu.RUnlock()
    artist, ok := c.songs[title]
    return artist, ok
}

func (c *Catalog) Add(title, artist string) {
    c.mu.Lock()          // exclusive write lock
    defer c.mu.Unlock()
    c.songs[title] = artist
}
```

`RLock` and `RUnlock` are the read-side pair. `Lock` and `Unlock` are the write-side pair, identical to `sync.Mutex`.

Tip: Only reach for `RWMutex` when you have measured a contention problem. A plain `Mutex` is faster for workloads with balanced reads and writes because `RWMutex` has higher overhead to track readers.

Tip: Yes, `sync.Map` exists, and yes, it is the answer to the "where did `ConcurrentHashMap` go?" question. No, you usually do not want it. A mutex-guarded map like `Catalog` above is the idiomatic default: type-safe, simple, and easy to reason about. `sync.Map` trades type safety (`any` keys and values) for better performance in one niche — append-mostly caches where many goroutines read, store, and overwrite **disjoint** sets of keys.

```
func (m *Map) Load(key any) (value any, ok bool)                    // lookup
func (m *Map) Store(key, value any)                                 // upsert
func (m *Map) LoadOrStore(key, value any) (actual any, loaded bool) // get/add
func (m *Map) Range(f func(key, value any) bool)                    // iterate
```

`LoadOrStore` returns the existing value (`loaded` is `true`) when the key is already present, and `Range` stops as soon as `f` returns `false`.

sync.WaitGroup

`sync.WaitGroup` lets a goroutine wait for a collection of other goroutines to finish. Java programmers typically use `CountDownLatch` or `CompletableFuture.allOf` for the same pattern; `WaitGroup` is simpler.

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    songs := []string{"Escape", "$100 Bills", "Legend"} // Jaroslav Beck
    var wg sync.WaitGroup

    for _, song := range songs {
        wg.Add(1)
        go func(s string) {
            defer wg.Done()
            fmt.Printf("playing: %s\n", s)
        }(song)
    }
```

```
    wg.Wait() // blocks until all three goroutines call Done
    fmt.Println("all songs finished")
}
```

The three operations are:

```
wg.Add(n)  // increment the counter by n --- call before launching goroutines
wg.Done()  // decrement the counter by 1 --- call when a goroutine finishes
wg.Wait()  // block until the counter reaches zero
```

Trap: Call `wg.Add(n)` **before** launching the goroutines, not inside them. If the goroutine calls `Add` and the main goroutine calls `Wait` before the goroutine starts, `Wait` will return immediately with a zero counter.

Trap: Pass the `WaitGroup` by pointer or use a closure that captures it. Copying a `WaitGroup` after first use is a bug.

`WaitGroup.Go` (Go 1.25+)

The `Add(1)` + `go func() { defer wg.Done() ... }()` dance is so common that Go 1.25 added a helper that bundles all three steps:

```
func (wg *WaitGroup) Go(f func()) // Add(1), run f in a new goroutine, Done when f returns
```

`Go` increments the counter, launches `f` in a new goroutine, and calls `Done` automatically when `f` returns — so you cannot forget the `Done`, and there is no chance of misplacing the `Add` (the Trap above simply cannot happen). The earlier example shrinks to this:

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    songs := []string{"Escape", "$100 Bills", "Legend"} // Jaroslav Beck
    var wg sync.WaitGroup

    for _, song := range songs {
        wg.Go(func() {
            fmt.Printf("playing: %s\n", song)
        })
    }

    wg.Wait() // blocks until all goroutines finish
    fmt.Println("all songs finished")
}
```

Note that `song` is captured directly: since Go 1.22 each loop iteration gets its own copy, so you no longer need the `func(s string){...}(song)` trick to avoid sharing one variable across goroutines.

Tip: Prefer `wg.Go(f)` over the manual `Add/go/defer Done` pattern in new code (Go 1.25+). It is shorter and structurally rules out the misplaced-`Add` and forgotten-`Done` bugs.

Fan-out / Fan-in Without Channels

`WaitGroup` is the idiomatic way to fire off N goroutines and wait for all of them without the ceremony of a results channel. If you need return values, combine `WaitGroup` with a pre-allocated slice (one slot per goroutine, no contention) or use `errgroup` from `golang.org/x/sync` (covered in Chapter 12).

`sync.Once`

`sync.Once` runs a function exactly once, no matter how many goroutines call it concurrently. It is the safe, idiomatic replacement for the double-checked locking pattern that Java programmers used before `volatile` was fixed in Java 5.

```
package main

import (
    "fmt"
    "sync"
)

var (
    once     sync.Once
    catalog  map[string]string
)

func loadCatalog() {
    once.Do(func() {
        // expensive initialization runs exactly once
        catalog = map[string]string{
            "Escape":               "Jaroslav Beck",
            "J'ai pas vingt ans !": "Alizée",
            "J'en ai marre !":      "Alizée",
        }
        fmt.Println("catalog loaded")
    })
}

func main() {
    var wg sync.WaitGroup
    for i := 0; i < 5; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            loadCatalog()
        }()
    }
    wg.Wait()
    fmt.Println(catalog["Escape"]) // Jaroslav Beck
}
```

Output:

```
catalog loaded
Jaroslav Beck
```

The message "catalog loaded" appears exactly once even though five goroutines called `loadCatalog` concurrently. Subsequent calls to `once.Do` return immediately.

Wut: `sync.Once` caches the **first** call's completion, not its result. If the function passed to `Do` panics, the once is still considered done — subsequent callers see the (partial) side effects and `Do` never runs again. Panic inside `Do` is almost always a bug.

The Java equivalent that `sync.Once` replaces:

```
// Java double-checked locking (error-prone before Java 5)
private volatile Map<String,String> catalog;

public Map<String,String> getCatalog() {
    if (catalog == null) {
        synchronized (this) {
            if (catalog == null) {
                catalog = loadExpensive();
            }
        }
    }
    return catalog;
}
```

Go's `sync.Once` does this correctly with zero boilerplate.

Since Go 1.21 you often do not even need the `sync.Once` variable: the `sync` package wraps the whole pattern in `OnceFunc`, `OnceValue`, and `OnceValues`.

```
func OnceFunc(f func()) func()                                  // runs f exactly once
func OnceValue[T any](f func() T) func() T                      // runs once, caches value
func OnceValues[T1, T2 any](f func() (T1, T2)) func() (T1, T2) // two values, read: value+error
```

The lazy-value idiom becomes a one-liner:

```
var getCatalog = sync.OnceValue(loadExpensive) // loadExpensive runs on the first call
```

Every call to `getCatalog()` after the first returns the cached map — the entire Java method above, in one line, with no `volatile` archaeology.

sync.Cond

`sync.Cond` is a condition variable — a way for goroutines to wait for a predicate to become true and for other goroutines to signal when state changes. Java's equivalent is `Object.wait()` / `Object.notify()` / `Object.notifyAll()`, or the more modern `java.util.concurrent.locks.Condition`. In Go, channels are usually the preferred tool for this kind of coordination; reach for `sync.Cond` mainly when you need to broadcast a wakeup to many waiters at once (something a single channel send cannot do).

A `sync.Cond` is always associated with a `sync.Locker` (usually a `*sync.Mutex`):

```
cond := sync.NewCond(&mu) // create a Cond associated with mu
```

The three operations are:

```
cond.Wait()      // atomically unlock mu and suspend; re-lock mu on wake
cond.Signal()    // wake one waiting goroutine
cond.Broadcast() // wake all waiting goroutines
```

Here is a producer/consumer example with a bounded queue:

```
package main

import (
    "fmt"
    "sync"
)

type Queue struct {
    mu    sync.Mutex
    cond  *sync.Cond
    items []string
    cap   int
}

func NewQueue(cap int) *Queue {
    q := &Queue{cap: cap}
    q.cond = sync.NewCond(&q.mu)
    return q
}

func (q *Queue) Push(item string) {
    q.mu.Lock()
    for len(q.items) == q.cap {
        q.cond.Wait() // releases lock, waits for signal, re-acquires lock
    }
    q.items = append(q.items, item)
    q.cond.Broadcast()
    q.mu.Unlock()
}

func (q *Queue) Pop() string {
    q.mu.Lock()
    for len(q.items) == 0 {
        q.cond.Wait()
    }
    item := q.items[0]
    q.items = q.items[1:]
    q.cond.Broadcast()
    q.mu.Unlock()
    return item
}

func main() {
    q := NewQueue(2)

    go func() {
        for _, song := range []string{"$100 Bills", "Legend", "Escape"} {
            q.Push(song)
            fmt.Println("pushed:", song)
```

```
        }
    }()

    for i := 0; i < 3; i++ {
        fmt.Println("popped:", q.Pop())
    }
}
```

Trap: Always check the wait condition in a `for` loop, not an `if`. Unlike Java's `Object.wait()`, Go's `Cond.Wait` never wakes spuriously — but another goroutine may have consumed the item between the wakeup and re-acquiring the lock (and `Broadcast` wakes waiters whose predicate may already be false), so you must re-check the predicate before proceeding. The discipline is the same as Java's `while (condition) { lock.wait(); }`.

Tip: Prefer `Broadcast` over `Signal` when multiple goroutines could each satisfy the predicate. A spurious `Signal` that wakes the wrong waiter wastes a cycle; `Broadcast` wakes them all and lets them re-check.

Wut: `sync.Cond` cannot be used with Go's `select` statement. If you need to select between "condition is met" and a timeout or another channel, restructure using channels instead.

sync/atomic

The `sync/atomic` package provides low-level atomic memory operations. In Go 1.19, typed atomic types were added that are much safer than the old function-based API.

Typed Atomics (Go 1.19+)

```
import "sync/atomic"
```

The typed atomic types:

```
atomic.Bool                     // atomic bool
atomic.Int32, atomic.Int64      // atomic signed integers
atomic.Uint32, atomic.Uint64    // atomic unsigned integers
atomic.Uintptr                  // atomic uintptr
atomic.Pointer[T]               // atomic pointer to T (generic)
```

Each type provides the same set of methods:

```
func (x *Int64) Load() int64                    // read atomically
func (x *Int64) Store(val int64)                // write atomically
func (x *Int64) Add(delta int64) (new int64)    // add and return new value
func (x *Int64) Swap(new int64) (old int64)     // set and return old value
func (x *Int64) CompareAndSwap(old, new int64) bool   // CAS: swap if current == old
```

`atomic.Pointer[T]` uses generics so you get type safety without a cast:

```
func (x *Pointer[T]) Load() *T          // load atomically
func (x *Pointer[T]) Store(val *T)      // store atomically
func (x *Pointer[T]) Swap(new *T) *T    // swap atomically
func (x *Pointer[T]) CompareAndSwap(old, new *T) bool // CAS
```

Here is a play-count tracker using atomic integers:

```go
package main

import (
    "fmt"
    "sync"
    "sync/atomic"
)

func main() {
    var plays atomic.Int64
    var wg sync.WaitGroup

    songs := []string{
        "Escape", "$100 Bills", "Legend",
        "J'ai pas vingt ans !", "J'en ai marre !",
    }

    for _, song := range songs {
        wg.Add(1)
        go func(s string) {
            defer wg.Done()
            plays.Add(1) // no mutex needed
            fmt.Printf("played: %s\n", s)
        }(song)
    }

    wg.Wait()
    fmt.Println("total plays:", plays.Load()) // 5
}
```

When to Use Atomics vs Mutexes

Use atomics for:

- Simple counters (hits, errors, bytes transferred).
- A single flag that goroutines read and one goroutine writes.
- Lock-free data structures where you understand the ABA problem.

The **ABA problem** is a subtle hazard in lock-free code built on `CompareAndSwap`: a value changes from A to B and back to A between your read and your CAS, so the CAS succeeds even though the underlying state was modified and restored in between. The successful swap hides the fact that other goroutines touched the data, which can corrupt structures like lock-free stacks.

Use a mutex when:

- You are protecting more than one variable together (invariant maintenance).
- The critical section does more than a single load/store/add.

Tip: Java's `java.util.concurrent.atomic.AtomicLong` maps directly to `atomic.Int64`. The semantics are the same: both behave as sequentially consistent atomics, and an atomic store observed by an atomic load establishes a happens-before edge for surrounding memory too (Go 1.19 memory model; Java `volatile` semantics). Still prefer a mutex when several variables must change together — atomics order memory but cannot make a multi-variable update atomic.

The Race Detector

Go ships a built-in race detector based on ThreadSanitizer. Enable it with the `-race` flag:

```
go test -race ./...
go run -race main.go
go build -race -o myapp .
```

When a data race is detected at runtime, the race detector prints a detailed report showing both the racing accesses and their goroutine stack traces:

```
==================
WARNING: DATA RACE
Write at 0x00c0000b4010 by goroutine 7:
  main.main.func1()
      /tmp/race.go:12 +0x2c

Previous read at 0x00c0000b4010 by goroutine 6:
  main.main.func1()
      /tmp/race.go:9 +0x30
==================
```

The race detector adds roughly 2–20x runtime overhead and 5–10x memory overhead. It is not suitable for production, but it should run in CI on every pull request and on every test suite.

Tip: Run `go test -race ./...` in CI on every push. Data races are undefined behavior: the program may produce wrong results, crash, or appear to work correctly on your machine while failing in production. The race detector is the only reliable way to find them.

Here is an example the race detector catches immediately:

```go
package main

import (
    "fmt"
    "sync"
)

func main() {
    counter := 0 // shared without synchronization
    var wg sync.WaitGroup
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            counter++ // DATA RACE: concurrent read-modify-write
        }()
    }
    wg.Wait()
    fmt.Println(counter) // result is unpredictable
}
```

The fix: replace `counter` with `atomic.Int64` or protect it with a `sync.Mutex`.

Try It

Type this in and run it twice with `go run .` and once with `go run -race .` to convince yourself it stays correct under the detector. It exercises four primitives at once: a `sync.Mutex` guarding a map, `sync.WaitGroup` to fan out and wait, `sync.Once` for one-time setup, and `atomic.Int64` for a lock-free counter.

```
package main

import (
    "fmt"
    "sort"
    "sync"
    "sync/atomic"
)

type Charts struct {
    mu    sync.Mutex     // protects spins
    spins map[string]int // title -> play count
}

func (c *Charts) Play(title string) {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.spins[title]++
}

func main() {
    charts := &Charts{spins: make(map[string]int)}
    var total atomic.Int64
    var ready sync.Once
    var wg sync.WaitGroup

    queue := []string{"Espresso", "Espresso", "Birds of a Feather", "Houdini"}
    for _, song := range queue {
        wg.Add(1)
        go func(s string) {
            defer wg.Done()
            ready.Do(func() { fmt.Println("dj booth online") }) // runs once
            charts.Play(s)
            total.Add(1)
        }(song)
    }

    wg.Wait()
    fmt.Println("total spins:", total.Load())

    titles := make([]string, 0, len(charts.spins))
    for t := range charts.spins {
        titles = append(titles, t)
    }
    sort.Strings(titles)
    for _, t := range titles {
        fmt.Printf("%s: %d\n", t, charts.spins[t])
    }
```

```
}
```

The "dj booth online" line prints exactly once and the total is always `4`, no matter how the goroutines interleave.

Try these modifications:

- Drop the `c.mu.Lock()` / `defer c.mu.Unlock()` from `Play` and run with `-race` — watch the detector flag the concurrent map writes.
- Swap the `atomic.Int64` for a plain `int` incremented with `total++` and confirm `-race` catches that too.
- Replace `charts.mu` with a `sync.RWMutex` and add a `Spins(title string) int` reader method that takes `RLock`, then call it concurrently with the writers.

Key Points

- The Go memory model defines **happens-before** relationships; without synchronization, goroutines may observe stale memory.
- `sync.Mutex` provides exclusive access; use `defer mu.Unlock()` immediately after `mu.Lock()`.
- `sync.RWMutex` allows concurrent readers; reach for it only when reads dominate and you have measured a contention problem.
- `sync.WaitGroup` is the idiomatic fan-out/fan-in primitive when you do not need to pass results back through a channel.
- `sync.Once` runs a function exactly once; it is the correct, simple replacement for double-checked locking.
- `sync.Cond` is a condition variable; always check the predicate in a `for` loop, not an `if`.
- `atomic.Int64`, `atomic.Pointer[T]`, and friends (Go 1.19+) are the type-safe atomic primitives; use them for simple counters and flags.
- Run `go test -race ./...` in CI; a data race is undefined behavior and may be silent on your laptop.

Exercises

1. **Think about it:** Java's `synchronized` keyword locks an object's monitor, which is built into every Java object. Go has no per-object monitor; instead you declare explicit `sync.Mutex` fields. What are the practical advantages and disadvantages of each approach? Consider: what happens when you need to protect two independent fields in the same struct, and how would you do it with each language's mechanism?

2. **What does this print?**

```go
package main

import (
    "fmt"
    "sync"
)

func main() {
    var once sync.Once
    var wg sync.WaitGroup
    results := make([]string, 3)

    for i := 0; i < 3; i++ {
        wg.Add(1)
        go func(n int) {
            defer wg.Done()
            once.Do(func() {
```

```
                results[n] = "loaded"
            })
            if results[n] == "" {
                results[n] = "skipped"
            }
        }(i)
    }

    wg.Wait()
    loaded := 0
    skipped := 0
    for _, r := range results {
        if r == "loaded" {
            loaded++
        } else if r == "skipped" {
            skipped++
        }
    }
    fmt.Printf("loaded=%d skipped=%d\n", loaded, skipped)
}
```

3. **Calculation:** Consider the following program fragment:

```
var counter atomic.Int64
var wg sync.WaitGroup

for i := 0; i < 4; i++ {
    wg.Add(1)
    go func() {
        defer wg.Done()
        counter.Add(10)
    }()
}
wg.Wait()
fmt.Println(counter.Load())
```

   (a) What value does `counter.Load()` always print, regardless of goroutine scheduling order?
   (b) If `counter.Add(10)` were replaced by `counter.Add(int64(i))` (capturing `i` from the loop), what value would always be printed? Would your answer have differed in Go 1.21 or earlier, and if so, why?

4. **Where is the bug?**

```
package main

import (
    "fmt"
    "sync"
)

type SafeMap struct {
    mu sync.Mutex
    m  map[string]int
}

func NewSafeMap() SafeMap {
```

```
    return SafeMap{m: make(map[string]int)}
}

func (s SafeMap) Inc(key string) {
    s.mu.Lock()
    defer s.mu.Unlock()
    s.m[key]++
}

func (s SafeMap) Get(key string) int {
    s.mu.Lock()
    defer s.mu.Unlock()
    return s.m[key]
}

func main() {
    sm := NewSafeMap()
    var wg sync.WaitGroup
    for i := 0; i < 100; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            sm.Inc("Escape")
        }()
    }
    wg.Wait()
    fmt.Println(sm.Get("Escape"))
}
```

5. **Write a program:** Implement a concurrent-safe `RateLimiter` struct that uses a `sync.Mutex` to protect a counter and a `time.Time` field tracking when the window resets. The struct should have a method `Allow(n int) bool` that returns `true` if `n` tokens are available in the current one-second window, deducting them if so, and `false` otherwise (without deducting). Write a `main` function that launches 10 goroutines, each calling `Allow(1)` in a loop 5 times, and prints how many calls were allowed versus denied across all goroutines combined. Use `sync.WaitGroup` to wait for all goroutines to finish.

6. **Where is the bug?**

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    var wg sync.WaitGroup
    results := make([]int, 5)
    for i := 0; i < 5; i++ {
        go func(n int) {
            wg.Add(1)
            defer wg.Done()
            results[n] = n * n
        }(i)
    }
```

```
    wg.Wait()
    fmt.Println(results)
}
```

The author expects this to print `[0 1 4 9 16]`, but it usually prints something like `[0 0 0 0 0]` or a partial result, and `go run -race` reports a data race on `results`. What is wrong, and how would you fix it? (Hint: where is `wg.Add(1)` called, and what does `go vet` say about it?)

Chapter 12

Context and Concurrency Patterns

Chapters 10 and 11 gave you goroutines, channels, and the `sync` primitives you need to coordinate them. This chapter adds the layer that sits on top of all of that: `context.Context`, the standard way to propagate cancellation and deadlines across goroutine boundaries. It also covers the patterns you will see in real Go services — worker pools, rate limiters, fan-out with error collection, and goroutine leak detection.

context.Context

Java has no direct equivalent to `context.Context`. The closest Java analog is a combination of `Future.cancel()`, `ExecutorService.shutdownNow()`, and a hand-rolled deadline field on a request object — all bolted together differently in every codebase. Go standardizes all of this in a single interface.

```
// context.Context is defined in the standard library as:
type Context interface {
    Deadline() (deadline time.Time, ok bool) // returns the deadline, if any
    Done() <-chan struct{}                   // closed when work should be cancelled
    Err() error                              // nil, then Canceled or DeadlineExceeded
    Value(key any) any                       // returns the value associated with key, or nil
}
```

Every long-running or network-bound function in idiomatic Go accepts a `context.Context` as its first parameter. When the context is cancelled — because a deadline expired, a timeout elapsed, or the caller called a cancel function — `Done()` is closed and `Err()` returns a non-nil error. Your function is expected to notice this and return promptly.

Tip: `context.Context` is not just for HTTP handlers. Use it everywhere work can be cancelled: database queries, RPC calls, file I/O, and long computations.

The Root Contexts

Every context tree starts with one of two roots:

```
ctx := context.Background() // the default root; never cancelled, no deadline, no values
ctx := context.TODO()       // placeholder for "I haven't wired up a context yet"
```

`context.Background()` is for `main`, top-level servers, and test helpers. `context.TODO()` is a compile-time signal that you know a context should be here but have not plumbed it through yet. Treat `context.TODO()` like a `// TODO` comment — it should not survive into production code.

Cancellation, Deadlines, and Timeouts

The three constructors that add cancellation to a context are `context.WithCancel`, `context.WithDeadline`, and `context.WithTimeout`. Each returns a derived context and a cancel function. **Always call the cancel function**, even if the operation finishes before the deadline. Failing to call cancel keeps the derived context (and its timer) alive until the deadline fires or the parent is cancelled, pinning everything the context references.

```
// WithCancel returns a copy of parent whose Done channel is closed when cancel is called.
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)

// WithDeadline returns a copy of parent with a deadline set to d.
func WithDeadline(parent Context, d time.Time) (ctx Context, cancel CancelFunc)

// WithTimeout returns WithDeadline(parent, time.Now().Add(timeout)).
func WithTimeout(parent Context, timeout time.Duration) (ctx Context, cancel CancelFunc)
```

Here is a function that fetches a song's lyrics from a slow API, with a two-second timeout:

```
package main

import (
    "context"
    "fmt"
    "time"
)

// fetchLyrics simulates a slow network call; it respects ctx cancellation.
func fetchLyrics(ctx context.Context, song string) (string, error) {
    done := make(chan string, 1)
    go func() {
        time.Sleep(3 * time.Second) // simulate slow work
        done <- "I can't keep loving you the way I do"
    }()
    select {
    case lyrics := <-done:
        return lyrics, nil
    case <-ctx.Done():
        return "", ctx.Err() // context.DeadlineExceeded
    }
}

func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 2*time.Second)
    defer cancel() // always call cancel

    lyrics, err := fetchLyrics(ctx, "Gouryella")
    if err != nil {
        fmt.Println("timed out:", err) // timed out: context deadline exceeded
        return
    }
    fmt.Println(lyrics)
}
```

When the timeout fires, `ctx.Done()` is closed and `ctx.Err()` returns `context.DeadlineExceeded`. If `cancel()` is called before the timeout, `ctx.Err()` returns `context.Canceled`. Your code can distinguish the two with

errors.Is:

```
if errors.Is(ctx.Err(), context.DeadlineExceeded) {
    fmt.Println("ran out of time")
}
if errors.Is(ctx.Err(), context.Canceled) {
    fmt.Println("caller gave up")
}
```

Trap: Do not store the cancel function and call it lazily. Use `defer cancel()` immediately after the `WithTimeout` / `WithCancel` call. If you forget, the runtime cannot release resources associated with the context until the parent is cancelled or the program exits.

Checking Cancellation Inside a Loop

Goroutines doing CPU-bound work need to poll the context rather than waiting on a channel:

```
func processTracks(ctx context.Context, tracks []string) error {
    for _, t := range tracks {
        select {
        case <-ctx.Done():
            return ctx.Err() // bail out early
        default:
        }
        // do real work with t
        fmt.Println("processing:", t)
    }
    return nil
}
```

The `select` with a `default` branch is non-blocking: it drains `Done` if it is already closed but does not block if it is still open. Chapter 10 covered `select` in detail.

context.WithValue

`context.WithValue` attaches a key-value pair to a context that child goroutines can retrieve with `ctx.Value(key)`. This is intended for **request-scoped metadata** — things like a trace ID or an authenticated user — not for passing optional function parameters.

```
func WithValue(parent Context, key, val any) Context
```

The value is retrieved by calling `Value(key)` on any derived context. If no value is found at the current level, Go walks up the context tree until it finds one or reaches the root.

Use Unexported Key Types

If you use a plain string as a key, any package in the call tree can accidentally shadow your value by using the same string key. The idiomatic fix is to define a **package-private type** for your keys. Because the type is unexported, no other package can construct a value of that type, so collisions are impossible.

```
package requestmeta

// traceKey is unexported; no other package can create a value of this type.
type traceKey struct{}

func WithTraceID(ctx context.Context, id string) context.Context {
```

```
    return context.WithValue(ctx, traceKey{}, id)
}

func TraceID(ctx context.Context) (string, bool) {
    id, ok := ctx.Value(traceKey{}).(string)
    return id, ok
}
```

Contrast this with the anti-pattern:

```
// ANTI-PATTERN: string keys can collide across packages
ctx = context.WithValue(ctx, "traceID", "abc-123")
ctx = context.WithValue(ctx, "traceID", "xyz-999") // silently shadows the first one!
```

Trap: Never use a built-in type (string, int, etc.) as a context key. Linters such as `staticcheck` (check SA1029) flag this with `should not use built-in type string as key for value` — note that plain `go vet` does *not* catch it, so do not rely on the standard tool alone here. Always define an unexported struct type for context keys.

Context as First Parameter

Go has a universal convention: if a function accepts a context, it is **always the first parameter** and it is **always named `ctx`**. [*ctx-for-context*]

```
// idiomatic
func SearchSongs(ctx context.Context, query string) ([]Song, error)

// wrong: context buried in the middle
func SearchSongs(query string, ctx context.Context) ([]Song, error)
```

This convention applies throughout the standard library, all major frameworks, and community packages. Do not put a context inside a struct (except when constructing a long-lived object like an HTTP server); pass it explicitly on each call.

Trap: Do not store a `context.Context` in a struct field and use it later. Contexts are request-scoped. A stored context will be cancelled at unpredictable times. Pass the context explicitly to every function that needs it.

The convention is so consistent that when you see a function signature like `func Foo(ctx context.Context, ...)` you immediately know it can be cancelled, timed out, and carries request metadata. Java has no equivalent signal at the call site.

errgroup — Fan-Out with Error Collection

Chapter 11 showed `sync.WaitGroup` for fan-out. `WaitGroup` works, but it cannot collect errors from goroutines. The `golang.org/x/sync/errgroup` package solves both problems: it waits for a group of goroutines to finish **and** returns the first non-nil error any of them produced. `errgroup.WithContext` returns a derived context that the group cancels as soon as any goroutine returns an error.

```
import "golang.org/x/sync/errgroup"

// Group is created with errgroup.WithContext.
// g.Go(f) launches f in a goroutine; g.Wait() blocks until all goroutines finish.
// g.Wait() returns the first non-nil error returned by any goroutine.
```

```
func WithContext(ctx context.Context) (*Group, context.Context)
func (g *Group) Go(f func() error)
func (g *Group) Wait() error
```

Here is a fan-out that fetches three song titles concurrently and cancels all of them if any fails:

```
package main

import (
    "context"
    "fmt"
    "time"

    "golang.org/x/sync/errgroup"
)

// fetchTitle simulates fetching a song title; slow returns an error.
func fetchTitle(ctx context.Context, id int) (string, error) {
    titles := []string{"Gouryella", "Flaming June", "Saltwater"}
    select {
    case <-time.After(time.Duration(id+1) * 100 * time.Millisecond):
        return titles[id%len(titles)], nil
    case <-ctx.Done():
        return "", ctx.Err()
    }
}

func main() {
    ctx := context.Background()
    g, ctx := errgroup.WithContext(ctx)

    results := make([]string, 3)
    for i := 0; i < 3; i++ {
        i := i // capture loop variable; unnecessary on Go 1.22+ (per-iteration scope)
        g.Go(func() error {
            title, err := fetchTitle(ctx, i)
            if err != nil {
                return err
            }
            results[i] = title
            return nil
        })
    }

    if err := g.Wait(); err != nil {
        fmt.Println("error:", err)
        return
    }
    for _, t := range results {
        fmt.Println(t)
    }
}
```

When any goroutine in the group returns a non-nil error, `errgroup` cancels the shared context. Goroutines that are still running will see `ctx.Done()` closed and should return promptly. `g.Wait()` blocks until all

goroutines have returned, then returns the first error.

Tip: `errgroup` is the idiomatic replacement for `sync.WaitGroup` whenever goroutines can fail. If none of them can fail, `sync.WaitGroup` is fine.

Goroutine Leak Detection

A goroutine leak is a goroutine that starts but never exits. Leaks accumulate over the lifetime of a server: each request might leave one goroutine behind, and after thousands of requests you have thousands of idle goroutines consuming memory and scheduler time. Each leaked goroutine also keeps every value it has closed over alive, preventing GC from reclaiming that memory. [*leaked-goroutine-grows-memory*]

The most common cause is a goroutine blocked on a channel send or receive with no path to exit:

```
// BUG: this goroutine leaks if the caller stops listening on results.
func streamHits(results chan<- string) {
    for {
        results <- "Gamemaster" // blocks forever if nobody reads
    }
}
```

Every goroutine must have a clear exit path. [*goroutine-must-exit*] The idiomatic exits are:

- the function returns naturally,
- a `done` channel is closed,
- the context passed in is cancelled.

The `go.uber.org/goleak` package detects leaked goroutines in tests:

```
import (
    "testing"
    "go.uber.org/goleak"
)

func TestMain(m *testing.M) {
    goleak.VerifyTestMain(m) // fails the test suite if any goroutine leaks
}

func TestNoLeak(t *testing.T) {
    defer goleak.VerifyNone(t) // fails this test if a goroutine leaks

    ctx, cancel := context.WithCancel(context.Background())

    done := make(chan struct{})
    go func() {
        defer close(done)
        select {
        case <-ctx.Done(): // goroutine exits when context is cancelled
        }
    }()

    cancel() // signal the goroutine to exit
    <-done   // wait for it to finish before VerifyNone runs
}
```

`goleak.VerifyTestMain(m)` runs `m.Run()` itself, checks for leaked goroutines after the suite completes, and then calls `os.Exit` with the test exit code. Because it calls `os.Exit` for you, it must be the only thing your `TestMain` does — do not call `m.Run()` or `os.Exit` yourself, or you will exit twice and skip the leak check. `goleak.VerifyNone(t)` checks for leaks at the end of a single test function.

Tip: Run `goleak.VerifyTestMain` in every package that uses goroutines. It is cheap, catches real bugs, and forces you to wire up context cancellation correctly.

Trap: A goroutine that loops forever without a `ctx.Done()` or `done` channel check is always a potential leak. Even if your current tests do not expose it, a future caller that cancels the operation will leave it running. Keep goroutine lifetimes simple enough that the exit paths are obvious at a glance. [***obvious-goroutine-lifetimes***]

GOMAXPROCS

Go's scheduler multiplexes goroutines onto OS threads. `GOMAXPROCS` controls how many OS threads the scheduler uses simultaneously. By default it is set to the number of logical CPUs available to the process (i.e., `runtime.NumCPU()`).

You can read or change it at runtime:

```
import "runtime"

n := runtime.GOMAXPROCS(0) // 0 means "don't change it; just return the current value"
fmt.Println("GOMAXPROCS:", n)

runtime.GOMAXPROCS(4) // limit to 4 OS threads
```

You can also set it via an environment variable before starting the process:

```
GOMAXPROCS=4 ./myserver
```

In container environments (Docker, Kubernetes), this used to be a famous trap: through Go 1.24, the default `GOMAXPROCS` read the host CPU count, not the container CPU limit, so a container limited to 2 vCPUs on a 32-core host started with `GOMAXPROCS=32`. Since Go 1.25 the runtime is container-aware on Linux: the default considers the cgroup CPU bandwidth limit and even updates periodically if the limit changes. The historical workaround, `go.uber.org/automaxprocs`, is only needed for programs built with Go 1.24 or earlier (or with `GODEBUG=containermaxprocs=0`):

```
import _ "go.uber.org/automaxprocs" // pre-1.25: sets GOMAXPROCS from cgroup quota
```

Tip: On Go 1.25+ the default `GOMAXPROCS` is correct on bare metal *and* in containers. Reach for `automaxprocs` or an explicit `GOMAXPROCS` in your deployment manifest only when you must support older toolchains.

Worker Pool

Spawning one goroutine per task is fine when tasks are cheap and bounded, but unbounded goroutine creation can exhaust memory or overwhelm a downstream service. A **worker pool** fixes the number of concurrent goroutines: a fixed set of `N` workers pull tasks off a shared jobs channel and push outcomes onto a results channel. This is the Go answer to Java's `ExecutorService` with a fixed thread pool, but built from channels and a `sync.WaitGroup` instead of a framework.

The pattern has three moving parts: the producer sends jobs and closes the jobs channel, the workers range over jobs until it is closed, and a closer goroutine waits for all workers to finish and then closes the results channel so the consumer's `range` terminates.

```
package main

import (
    "fmt"
    "sort"
    "sync"
)

// worker pulls jobs until the jobs channel is closed, then returns.
func worker(id int, jobs <-chan int, results chan<- string, wg *sync.WaitGroup) {
    defer wg.Done()
    for n := range jobs { // exits when jobs is closed and drained
        results <- fmt.Sprintf("worker %d squared %d = %d", id, n, n*n)
    }
}

func main() {
    const numWorkers = 3
    jobs := make(chan int)
    results := make(chan string)

    var wg sync.WaitGroup
    for id := 1; id <= numWorkers; id++ {
        wg.Add(1)
        go worker(id, jobs, results, &wg)
    }

    // producer: send all jobs, then close so workers can exit.
    go func() {
        for n := 1; n <= 7; n++ {
            jobs <- n
        }
        close(jobs)
    }()

    // closer: once every worker has returned, close results.
    go func() {
        wg.Wait()
        close(results)
    }()

    var out []string
    for r := range results { // drains until results is closed
        out = append(out, r)
    }
    sort.Strings(out) // worker order is nondeterministic; sort for a stable display
    for _, r := range out {
        fmt.Println(r)
    }
}
```

The key invariant: the producer closes `jobs`, which lets each worker's `range` loop return; the `WaitGroup` tracks those returns; and the closer goroutine closes `results` only after the last worker is done. Closing `results` is what lets `main`'s `range` terminate. If you forget to close `results`, `main` blocks forever on the final receive — a classic deadlock.

Trap: Do not close `results` from inside a worker. With `N` workers you would close it `N` times, and closing an already-closed channel panics. Close it exactly once, from a goroutine that waits on the `WaitGroup`.

Wut: The worker order in the output is nondeterministic — whichever worker the scheduler wakes first grabs the next job. With three workers and seven instant tasks, even the split is nondeterministic — one eager worker may grab five of the seven jobs. Only when every task takes the same non-trivial time does the distribution settle near 3/2/2.

Rate Limiting

Sometimes the problem is not too few workers but too many requests. **Rate limiting** caps how often an operation runs — protecting a downstream API, a database, or your own service from a thundering herd. Go does not need a library for the common cases; a ticker or a buffered channel is enough.

The simplest throttle uses `time.NewTicker`, which sends the current time on its channel at a fixed interval. Receiving from the ticker channel before each operation paces the loop to one operation per tick.

```
// NewTicker returns a Ticker that sends the time on its C channel every d.
func NewTicker(d Duration) *Ticker
func (t *Ticker) Stop() // stops the ticker; the channel is not closed
```

```
package main

import (
    "fmt"
    "time"
)

func main() {
    requests := []string{"As It Was", "Vampire", "Anti-Hero", "Houdini"}

    limiter := time.NewTicker(200 * time.Millisecond)
    defer limiter.Stop() // release the ticker's resources

    for _, req := range requests {
        <-limiter.C // wait for the next tick before proceeding
        fmt.Println("serving", req)
    }
}
```

This serves at most one request every 200 ms. There is also `time.Tick`, a convenience wrapper that returns just the channel — it has no `Stop`, which used to make it a leak machine.

Tip: Before Go 1.23, every `time.Tick` call leaked its ticker: there is no `Stop`, and unreferenced tickers were never garbage collected. Since Go 1.23 the GC reclaims unreferenced tickers, so `time.Tick` is safe; `time.NewTicker` plus `defer t.Stop()` is still the explicit, version-proof habit and releases the timer immediately instead of waiting for the GC.

A plain ticker throttles to a steady rate but allows no bursts. When you want to permit a short burst and then settle to a steady rate, use a **token bucket**: a buffered channel pre-filled with tokens, refilled on a ticker. Each operation takes a token (blocking if the bucket is empty); a background goroutine drops a token in on every tick.

```
package main

import (
    "fmt"
    "time"
)

func main() {
    const burst = 3
    tokens := make(chan struct{}, burst)
    for range burst {
        tokens <- struct{}{} // start full: allow an initial burst of 3
    }

    refill := time.NewTicker(200 * time.Millisecond)
    defer refill.Stop()
    go func() {
        for range refill.C {
            select {
            case tokens <- struct{}{}: // add a token if there is room
            default:                   // bucket full; drop this token
            }
        }
    }()

    for i := 1; i <= 5; i++ {
        <-tokens // take a token, blocking if none are available
        fmt.Println("request", i, "at", time.Now().Format("15:04:05.000"))
    }
}
```

The first three requests fire immediately (draining the initial burst), then the remaining two are paced at 200 ms each as the refiller tops the bucket back up. The `select` with a `default` in the refiller is what enforces the cap: if the bucket is already full, the new token is discarded rather than blocking.

Try It

Type this in and run it. It threads a request ID through a context with `WithValue`, wraps that context with a `WithTimeout`, and plays a short playlist where each track respects cancellation. Two tracks finish before the deadline; the third runs out of time.

```
package main

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

```
type reqIDKey struct{}

func withReqID(ctx context.Context, id string) context.Context {
    return context.WithValue(ctx, reqIDKey{}, id)
}

func reqID(ctx context.Context) string {
    if id, ok := ctx.Value(reqIDKey{}).(string); ok {
        return id
    }
    return "unknown"
}

func play(ctx context.Context, track string) error {
    select {
    case <-time.After(150 * time.Millisecond):
        fmt.Printf("[%s] played %q\n", reqID(ctx), track)
        return nil
    case <-ctx.Done():
        return fmt.Errorf("[%s] %q aborted: %w", reqID(ctx), track, ctx.Err())
    }
}

func main() {
    ctx := withReqID(context.Background(), "req-2026")
    ctx, cancel := context.WithTimeout(ctx, 400*time.Millisecond)
    defer cancel()

    tracks := []string{"As It Was", "Vampire", "Anti-Hero"}
    for _, t := range tracks {
        if err := play(ctx, t); err != nil {
            fmt.Println(err)
            if errors.Is(err, context.DeadlineExceeded) {
                break
            }
        }
    }
}
```

The deterministic output is the first two tracks playing, then `"Anti-Hero" aborted: context deadline exceeded`.

Try these modifications:

- Bump the timeout to `600 * time.Millisecond` and confirm all three tracks play.
- Replace `WithTimeout` with `WithCancel` and call `cancel()` from a separate goroutine after 250 ms; watch `ctx.Err()` switch from `DeadlineExceeded` to `Canceled`.
- Fetch the request ID with a plain string key (`ctx.Value("req")`) and observe that it returns `nil` — the key type matters, not just the underlying value.

Key Points

- `context.Context` carries cancellation, deadlines, and request-scoped values across goroutine boundaries; Java has no direct equivalent.

- The three context constructors are `WithCancel` (manual cancel), `WithDeadline` (absolute time), and `WithTimeout` (relative duration); always `defer cancel()`.
- Use unexported struct types as context keys to prevent collisions; never use strings or other built-in types as keys.
- The universal convention is `func Foo(ctx context.Context, ...)` — context is always the first parameter, always named `ctx`, never stored in a struct.
- `golang.org/x/sync/errgroup` is the idiomatic fan-out tool when goroutines can fail; it collects the first error and cancels the shared context.
- A worker pool fixes concurrency at `N` goroutines ranging over a shared jobs channel; the producer closes `jobs`, a `WaitGroup` tracks the workers, and a closer goroutine closes `results` exactly once after they finish.
- Rate limiting needs no library: pace a loop with `time.NewTicker` for a steady rate, or use a buffered channel as a token bucket to allow a burst before settling; `defer t.Stop()` to release the timer promptly.
- Every goroutine must have an exit path; use `go.uber.org/goleak` in tests to catch leaks automatically.
- `GOMAXPROCS` defaults to the number of logical CPUs, and since Go 1.25 it also respects container cgroup CPU limits; `go.uber.org/automaxprocs` is only needed on older toolchains.

Exercises

1. **Think about it:** In Java, cancelling an in-flight operation typically means calling `Future.cancel(true)` or interrupting a thread via `Thread.interrupt()`. Describe how Go's `context.Context` model differs from Java's thread-interrupt approach. What are the advantages of passing a context explicitly rather than relying on a thread-level interrupt mechanism? Consider what happens when a Java thread is blocked in a third-party library that does not handle `InterruptedException`, compared to how a Go function using a context-aware library would behave.

2. **What does this print?**

```go
package main

import (
    "context"
    "fmt"
    "time"
)

func work(ctx context.Context, label string) {
    select {
    case <-time.After(500 * time.Millisecond):
        fmt.Println(label, "done")
    case <-ctx.Done():
        fmt.Println(label, "cancelled:", ctx.Err())
    }
}

func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 200*time.Millisecond)
    defer cancel()

    go work(ctx, "Flaming June")
    go work(ctx, "Saltwater")
    time.Sleep(400 * time.Millisecond)
    fmt.Println("main done")
}
```

3. **Calculation:** You run a worker pool with `workers = 3` and feed it a slice of 7 tasks. Each task takes exactly 100 ms. Assuming no overhead and perfect parallelism, how many milliseconds does the pool take to complete all 7 tasks? Show your work: how many rounds of 3 concurrent workers are needed and what does each round contribute?

4. **Where is the bug?**

```
package main

import (
    "context"
    "fmt"
    "time"
)

func fetchData(url string) <-chan string {
    ch := make(chan string)
    go func() {
        time.Sleep(2 * time.Second)
        ch <- "result for " + url
    }()
    return ch
}

func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 500*time.Millisecond)
    defer cancel()

    ch := fetchData("https://example.com/songs")
    select {
    case result := <-ch:
        fmt.Println(result)
    case <-ctx.Done():
        fmt.Println("timed out")
    }
}
```

5. **Write a program:** Implement a function `fanOutFetch(ctx context.Context, songs []string) ([]string, error)` that uses `errgroup` to fetch all song titles concurrently. Simulate each fetch with a `time.Sleep` of a random duration between 50 and 150 ms (use `math/rand`). If any fetch takes longer than 300 ms total (enforced by a timeout on the context passed to `fanOutFetch`), the entire operation should be cancelled and an error returned. Print either all results in order or the cancellation error.

6. **What does this print?**

```
package main

import (
    "context"
    "fmt"
)

type ctxKey string

func main() {
    const userKey ctxKey = "user"
```

```
    ctx := context.Background()
    ctx = context.WithValue(ctx, userKey, "ana")
    ctx = context.WithValue(ctx, ctxKey("user"), "beto")

    fmt.Println(ctx.Value(userKey))
    fmt.Println(ctx.Value("user"))
}
```

Chapter 13

Packages and Modules

Go's module system replaces Maven and Gradle with a single tool that is part of the language distribution. Before modules (pre-Go 1.11), Go used a single workspace called `GOPATH` where all code — yours and every dependency — lived in one directory tree. There were no version numbers: you just got whatever was on `main` at the time of checkout. Reproducible builds were basically impossible. Modules fixed all of that: a `go.mod` file defines a self-contained unit with a name, a Go version requirement, and pinned dependency versions. This chapter covers everything you need to organize code into packages, share it across modules, manage dependencies, and control what the compiler sees at build time — from package naming conventions through build tags and embedded files.

Package Naming

In Java, package names mirror a reversed domain and tend to be long: `com.example.music.catalog`. In Go, the convention is the opposite: package names are **short, lowercase, and match their directory name**.

```
audio/       → package audio
catalog/     → package catalog
httputil/    → package httputil
```

No underscores, no `camelCase`, no reversed domains in the name itself (though the module path can contain a domain prefix — that is separate from the package name).

Tip: The package name is what callers type before the dot: `audio.Track`, `catalog.Search`. If the name is long or awkward to type repeatedly, shorten it. `util` is a warning sign — it means you have not found the right abstraction yet.

Trap: By convention the directory name and the `package` declaration at the top of each file match, with `package main` as the usual exception. The compiler does not enforce this, but a mismatch is a trap: if the directory is `catalog` and the file says `package catlog` (a typo), the import path ends in `catalog` while every caller must type `catlog.`, and tooling like `goimports` will guess the wrong qualifier. If two files in the same directory disagree on the package name, the build does break: `found packages catalog and catlog`.

When you import a package, the last segment of the import path is the package name you use in code:

```
import "github.com/angoscia/lyrics/emerald"

// package name is "emerald", not "lyrics/emerald"
fmt.Println(emerald.Verse)
```

If two imports have the same last segment, give one an alias:

```
import (
    rbutil "github.com/robertdreamhouse/children/util"
    agutil "github.com/angoscia/emerald/util"
)
```

Exported vs Unexported Symbols

Chapter 1 introduced this briefly: uppercase first letter = exported (visible outside the package); lowercase = unexported (visible only inside the package). This section recaps the rule and covers the edge cases that trip up Java programmers.

```
package catalog

type Track struct {       // exported --- callers can use catalog.Track
    Title  string         // exported field
    Artist string         // exported field
    bpm    int            // unexported --- callers cannot read or set this
}

func Search(q string) []Track { ... }   // exported function
func normalize(s string) string { ... } // unexported helper
```

There is no `protected`. Unexported means **package-local**, period. A sub-package such as `catalog/internal` is a separate package and cannot see `catalog`'s unexported names.

Wut: Struct fields are governed by the same rule. A struct literal `catalog.Track{Title: "Emerald Triangle 2012", bpm: 78}` is a compile error outside `catalog` because `bpm` is unexported. This is Go's equivalent of `private` fields combined with the rule that there is no Java-style `public Track(String title, int bpm)` constructor — callers must go through exported fields or a constructor function.

Use an unexported field with an exported accessor function when you want controlled mutation:

```
func (t *Track) BPM() int       { return t.bpm } // pointer receiver: matches SetBPM
func (t *Track) SetBPM(bpm int) { t.bpm = bpm }  // pointer receiver: mutates bpm
```

`go.mod` and `go.sum`

A **module** is the unit of versioning and distribution — roughly equivalent to a Maven artifact or a Gradle subproject. A **package** is the unit of code organization within a module — roughly equivalent to a Java package. One module contains many packages. `go.mod` sits at the root of a module and tells the Go toolchain the module's name, which version of Go it requires, and what external modules it depends on. Chapter 1 introduced `go mod init` and the basic shape of `go.mod`. This section covers the directives you will encounter in real projects.

The `module` Directive

The first line of every `go.mod` declares the module path. This is the root import path for every package in the module:

```
module github.com/darude/sandstorm

go 1.26
```

A package in `cmd/server/main.go` would belong to `github.com/darude/sandstorm/cmd/server`.

The `require` Directive

Each `require` line pins a direct dependency to an exact version:

```
require (
    github.com/robertdreamhouse/children v1.3.0
    github.com/angoscia/emeraldtriangle v2.1.0+incompatible
    golang.org/x/text v0.14.0 // indirect
)
```

`// indirect` marks a transitive dependency — one you do not import directly but that your dependencies need. `go mod tidy` adds and removes `// indirect` entries automatically.

Go resolves version conflicts using **Minimum Version Selection** (**MVS**): when multiple modules require different minimum versions of the same dependency, Go picks the highest of those minimums — the smallest version that satisfies everyone. Unlike Maven (which picks the nearest version) or npm (which can pull in duplicates), MVS always produces the same build from the same `go.mod`, with no surprises after a `go get` on an unrelated package. The trade-off is that MVS never automatically upgrades beyond what someone has explicitly required, so you have to run `go get dep@latest` intentionally when you want a newer version.

The `replace` Directive

`replace` overrides where a module is fetched from. The two common uses are local development with a forked module and pointing at an untagged local directory:

```
replace (
    github.com/robertdreamhouse/children => ../children             // local fork
    github.com/some/dep v1.2.0 => github.com/myfork/dep v1.2.1      // published fork
)
```

This is the Go equivalent of Maven's `<dependency>` with `<scope>system</scope>` or a Gradle `includeBuild` composite.

Trap: `replace` directives are respected only in the **main module** — the one whose `go.mod` is at the root of your build. If you publish a library with a `replace` directive, consumers of that library will not see the replacement.

`go.sum`

`go.sum` records the cryptographic hash of every module version ever downloaded into the build. Never edit it by hand. Commit it to source control alongside `go.mod`. It is not quite a lock file — `go.mod` already pins versions; `go.sum` is an integrity check, closer to the `integrity` hashes inside `package-lock.json` than to the lock file itself.

`go get`, `go mod tidy`, `go mod vendor`

The typical dependency workflow is: use `go get` to add or change a specific version, then `go mod tidy` to clean up any entries that are now unused or missing. `go mod vendor` is for teams that want all dependencies checked in to the repo — useful when the build environment has no network access.

Command	What it does
`go get pkg@v1.2.3`	Adds or upgrades a dependency to the specified version; updates `go.mod` and `go.sum`
`go get pkg@latest`	Upgrades to the latest tagged release
`go get pkg@none`	Removes the dependency
`go mod tidy`	Adds missing and removes unused `require` entries; updates `go.sum`
`go mod vendor`	Copies all dependencies into a `vendor/` directory for offline or audited builds

Tip: Run `go mod tidy` before every commit. It is the Go equivalent of running `mvn dependency:analyze` and then actually acting on the unused-declared warnings — except it edits the file for you.

Tip: `go mod vendor` is useful in environments where the module proxy is not accessible — CI pipelines with restricted network access, for example. Once the `vendor/` directory exists, pass `-mod=vendor` to any `go` command to use it instead of the module cache.

Internal Packages

A common problem when publishing a library is that users start importing your private helper packages even though you never intended them to be public. Once that happens you are stuck: changing the helpers is a breaking change. Go solves this with the `internal/` directory. Any package whose import path contains `internal` as a path segment can **only** be imported by code rooted at the parent of `internal`.

```
myapp/
├── go.mod
├── cmd/
│   └── server/
│       └── main.go     // can import myapp/internal/db
├── internal/
│   └── db/
│       └── db.go       // package db
└── api/
    └── handler.go      // can import myapp/internal/db
```

An external module that tries `import "myapp/internal/db"` gets a compile error:

```
use of internal package myapp/internal/db not allowed
```

This is enforced by the compiler — no workaround exists. It is stronger than Java's package-private (`default` access) because it enforces a module-level boundary, not just a package boundary.

Tip: Use `internal/` for packages that are implementation details of your module: database helpers, configuration parsers, shared types that are not part of your public API. This lets you refactor freely without worrying about breaking external callers.

Standard Project Layout

Go does not mandate a directory structure, but a widely adopted layout for applications looks like this:

```
myapp/
├── go.mod
├── go.sum
├── cmd/
│   ├── server/
│   │   └── main.go         // binary: the HTTP server
│   └── worker/
│       └── main.go         // binary: the background worker
├── internal/
│   ├── catalog/
│   │   └── catalog.go      // private business logic
│   └── db/
│       └── db.go           // private database layer
└── pkg/
    └── audio/
        └── audio.go        // public library code other modules may import
```

`cmd/` holds one directory per executable, each with its own `main.go`. `internal/` holds packages that must not leak outside this module. `pkg/` (optional) holds packages that are intentionally public — libraries other modules can import.

Tip: If you have only one binary, skip `cmd/` and put `main.go` at the root. Add `cmd/` only when you have multiple executables. If you never intend to be imported as a library, skip `pkg/` too.

Compare this to a Maven multi-module project: `cmd/server` is like a Maven module with jar packaging and a `main` class; `internal/catalog` is like a Maven module that is built as part of the reactor but never deployed to a repository — usable by sibling modules, invisible to the outside world.

Go Workspaces

Suppose you are developing two modules side by side: the main application `myapp` and a library `mylibrary` that it imports. Without workspaces you would add a `replace` directive to `myapp/go.mod` pointing at the local `mylibrary` directory, and remember to remove it before pushing. Go workspaces, introduced in Go 1.18, eliminate this dance.

Create a `go.work` file at the root of your checkout:

```
go work init ./myapp ./mylibrary
```

This generates:

```
go 1.26

use (
    ./myapp
    ./mylibrary
)
```

Depending on your installed toolchain, the `go` directive may be written with the full patch version (for example `go 1.26.3`) rather than the bare `go 1.26`; both `go work init` and `go mod init` do this, and either form is valid.

Now any `go` command run from anywhere inside that directory tree resolves `mylibrary` from the local disk, with no changes to either module's `go.mod`. When you are done, delete or ignore `go.work` — the individual

modules are unaffected.

Tip: Add `go.work` and `go.work.sum` to your `.gitignore` at the repository root. Workspaces are a local developer convenience; they should not be checked into source control for shared repositories.

Trap: `go.work` takes priority over `replace` directives. If both exist, the workspace wins. When sharing a repo, make sure `go.work` is gitignored so collaborators are not surprised.

Major Version Suffixes

Go follows semantic versioning. Versions `v0.x.x` and `v1.x.x` have the same module path. Starting at `v2`, the module path must end with the major version number:

```
module github.com/djcobra/betteroffalone/v2

go 1.26
```

Every import of that module must include `/v2`:

```
import "github.com/djcobra/betteroffalone/v2/alone"
```

This is intentional: a `v2` module is a **different module** from `v1`. Your application can import both at the same time if different dependencies require different major versions.

Wut: This surprises Java programmers. In Maven, upgrading from `1.x` to `2.x` means changing the version number in `pom.xml`; the artifact ID stays the same. In Go, upgrading from `v1` to `v2` means updating every import statement in your codebase. The rationale is that `v2` is API-incompatible by definition, so the change should be visible everywhere it matters.

Tip: If you are maintaining a library and want to publish a `v2`, the easiest path is to create a `v2/` subdirectory at the module root, copy the code there, update the `module` line to end in `/v2`, and maintain both versions side by side. The alternative is to tag the root module at `v2.0.0` and update the `go.mod` there, but the subdirectory approach keeps the history clean.

Build Tags

A **build tag** (also called a build constraint) tells the Go compiler to include or exclude a file from a build. The most common uses are platform-specific code, feature flags, and separating integration tests from unit tests.

Syntax

Place the constraint near the top of the file, before the `package` clause and preceded only by blank lines or other line comments (a license header is fine), with a blank line between the constraint and the `package` clause:

```
//go:build linux

package platform
```

The expression can use `&&`, `||`, and `!`:

```
//go:build linux && amd64
```

```
//go:build !windows
```

Predefined Tags

The Go toolchain defines tags automatically:

Tag	When true
`linux`, `darwin`, `windows`	GOOS matches
`amd64`, `arm64`	GOARCH matches
`go1.21`, `go1.22`, …	Go version is at least that release
`cgo`	cgo is enabled

Custom Tags

Define your own tags by passing `-tags` to the `go` command:

```
//go:build integration
```

```
go test -tags=integration ./...
```

Files with the `integration` constraint are excluded from ordinary builds and compiled only when you pass `-tags=integration`. This is how integration tests are kept separate from unit tests without putting them in a different directory.

Tip: Use build tags to separate slow integration tests from fast unit tests. Name the tag `integration` and document it in your README. Your CI pipeline can run `go test ./...` for fast tests on every commit and `go test -tags=integration ./...` on a slower schedule.

Trap: Before Go 1.17 the syntax was `// +build linux` (a comment, not a directive). You may still encounter this in older code. The old syntax is still accepted for compatibility, but `//go:build` is the modern form. Do not mix them in the same file.

`//go:embed`

Before Go 1.16, embedding static assets — HTML templates, SQL schemas, configuration files — in a binary required third-party code generators or reading files at runtime. `//go:embed` makes this a first-class language feature.

Embedding a Single File

```
package lyrics

import _ "embed"

//go:embed emerald.txt
var emeraldLyrics string
```

At compile time, the contents of `emerald.txt` are baked into the binary and assigned to `emeraldLyrics`. No file I/O at runtime.

Embedding Multiple Files

```
package web

import "embed"

//go:embed static/*.html static/*.css
var webFiles embed.FS
```

`embed.FS` is a read-only filesystem rooted at the directory containing the `.go` file. It satisfies `fs.FS`, so it works with `http.FS`, `template.ParseFS`, and any other function that accepts an `fs.FS`.

```
http.Handle("/static/", http.StripPrefix("/static/", http.FileServer(http.FS(webFiles))))
```

Tip: The `//go:embed` directive must immediately precede the variable's declaration; only blank lines and `//` line comments may sit between them. An intervening declaration either steals the directive (`go:embed cannot apply to var of type int`) or, for non-var lines, fails with a `misplaced go:embed directive` error. When embedding into a `string` or `[]byte` variable, the `embed` package must be imported for its side effect — use `import _ "embed"` because you are not naming anything from the package. When embedding into `embed.FS`, use a normal `import "embed"` because you reference `embed.FS` by name.

Trap: Glob patterns in `//go:embed` do not match files or directories whose names begin with `.` or `_`. If you want to embed a `.gitignore` or `_headers` file, name it explicitly in the pattern (this works with any variable type, including `string`), or use the `all:` prefix when embedding a whole directory tree (`//go:embed all:static`).

The compile-time inclusion is the key difference from Java's `getClass().getResourceAsStream();` with `//go:embed` there is no path to get wrong at runtime.

Try It

Type this in and run it to see `//go:embed` bake a file into the binary at compile time. Create a file named `tracks.txt` next to `main.go` containing three lines: `Emerald Triangle 2012`, `Sandstorm`, and `Better Off Alone`, then run `go run .` (a `go.mod` is required, so `go mod init example.com/tryit` first).

```
package main

import (
    "embed"
    "fmt"
    "io/fs"
    "strings"
)

//go:embed tracks.txt
var trackList string

//go:embed *.txt
var assets embed.FS

func main() {
    titles := strings.Split(strings.TrimSpace(trackList), "\n")
    fmt.Printf("embedded %d titles:\n", len(titles))
```

```
    for i, t := range titles {
        fmt.Printf("  %d. %s\n", i+1, t)
    }

    // embed.FS satisfies fs.FS, so we can walk it at runtime.
    fs.WalkDir(assets, ".", func(path string, d fs.DirEntry, err error) error {
        if err == nil && !d.IsDir() {
            fmt.Println("asset file:", path)
        }
        return nil
    })
}
```

Try these modifications:

- Insert another declaration (for example `const placeholder = 1`) between the `//go:embed` directive and the `var` declaration it applies to, then rebuild — watch it fail with `misplaced go:embed directive`. (A blank line alone is fine; the directive only needs to be the line immediately above the declaration, ignoring blanks and `//` comments.)
- Add a second `.txt` file and confirm the `embed.FS` walk finds it without any code change, while the `string` variable still holds only `tracks.txt`.
- Rename the second `.txt` file from the previous bullet to start with an underscore (say `_extra.txt`) and observe that the `*.txt` glob silently skips it in the walk output while the build still succeeds — the explicit `tracks.txt` embed is unaffected.

Key Points

- Package names are short, lowercase, match their directory, and no underscores.
- Capitalization controls visibility: uppercase = exported, lowercase = unexported; there is no `protected`.
- A module is the unit of versioning (one `go.mod`); a package is the unit of code organization within a module.
- `go.mod` declares the module path and pins dependencies with `require`; `replace` overrides the source of a module for local development or forks.
- Go uses Minimum Version Selection (MVS): the highest minimum version required by any module in the build graph wins — reproducible by design.
- `go.sum` records checksums that verify downloaded modules; commit it alongside `go.mod`.
- `go get` adds/upgrades/removes dependencies; `go mod tidy` keeps `go.mod` clean; `go mod vendor` copies dependencies locally.
- `internal/` packages are enforced by the compiler: only code rooted at the parent of `internal` may import them.
- The standard layout uses `cmd/` for executables and `internal/` for private packages.
- Go workspaces (`go work`) let you develop multiple modules side by side without `replace` directives.
- Modules at `v2` or higher must include the major version in the module path and every import.
- Build tags (`//go:build`) include or exclude files based on OS, architecture, Go version, or custom tags passed with `-tags`.
- `//go:embed` bakes files into the binary at compile time; use `string`, `[]byte`, or `embed.FS` as the variable type.

Exercises

1. **Think about it:** Maven and Gradle resolve transitive dependencies automatically and let two artifacts declare conflicting version requirements for the same library. They use a strategy (nearest-wins in Maven, highest-requested in Gradle) to pick a single version at build time. Go's module system takes a

different approach called Minimum Version Selection (MVS): it always picks the minimum version that satisfies all requirements. Compare these two philosophies. What problems does MVS avoid? What does it make harder? When might the Go approach cause a surprise after running `go get pkg@latest`?

2. **Where is the bug?**

   Given the following three files in a module `github.com/angoscia/demo`:

   File `lyrics/lyrics.go`:

   ```
   package lyrics

   import "fmt"

   func Print() {
       fmt.Println("Emerald Triangle 2012")
   }
   ```

   File `lyrics/internal/detail/detail.go`:

   ```
   package detail

   import "fmt"

   func Show() {
       fmt.Println("internal detail")
   }
   ```

   File `main.go`:

   ```
   package main

   import (
       "github.com/angoscia/demo/lyrics"
       "github.com/angoscia/demo/lyrics/internal/detail"
   )

   func main() {
       lyrics.Print()
       detail.Show()
   }
   ```

   What happens when you run `go build`? If the build succeeds, what does the program print? If not, explain why.

3. **Calculation:** A module's `go.mod` contains the following:

   ```
   module github.com/angoscia/app

   go 1.26

   require (
       github.com/angoscia/audio v1.4.0
       github.com/angoscia/catalog v0.9.2
       golang.org/x/text v0.14.0 // indirect
   )
   ```

   The `audio` module at `v1.4.0` itself requires `golang.org/x/text v0.12.0`. The `catalog` module at `v0.9.2` requires `golang.org/x/text v0.14.0`.

Under Go's Minimum Version Selection, which version of `golang.org/x/text` will the final build use? Explain why. Now suppose you add a new dependency that requires `golang.org/x/text v0.16.0`. What version will MVS select then?

4. **What does this print?** A single-file `package main` contains the following. Predict the exact output, then explain the order in which the package-level `var` declarations and the `init` function run.

```go
package main

import "fmt"

var a = b + c
var b = f()
var c = 2

func f() int {
    fmt.Println("f called")
    return 3
}

func init() {
    fmt.Println("init, a =", a)
}

func main() {
    fmt.Println("main, a =", a)
}
```

5. **Where is the bug?** The following module has this layout and code:

```
betteroffalone/
├── go.mod           (module github.com/djcobra/betteroffalone)
├── main.go
└── internal/
    └── config/
        └── config.go
```

`main.go`:

```go
package main

import (
    "fmt"
    "github.com/djcobra/betteroffalone/internal/config"
)

func main() {
    fmt.Println(config.DefaultRegion)
}
```

A second module lives alongside it:

```
player/
├── go.mod           (module github.com/djcobra/player)
└── main.go
```

`player/main.go`:

```go
package main

import (
    "fmt"
    "github.com/djcobra/betteroffalone/internal/config"
)

func main() {
    fmt.Println(config.DefaultRegion)
}
```

What happens when you run `go build ./...` inside the `player/` module? Identify the bug and describe how to fix it without moving the `config` package out of `internal/`.

6. **Write a program:** Create a small multi-package module with the following layout:

```
children/
├── go.mod            (module github.com/robertdreamhouse/children)
├── main.go
├── tracks/
│   └── tracks.go
└── internal/
    └── format/
        └── format.go
```

`tracks.go` should define an exported `Track` struct with `Title` and `Artist` string fields and a slice `Catalog` containing at least two entries. `format.go` should define an unexported-to-outside but exported-within-module function `Label(t tracks.Track) string` that returns `"Title by Artist"`. `main.go` should import both `tracks` and `internal/format`, iterate over `tracks.Catalog`, and print the label for each track using `format.Label`. Build and run the program with `go run ./...` (or `go run main.go`) and confirm it prints the expected output.

Chapter 14

Essential Standard Library

Without the standard library you would hand-roll buffered I/O, rewrite time parsing from scratch, build your own structured logger, and wire up command-line flags by hand — all before writing a single line of business logic. Go's standard library covers most of what you reach for in daily backend work — I/O, file access, time, logging, CLI flags, pattern matching, and more — without pulling in external dependencies. The Java equivalents (`BufferedReader`, `SimpleDateFormat`, SLF4J, Apache Commons CLI) often need boilerplate that Go's built-ins deliberately eliminate. This chapter walks through the packages every Go programmer uses constantly, with Java comparisons where the mental model transfer is non-obvious.

`fmt` — Revisited

Chapter 1 introduced `fmt.Println`, `fmt.Printf`, and `fmt.Sprintf`. This section adds the verbs and writer-directed functions you will use daily.

Diagnostic Verbs

Three verbs are especially useful for debugging:

Verb	Output
`%v`	Default format — values only
`%+v`	Struct with field names
`%#v`	Go-syntax representation; can paste back in code
`%T`	Go type name

```
type Track struct {
    Title  string
    Artist string
    BPM    int
}

t := Track{Title: "Crazy Train", Artist: "Ozzy Osbourne", BPM: 114}
fmt.Printf("%v\n",  t)   // {Crazy Train Ozzy Osbourne 114}
fmt.Printf("%+v\n", t)   // {Title:Crazy Train Artist:Ozzy Osbourne BPM:114}
fmt.Printf("%#v\n", t)   // main.Track{Title:"Crazy Train", Artist:"Ozzy Osbourne", BPM:114}
fmt.Printf("%T\n",  t)   // main.Track
```

`%+v` is your first move when inspecting an unknown struct value. `%#v` gives you a snippet you can paste directly into a test or a `var` declaration.

Tip: `%#v` on a slice prints `[]int{1, 2, 3}` rather than `[1 2 3]`. It is more verbose but unambiguous — very useful in test failure messages.

`fmt.Fprintf` to Any `io.Writer`

```
func Fprintf(w io.Writer, format string, a ...any) (n int, err error)
```

`fmt.Printf(format, args...)` is simply `fmt.Fprintf(os.Stdout, format, args...)`. Passing any `io.Writer` redirects the output:

```
fmt.Fprintf(os.Stderr, "warn: retrying in %v\n", delay)
fmt.Fprintf(logFile, "[INFO] loaded %d tracks\n", n)
fmt.Fprintf(&buf, "SELECT * FROM tracks WHERE bpm > %d", minBPM)
```

Because `*os.File`, `*bytes.Buffer`, `*strings.Builder`, `net.Conn`, and `http.ResponseWriter` all satisfy `io.Writer`, a single `Fprintf` call works with any of them.

Tip: In Java you might have separate `PrintStream`, `PrintWriter`, and `StringBuilder.append` paths. In Go there is one path: accept an `io.Writer`, call `fmt.Fprintf`.

`io` — The Glue of Go I/O

The `io` package defines the core interfaces that tie all I/O in Go together. You saw `io.Reader` and `io.Writer` in Chapter 8; this section covers the functions that compose them.

`io.ReadAll` and `io.Copy`

```
func ReadAll(r Reader) ([]byte, error)              // read r until EOF; return all bytes
func Copy(dst Writer, src Reader) (int64, error)    // copy src to dst until EOF or error
```

`io.ReadAll` is the Go equivalent of Java's `InputStream.readAllBytes()` (Java 9+). Use it when you need the entire contents in memory:

```
data, err := io.ReadAll(resp.Body)
```

`io.Copy` streams from a `Reader` to a `Writer` without loading everything into memory at once. It is Go's answer to Java's `InputStream.transferTo(OutputStream)`:

```
n, err := io.Copy(dst, src)  // n is the number of bytes copied
```

Trap: `io.ReadAll` on a large HTTP response body allocates the entire response in memory. For large or unbounded responses, prefer `io.Copy` to stream directly to a file or another writer.

Composing Readers and Writers

The `io` package provides several functions that wrap and combine readers and writers without copying data:

```
func MultiReader(readers ...Reader) Reader // reads from each reader in sequence
func MultiWriter(writers ...Writer) Writer // writes to all writers simultaneously
func TeeReader(r Reader, w Writer) Reader  // returns reader that copies to w as it reads
```

```
func LimitReader(r Reader, n int64) Reader // reads at most n bytes from r
func Pipe() (*PipeReader, *PipeWriter)     // creates synchronous in-memory pipe
```

`MultiReader` is like Java's `SequenceInputStream`. `MultiWriter` is like Apache Commons' `TeeOutputStream` but for any number of writers. `TeeReader` is useful for logging raw bytes while processing them:

```
// Read from r and simultaneously write everything read to log.
logged := io.TeeReader(r, log)
io.Copy(processor, logged)
```

`LimitReader` prevents reading past a byte budget:

```
limited := io.LimitReader(req.Body, 1<<20) // never read more than 1 MiB
data, err := io.ReadAll(limited)
```

`io.Pipe` creates a synchronous in-memory pipe: writes to the `PipeWriter` block until the `PipeReader` consumes them. It connects a producer goroutine to a consumer that expects an `io.Reader`, without a buffer:

```
pr, pw := io.Pipe()
go func() {
    fmt.Fprintln(pw, "The Sound of Silence")  // Disturbed
    pw.Close()
}()
data, _ := io.ReadAll(pr)
fmt.Println(string(data))  // The Sound of Silence
```

Tip: The `io` composition functions never allocate a large intermediate buffer. They are building blocks for streaming pipelines: `LimitReader` guards against oversized input, `TeeReader` adds tap-style logging, and `MultiWriter` fans out to several sinks.

Other `io` Signatures

```
func ReadFull(r Reader, buf []byte) (n int, err error)  // reads exactly len(buf) bytes
func WriteString(w Writer, s string) (n int, err error) // writes a string to w
var  Discard io.Writer                                  // discards all writes; useful in tests
```

`io.Discard` is a writer that throws data away — handy in tests when you want to drain a reader without storing the bytes.

`bufio` — Buffered I/O

Raw `io.Reader` and `io.Writer` operations may call the OS for every byte. `bufio` wraps any reader or writer in a userspace buffer, batching syscalls for performance. The Java equivalent is `BufferedReader` / `BufferedWriter`.

`bufio.Scanner`

`Scanner` is the idiomatic way to read text line by line (or word by word, or any custom token):

```
func NewScanner(r io.Reader) *Scanner
```

```
scanner := bufio.NewScanner(os.Stdin)
for scanner.Scan() {          // Scan returns false at EOF or error
    line := scanner.Text()    // current line, no trailing newline
    fmt.Println(line)
    fmt.Println(scanner.Text()) // prints same line again
}
```

```
if err := scanner.Err(); err != nil {
    log.Fatal(err)
}
```

`Scan()` tries to read the next line from input, and `Text()` returns the current line but doesn't read the next lines, so calling `Text()` twice will print the current line twice. The loop pattern — `for scanner.Scan()` — is not how Java's `java.util.Scanner` works, where `nextLine()` both returns the line and advances. The `Matcher` class in the Java regular expression library does have a similar usage; `Matcher.find()` advances the match and `Matcher.group()` returns the text of the current match.

Real programs usually collect or process lines inside the loop rather than just echoing them. This example reads track names from stdin, skips blank lines, and prints a numbered playlist when input ends:

```
scanner := bufio.NewScanner(os.Stdin)
var tracks []string
for scanner.Scan() {
    line := scanner.Text()
    if line != "" {
        tracks = append(tracks, line)
    }
}
if err := scanner.Err(); err != nil {
    log.Fatal(err)
}
fmt.Printf("playlist: %d tracks\n", len(tracks))
for i, t := range tracks {
    fmt.Printf("%2d. %s\n", i+1, t)
}
```

Piping three song titles in:

```
$ printf "Crazy Train\nThe Sound of Silence\nCafé Del Mar\n" | ./play
playlist: 3 tracks
 1. Crazy Train
 2. The Sound of Silence
 3. Café Del Mar
```

The `append` call inside the loop and the `range` loop after it are covered in Chapter 7, but the shape is easy to read now: accumulate lines during the scan, process the slice once scanning is done.

Trap: Always check `scanner.Err()` after the loop. `Scan` returns `false` both at EOF (no error) and on a read error. Skipping the check silently drops I/O errors. [***no-discard-error***]

By default `Scanner` splits on newlines. You can change the split function:

```
scanner.Split(bufio.ScanWords)   // split on whitespace
scanner.Split(bufio.ScanBytes)   // one byte at a time
scanner.Split(bufio.ScanRunes)   // one UTF-8 rune at a time
```

`Scanner` enforces a default maximum token size of 64 KiB; the initial buffer is much smaller (4 KiB) and grows as needed up to that limit.

```
func (s *Scanner) Buffer(buf []byte, max int) // set the starting buffer and the max token size
```

For very long lines, call `scanner.Buffer(make([]byte, cap), cap)` before the first `Scan`.

Tip: The `max` argument to `scanner.Buffer` is a hard ceiling on token size, not just a starting size. Raising it lets you read legitimately long lines, but set it no higher than you actually need: with untrusted input, a single delimiter-less "runaway" line would otherwise make `Scanner` grow its buffer all the way up to that ceiling. When a token exceeds the limit, `Scan` returns `false` and `scanner.Err()` reports `bufio.ErrTooLong` — a clean, bounded failure instead of letting one giant line balloon your memory.

`bufio.NewReader` and `bufio.NewWriter`

```
func NewReader(rd io.Reader) *Reader    // wraps rd in a 4096-byte read buffer
func NewWriter(w io.Writer) *Writer     // wraps w in a 4096-byte write buffer
```

Use `bufio.NewReader` when you need `ReadString`, `ReadByte`, or `Peek` on an existing `io.Reader`:

```
br := bufio.NewReader(conn)
header, err := br.ReadString('\n')  // read up to and including the newline
```

Use `bufio.NewWriter` to batch small writes to an expensive underlying writer (a file or network connection). You **must** call `Flush` when done or buffered data will be silently discarded:

```
bw := bufio.NewWriter(file)
fmt.Fprintln(bw, "Zombie")   // Andrew Spencer --- written to buffer, not file yet
bw.Flush()                   // now the data reaches the file
```

Trap: Forgetting `bw.Flush()` is one of the most common Go I/O bugs. Use `defer bw.Flush()` immediately after creating the `bufio.Writer` so you cannot forget it.

`os` — Files and the Process Environment

Opening and Creating Files

```
func Open(name string) (*File, error)                                  // open for reading only
func Create(name string) (*File, error)                                // create or truncate (R/W)
func OpenFile(name string, flag int, perm FileMode) (*File, error) // full flag/perm control
func ReadFile(name string) ([]byte, error)                             // read entire file at once
func WriteFile(name string, data []byte, perm FileMode) error      // write data, creating file
```

For most file operations you need just two patterns:

Read the whole thing:

```
data, err := os.ReadFile("playlist.json")
if err != nil {
    return fmt.Errorf("reading playlist: %w", err)
}
```

Stream a large file:

```
f, err := os.Open("tracks.csv")
if err != nil {
    return err
}
defer f.Close()
scanner := bufio.NewScanner(f)
for scanner.Scan() {
```

```
    process(scanner.Text())
}
```

`*os.File` satisfies both `io.Reader` and `io.Writer`, so it plugs directly into any function that accepts those interfaces.

Trap: Always `defer f.Close()` immediately after a successful `os.Open` or `os.Create`, unless ownership of the `*os.File` is handed off — returned to the caller or passed to a goroutine that outlives the current function. When ownership is transferred, whoever takes it is responsible for closing the file. This is similar to Java's recommendation to use `try-with-resources` when possible. Just like Java, if you forget to close a file you risk running out of file descriptors if the garbage collector doesn't collect the unused `*os.File` objects fast enough.

`os.WriteFile` is the simplest way to replace a small file:

```
err := os.WriteFile("config.json", data, 0o644)
```

Process Environment

```
var Args []string                    // command-line arguments; Args[0] is the program name
var Stdin  *File                     // standard input
var Stdout *File                     // standard output
var Stderr *File                     // standard error
func Getenv(key string) string       // returns the value of an environment variable
func LookupEnv(key string) (string, bool) // like Getenv but distinguishes missing from empty
```

`os.Stdin`, `os.Stdout`, and `os.Stderr` are ordinary `*os.File` values that satisfy `io.Reader` / `io.Writer`. That is why `fmt.Fprintln(os.Stderr, msg)` works without any special cast.

```
token := os.Getenv("API_TOKEN")
if token == "" {
    fmt.Fprintln(os.Stderr, "API_TOKEN not set")
    os.Exit(1)
}
```

`os.Args[0]` is the program name; `os.Args[1:]` are the user-supplied arguments — identical to Java's `String[] args` but global. For real CLI tools, use the `flag` package (see below) rather than parsing `os.Args` by hand.

os/exec — Running Subprocesses

The `os/exec` package runs external programs. The Java counterpart is `ProcessBuilder`.

```
func Command(name string, arg ...string) *Cmd   // builds a Cmd ready to run
```

`Command` returns a `*Cmd` you configure and then execute. There are four execution methods that differ in how much they do for you:

```
func (c *Cmd) Start() error                  // start the process and return immediately
func (c *Cmd) Wait() error                   // block until a Start()ed process finishes
func (c *Cmd) Run() error                    // Start + Wait; use when you don't need output
func (c *Cmd) Output() ([]byte, error)       // Start + Wait + return stdout as []byte
func (c *Cmd) CombinedOutput() ([]byte, error) // Start + Wait + return stdout and stderr merged
```

`Start()` and `Wait()` are the low-level pair for when you need manual control — everything else is a convenience wrapper around them.

```go
// Run a command and capture its standard output.
out, err := exec.Command("git", "rev-parse", "--short", "HEAD").Output()
if err != nil {
    log.Fatal(err)
}
fmt.Printf("commit: %s", out)
```

Use `Cmd.Run()` when you don't need to capture output — you just want to know if it succeeded:

```go
cmd := exec.Command("gofmt", "-w", "main.go")
cmd.Stdout = os.Stdout   // forward subprocess stdout to our stdout
cmd.Stderr = os.Stderr   // forward subprocess stderr to our stderr
if err := cmd.Run(); err != nil {
    log.Fatalf("gofmt failed: %v", err)
}
```

Piping stdio

`Cmd.Stdin`, `Cmd.Stdout`, and `Cmd.Stderr` are `io.Reader` / `io.Writer` fields. Assign any compatible value:

```go
cmd := exec.Command("sort")
cmd.Stdin  = strings.NewReader("banana\napple\ncherry\n")
cmd.Stdout = os.Stdout
cmd.Run()
// apple
// banana
// cherry
```

For a pipeline, connect one command's stdout to another's stdin using `io.Pipe` or `cmd.StdoutPipe()`:

```go
c1 := exec.Command("cat", "tracks.txt")
c2 := exec.Command("grep", "pop")
c2.Stdin, _ = c1.StdoutPipe()   // c2 reads from c1's stdout
c2.Stdout = os.Stdout
c2.Start()
c1.Run()
c2.Wait()
```

The `_` on `StdoutPipe()` discards an error for brevity — in real code, check it (`StdoutPipe` fails if `Cmd.Stdout` was already set or if `Start` has already been called), and check the errors from `Start`, `Run`, and `Wait` too.

Trap: `Run()`, `Output()`, and `CombinedOutput()` call `Start()` and `Wait()` internally. If you call `Start()` yourself and then call one of these methods, the command will fail because `Start()` was already called. Use `Start()` + `Wait()` only when you need manual control (e.g., reading stdout mid-run with `StdoutPipe()`); otherwise use the convenience methods.

Trap: `exec.Command` does not invoke a shell. `exec.Command("ls -la")` does **not** work — it looks for a program literally named `"ls -la"`. Pass each argument separately: `exec.Command("ls", "-la")`.

`flag` — CLI Flag Parsing

The `flag` package parses command-line flags in the style of Go tools themselves. The Java equivalent is Apache Commons CLI or picocli, but `flag` is built in and needs no dependency.

```
func Bool(name string, value bool, usage string) *bool      // define a boolean flag
func Int(name string, value int, usage string) *int         // define an integer flag
func String(name string, value string, usage string) *string // define a string flag
func Parse()                                                 // parse os.Args[1:]
```

Say you are building a `play` command that reads a playlist file and prints the tracks. Define the flags as package-level variables so they are initialized before `main` runs:

```
var (
    shuffle = flag.Bool("shuffle", false, "shuffle playback order")
    repeat  = flag.Int("repeat", 1, "number of times to play the playlist")
    format  = flag.String("format", "m3u", "output format: m3u or json")
)

func main() {
    flag.Parse()  // must call before reading flag values
    fmt.Printf("shuffle=%v repeat=%d format=%s\n", *shuffle, *repeat, *format)
    fmt.Println("playlist:", flag.Args()) // positional args after the flags
}
```

Running `./play -shuffle -repeat 3 bangers.m3u` sets `*shuffle` to `true`, `*repeat` to `3`, and leaves `"bangers.m3u"` as a positional argument.

Processing Positional Arguments

Everything left on the command line after the flags is a **positional** (non-flag) argument — the file names, sub-commands, or other operands your program actually acts on. Three functions read them back:

```
func Args() []string   // all positional arguments, in order
func Arg(i int) string // the i-th positional argument, "" if out of range
func NArg() int        // how many positional arguments there are
```

`flag.Args()` returns a plain `[]string`, so when you accept a variable number of operands you just range over it:

```
flag.Parse()
for i, path := range flag.Args() {
    fmt.Printf("playlist %d: %s\n", i+1, path)
}
```

When you expect a fixed shape — say a sub-command followed by a file — check the count first, then index directly with `flag.Arg`:

```
if flag.NArg() < 2 {
    flag.Usage()
    os.Exit(1)
}
cmd  := flag.Arg(0) // e.g. "play"
file := flag.Arg(1) // e.g. "bangers.m3u"
```

`flag.Arg(i)` never panics: an out-of-range index returns `""` rather than crashing, so guarding with `flag.NArg()` is about giving the user a clear error, not about avoiding a slice-bounds panic.

Trap: `flag.Parse()` stops at the **first** non-flag argument, and everything after it — even a token that looks like `-shuffle` — is treated as positional. So `./play bangers.m3u -shuffle` leaves `-shuffle` unparsed in `flag.Args()` and `*shuffle` stays `false`. Put your flags **before** the positional arguments, or use the `--` terminator to mark the boundary explicitly.

Customizing the Help Message

Running `./play -help` prints something like:

```
Usage of ./play:
  -format string
        output format: m3u or json (default "m3u")
  -repeat int
        number of times to play the playlist (default 1)
  -shuffle
        shuffle playback order
```

That output lists flags, but it says nothing about the required `<playlist>` positional argument. `flag.Usage` is a package-level variable holding the function the package calls when it needs to print help — on `-help`, `-h`, or a parse error. Override it to add your own synopsis line:

```
var Usage func() // called by flag.Parse on -help or a parse error
```

```
func main() {
    flag.Usage = func() {
        fmt.Fprintf(os.Stderr, "Usage: %s [flags] <playlist>\n", os.Args[0])
        flag.PrintDefaults() // prints the standard flag table
    }
    flag.Parse()
    if flag.NArg() < 1 {
        flag.Usage()
        os.Exit(1)
    }
    // ...
}
```

Now `-help` prints:

```
Usage: ./play [flags] <playlist>
  -format string
        output format: m3u or json (default "m3u")
  -repeat int
        number of times to play the playlist (default 1)
  -shuffle
        shuffle playback order
```

Assign `flag.Usage` before calling `flag.Parse()`. Write to `os.Stderr` — help text goes to stderr so it does not pollute stdout when the caller pipes output. `flag.PrintDefaults()` renders the standard flag table so you get the flag descriptions for free without rewriting them. `os.Args[0]` gives the actual binary name whether the program is run with `go run` or as an installed binary. `flag.NArg()` returns the count of positional arguments; call `flag.Usage()` yourself when required arguments are missing.

Tip: Call `flag.Parse()` in `main`, not in `init`. Calling it from `init` makes testing harder because `init` runs before your test code can set up the argument list.

Tip: `flag` is deliberately minimal — no sub-commands, no `--long-form` aliases, no automatic `--help` grouping, no shell completion. If your CLI needs any of those, reach for Cobra. Cobra is what the cool devs use: kubectl, Docker, the GitHub CLI, and the Go toolchain itself are all built with it. It is a drop-in upgrade — you still define flags the same way, but you get sub-commands, persistent flags, generated man pages, and shell completion for free.

`time` — Clocks, Durations, and Timers

`time.Duration` and `time.Time`

`time.Duration` is an `int64` counting nanoseconds. Named constants give you human-readable literals:

```
const (
    Nanosecond  Duration = 1
    Microsecond          = 1000 * Nanosecond
    Millisecond          = 1000 * Microsecond
    Second               = 1000 * Millisecond
    Minute               = 60 * Second
    Hour                 = 60 * Minute
)
```

You write durations as arithmetic:

```
timeout := 30 * time.Second   // 30s
jitter  := 500 * time.Millisecond
```

In Java you use `Duration.ofSeconds(30)` and `TimeUnit.MILLISECONDS`. Go's arithmetic on named constants is more concise.

`time.Time` represents an instant in time with nanosecond precision. The zero value (`time.Time{}`) is January 1, year 1, UTC — not `null`.

```
now := time.Now()                          // current local time
fmt.Println(now.Format(time.RFC3339))      // 2024-11-22T15:04:05-08:00
fmt.Println(now.UTC().Format(time.RFC3339)) // same instant in UTC: ends in Z, not an offset
```

Wut: Go's reference time for layout strings is `Mon Jan 2 15:04:05 MST 2006` — a specific instant, not placeholder tokens like Java's `yyyy-MM-dd`. `time.RFC3339` is the constant `"2006-01-02T15:04:05Z07:00"`. If your format string looks wrong, check that you used the reference digits (month=01, day=02, hour=15, minute=04, second=05, year=2006).

`time.Now`, `time.Since`, and `time.After`

```
func Now() Time                           // current time
func Since(t Time) Duration               // elapsed time since t; equivalent to Now().Sub(t)
func After(d Duration) <-chan Time         // returns a channel that receives after duration d
func Sleep(d Duration)                    // blocks the calling goroutine for d
```

```
start := time.Now()
doWork()
fmt.Printf("finished in %v\n", time.Since(start))
```

`time.After` is the idiomatic timeout in a `select`:

```
select {
case result := <-work:
    fmt.Println("got result:", result)
case <-time.After(5 * time.Second):
    fmt.Println("timed out")
}
```

`time.Ticker` and `time.Timer`

```
func NewTicker(d Duration) *Ticker  // fires every d; Ticker.C is the receive channel
func NewTimer(d Duration) *Timer    // fires once after d; Timer.C is the receive channel
```

`time.Ticker` delivers a tick on its channel at every interval — the same idea as calling `scheduleAtFixedRate` on Java's `ScheduledExecutorService`.

```
ticker := time.NewTicker(1 * time.Second)
defer ticker.Stop()  // release the timer when you are done

for range ticker.C {
    fmt.Println("tick:", time.Now().Format(time.TimeOnly))
}
```

Note that in this snippet the `defer ticker.Stop()` never actually runs, because the loop never terminates and the function never returns. It is there for good habit — if the loop ever gains an exit condition (a `break` or a `return`), the `defer` is already in place to clean up.

`time.Timer` fires once after a delay. If you do not need the channel — you just want to pause — use `time.Sleep` instead.

Trap: Call `ticker.Stop()` (or `timer.Stop()`) when you are done with a Ticker or Timer. While the Ticker is still referenced, an unstopped ticker keeps its runtime timer firing and allocating. (Since Go 1.23 an unreferenced Ticker is garbage-collected even without `Stop`, but calling `Stop` releases the timer immediately and is still the idiom.)

`path/filepath` — Cross-Platform Path Manipulation

`path/filepath` handles OS path separators correctly (\ on Windows, / on Unix), unlike the `path` package which is hardcoded to forward slashes. The Java equivalent is `java.nio.file.Path`.

```
func Join(elem ...string) string        // joins path elements with the OS separator
func Dir(path string) string            // returns all but the last element of path
func Base(path string) string           // returns the last element of path
func Ext(path string) string            // returns the file name extension including the dot
func Abs(path string) (string, error)   // returns the absolute path
func Walk(root string, fn WalkFunc) error           // walks the file tree (pre-order)
func WalkDir(root string, fn fs.WalkDirFunc) error  // like Walk but more efficient; preferred
```

```
p := filepath.Join("music", "trance", "sandstorm.flac")
fmt.Println(filepath.Dir(p))   // music/trance
fmt.Println(filepath.Base(p))  // sandstorm.flac
fmt.Println(filepath.Ext(p))   // .flac
```

`filepath.WalkDir` recursively visits every file in a tree:

```
filepath.WalkDir(".", func(path string, d fs.DirEntry, err error) error {
    if err != nil {
        return err
    }
    if !d.IsDir() {
        fmt.Println(path)
    }
    return nil
})
```

Note that `fs.DirEntry` and `fs.WalkDirFunc` live in the `io/fs` package, so this code needs `import "io/fs"` alongside `path/filepath`.

Tip: Use `filepath.Join` instead of string concatenation for paths. It handles leading/trailing separators, double slashes, and platform differences automatically. `filepath.Join("music/", "/pop")` returns `"music/pop"`, not `"music//pop"`.

`log/slog` — Structured Logging

Go's classic `log` package writes plain text lines: great for a quick script, hopeless in production. When your service runs on twenty containers and emits ten thousand lines per minute, you need to search by field, not by regex. **Structured logging** — attaching typed key-value pairs to every log record — makes that possible. Go 1.21 added `log/slog` as the standard structured logger. The Java equivalent is SLF4J + Logback or Log4j2 with a JSON appender, but `slog` is simpler to configure and ships with the standard library.

Handlers and Output Formats

A **handler** controls where and how records are written. `slog` ships two:

```
// logfmt-style text --- readable in a terminal
logger := slog.New(slog.NewTextHandler(os.Stderr, nil))
// time=2026-05-30T15:04:05Z level=INFO msg="track loaded" title="Café Del Mar" bpm=130

// JSON --- machine-readable; use in production services
logger := slog.New(slog.NewJSONHandler(os.Stderr, nil))
// {"time":"2026-05-30T15:04:05Z","level":"INFO","msg":"track loaded",
//   "title":"Café Del Mar","bpm":130}
```

Tip: Use `TextHandler` during development (it's readable in a terminal) and `JSONHandler` in production (log aggregators like Datadog, Loki, and Cloud Logging parse it automatically). Switch by reading an environment variable at startup.

Log Levels

`slog` has four built-in levels in increasing severity:

Level	Use for
`Debug`	Detailed internal state; off in production
`Info`	Normal events: requests handled, tasks completed
`Warn`	Recoverable anomalies: retried operations, degraded mode
`Error`	Failures that need attention but did not crash the process

```
logger.Debug("cache miss", slog.String("key", key))
logger.Info("track loaded", slog.String("title", "Café Del Mar"))
logger.Warn("rate limit approaching", slog.Int("remaining", 5))
logger.Error("database query failed", slog.Any("err", err))
```

Trap: Do not log and then return an error for the same event. Either log it (at the point where you handle it) or return it (so the caller can decide whether to log), never both. Doubling up fills logs with duplicate noise and makes it hard to tell where a problem actually originated.

Configuring the Log Level

`slog.HandlerOptions` controls the minimum level and whether source file locations are included:

```
opts := &slog.HandlerOptions{
    Level:     slog.LevelDebug, // log everything
    AddSource: true,            // include file:line in every record
}
logger := slog.New(slog.NewJSONHandler(os.Stderr, opts))
```

For a level you can change at runtime without restarting, use `slog.LevelVar`:

```
var logLevel slog.LevelVar // defaults to Info

func main() {
    if os.Getenv("DEBUG") != "" {
        logLevel.Set(slog.LevelDebug)
    }
    logger := slog.New(slog.NewJSONHandler(os.Stderr, &slog.HandlerOptions{
        Level: &logLevel,
    }))
    slog.SetDefault(logger)
}
```

Setting the Default Logger

`slog.SetDefault(logger)` installs your configured logger as the process-wide default. After this call, the package-level functions `slog.Info(...)`, `slog.Error(...)`, etc. all use your handler. This lets library code call `slog.Info(...)` without knowing which handler the application chose.

```
func main() {
    logger := slog.New(slog.NewJSONHandler(os.Stderr, nil))
    slog.SetDefault(logger) // all subsequent slog.* calls use this logger
    // ...
}
```

Tip: Call `slog.SetDefault` once, early in `main`, before starting any goroutines. Libraries should never call `slog.SetDefault` — that is the application's job.

Typed Attributes

Always use the typed constructors rather than bare strings:

```
func String(key, value string) Attr             // string attribute
func Int(key string, v int) Attr                // int attribute
func Float64(key string, v float64) Attr        // float attribute
func Bool(key string, v bool) Attr              // bool attribute
func Duration(key string, v time.Duration) Attr // duration attribute
func Any(key string, value any) Attr            // escape hatch for custom types
```

```
logger.Info("track loaded",
    slog.String("title", "Café Del Mar"),    // Energy 52
    slog.Int("bpm", 130),
    slog.Duration("elapsed", time.Since(start)),
)
```

Tip: Keep attribute key names lowercase with underscores (`request_id`, `user_email`, `elapsed_ms`). Consistent naming means you can write the same query in your log aggregator regardless of which service emitted the record. Treat your log schema like an API — it has consumers.

`logger.With` — Per-Request Context

`logger.With(attrs...)` returns a **child logger** that includes the given attributes on every record it emits. Use it to stamp every log line in a request handler with the request ID and user, so you can filter all lines for one request in one query:

```
func handleUpload(w http.ResponseWriter, r *http.Request) {
    log := slog.Default().With(
        slog.String("request_id", r.Header.Get("X-Request-ID")),
        slog.String("user", userFromContext(r.Context())),
    )
    log.Info("upload started")
    // ... do work ...
    log.Info("upload complete", slog.Int64("bytes", n))
    // both lines carry request_id and user automatically
}
```

This is the Go equivalent of SLF4J's MDC (Mapped Diagnostic Context), but without thread-local storage — the context travels with the logger value, not with the goroutine.

`slog.Group` — Nested Attributes

`slog.Group` nests attributes under a named key. JSON handlers render it as a nested object; text handlers join the names with a dot:

```
logger.Info("upload complete",
    slog.Group("file",
        slog.String("name", "tracks.csv"),
        slog.Int64("bytes", 204800),
    ),
)
// JSON: {"msg":"upload complete","file":{"name":"tracks.csv","bytes":204800}}
// text: msg="upload complete" file.name=tracks.csv file.bytes=204800
```

Context-Aware Logging

Every log method has a `*Context` variant that accepts a `context.Context` as its first argument:

```
logger.InfoContext(ctx, "query executed", slog.Duration("elapsed", elapsed))
```

Custom handlers can extract trace IDs or span IDs from the context and include them automatically in every record — essential for correlating logs with distributed traces. Even if you do not implement this today, using `*Context` methods now means you can add trace correlation later without changing every log call site.

`regexp` — Regular Expressions

Go's `regexp` package uses RE2 syntax, which is a strict subset of the PCRE patterns you may know from Java's `java.util.regex`. RE2 guarantees linear-time matching — no catastrophic backtracking — by prohibiting backreferences and look-arounds.

RE2 Syntax Quick Reference

Pattern	Matches	Pattern	Matches
`.`	Any char except newline	`a*`	Zero or more (greedy)
`^`	Start of text (`(?m)` for line)	`a+`	One or more (greedy)
`$`	End of text (`(?m)` for line)	`a?`	Zero or one
`\d`	Digit `[0-9]`	`a{n}`	Exactly `n`
`\D`	Non-digit	`a{n,m}`	Between `n` and `m`
`\w`	Word char `[0-9A-Za-z_]`	`a*?`	Zero or more (non-greedy)
`\W`	Non-word character	`a+?`	One or more (non-greedy)
`\s`	Whitespace	`(abc)`	Capturing group
`\S`	Non-whitespace	`(?:abc)`	Non-capturing group
`[abc]`	Any of `a`, `b`, `c`	`a|b`	Alternation
`[^abc]`	Any except `a`, `b`, `c`	`\b`	Word boundary
`[a-z]`	Character range		

RE2 does **not** support backreferences (`\1`), lookaheads (`(?=...)`), or lookbehinds (`(?<=...)`).

Compile Once, Use Many Times

```
func Compile(expr string) (*Regexp, error)   // compile; returns error on bad syntax
func MustCompile(expr string) *Regexp        // like Compile but panics; for package vars
```

The critical pattern: compile the expression once (at package level or in an `init` function) and reuse the `*Regexp` value for every match. Compiling on every call is expensive — the equivalent of calling `Pattern.compile(regex)` inside a loop in Java.

```
// At package level: compiled once when the program starts.
var titleRE = regexp.MustCompile(`^[A-Z][a-z]+ \w+$`)

func isValidTitle(s string) bool {
    return titleRE.MatchString(s)
}
```

Trap: Never call `regexp.MustCompile` (or `regexp.Compile`) inside a function that is called repeatedly. Each call re-parses and re-compiles the pattern — orders of magnitude slower than reusing a compiled `*Regexp`.

Common *Regexp Methods

```
func (re *Regexp) MatchString(s string) bool                // reports whether s matches
func (re *Regexp) FindString(s string) string               // returns leftmost match, or ""
func (re *Regexp) FindAllString(s string, n int) []string   // all matches; n=-1 means all
func (re *Regexp) FindStringSubmatch(s string) []string     // match + capture groups
func (re *Regexp) ReplaceAllString(src, repl string) string // replace all matches
// ReplaceAllStringFunc replaces each match with f(match):
func (re *Regexp) ReplaceAllStringFunc(src string, f func(string) string) string
```

```
var isbnRE = regexp.MustCompile(`\d{3}-\d{10}`)

fmt.Println(isbnRE.MatchString("978-0135182789"))  // true
fmt.Println(isbnRE.FindString("isbn: 978-0135182789 end"))  // 978-0135182789
```

```go
var versionRE = regexp.MustCompile(`(\d+)\.(\d+)\.(\d+)`)
m := versionRE.FindStringSubmatch("version 1.21.0 released")
fmt.Println(m[0], m[1], m[2], m[3])  // 1.21.0  1  21  0
```

`FindStringSubmatch` returns a slice where `m[0]` is the whole match and `m[1]`, `m[2]`, … are the capture groups — the same model as Java's `Matcher.group(0)`, `group(1)`, etc.

Tip: Go's RE2 does not support backreferences (`\1`) or lookaheads (`(?=...)`). If you are porting a Java pattern that uses these, you will need to restructure the logic, typically by splitting the match into multiple passes or using `FindStringSubmatch` with post-processing.

`cmp` and `maps` — Comparison and Map Utilities

Go 1.21 added the `cmp` and `maps` packages with generic utilities that complement the collections and sorting functions already covered.

`cmp` — Three-Way Comparison

It provides generic comparison utilities that are useful when sorting slices of structs.

```go
// package cmp
func Compare[T Ordered](x, y T) int // -1 if x < y, 0 if x == y, +1 if x > y
```

`cmp.Ordered` is a type constraint satisfied by all integer types, floating-point types, and `string`.

The practical use case is sorting a slice of structs by a numeric or string field:

```go
import (
    "cmp"
    "fmt"
    "slices"
)

type Track struct {
    Title  string
    BPM    int
}

func main() {
    playlist := []Track{
        {Title: "Zombie",               BPM: 152},
        {Title: "Café Del Mar",         BPM: 126},
        {Title: "The Sound of Silence", BPM:  94},
        {Title: "Crazy Train",          BPM: 138},
    }

    slices.SortFunc(playlist, func(a, b Track) int {
        return cmp.Compare(a.BPM, b.BPM)
    })

    for _, t := range playlist {
        fmt.Printf("%s: %d\n", t.Title, t.BPM)
    }
}
// The Sound of Silence: 94
```

```
// Café Del Mar: 126
// Crazy Train: 138
// Zombie: 152
```

`cmp.Compare` replaces the classic three-way comparison pattern and is safe for floats (it handles NaN consistently).

`maps` — Map Utilities

Go 1.21 added the `maps` package to the standard library with utility functions for working with maps. Go 1.23 added `Keys` and `Values`, which return `iter.Seq` iterators rather than slices (the older slice-returning versions only ever lived in the experimental `golang.org/x/exp/maps`); `Clone` has been in the standard package since Go 1.21.

```
import "maps"

func Keys[Map ~map[K]V, K comparable, V any](m Map) iter.Seq[K]   // yields each key (1.23)
func Values[Map ~map[K]V, K comparable, V any](m Map) iter.Seq[V] // yields each value (1.23)
func Clone[Map ~map[K]V, K comparable, V any](m Map) Map          // shallow copy of m (1.21)
```

The most common use is combining `maps.Keys` with `slices.Sorted` to iterate a map in sorted key order without manually building a key slice:

```
import (
    "fmt"
    "maps"
    "slices"
)

func main() {
    bpm := map[string]int{
        "amapiano": 112,
        "hyperpop": 160,
        "lo-fi":    85,
    }

    for _, genre := range slices.Sorted(maps.Keys(bpm)) {
        fmt.Printf("%s: %d\n", genre, bpm[genre])
    }
}
// amapiano: 112
// hyperpop: 160
// lo-fi: 85
```

`maps.Clone` makes a shallow copy of a map — useful when you want to mutate a map without affecting the original.

`iter` — Iterators

Before Go 1.23, lazily processing a sequence without collecting it into a slice first required either a callback, a goroutine-backed channel, or a bespoke struct with `Next()/Value()` methods. None of those compose well: you can't feed one into another without glue code. Go 1.23 introduced the `iter` package and range-over-function support, giving the language a standard iterator contract.

```
type Seq[V any]     func(yield func(V) bool)            // iterator over single values
type Seq2[K, V any] func(yield func(K, V) bool)         // iterator over key-value pairs
```

```
func Pull[V any](seq Seq[V]) (next func() (V, bool), stop func())            // push -> pull
func Pull2[K, V any](seq Seq2[K, V]) (next func() (K, V, bool), stop func()) // same for Seq2
```

An `iter.Seq[V]` is just a function that calls `yield` once per item. If `yield` returns `false` the iterator must stop — that is how `break` and `return` inside a `for range` loop are communicated back to the producer.

Iterating over Strings

Go 1.24 added `strings.Lines` alongside several other lazy string iterators:

```
func strings.Lines(s string) iter.Seq[string]         // newline-delimited lines (1.24)
func strings.SplitSeq(s, sep string) iter.Seq[string] // lazy strings.Split (1.24)
func strings.FieldsSeq(s string) iter.Seq[string]     // lazy strings.Fields (1.24)
```

```
const lyrics = "Flowers\nWater\nSun and moon\nAin't it something\n"

for line := range strings.Lines(lyrics) {
    fmt.Print(line)
}
// Flowers
// Water
// Sun and moon
// Ain't it something
```

`strings.Lines` keeps the trailing newline on each element; strip it with `strings.TrimRight` if you need a clean token.

`strings.SplitSeq` is handy when the delimiter is not a newline:

```
csv := "Beyoncé,SZA,Doja Cat,Cardi B"

for artist := range strings.SplitSeq(csv, ",") {
    fmt.Println(artist)
}
```

`iter.Pull` — Consuming Values One at a Time

`for range` is all-or-nothing: it drives the whole sequence. `iter.Pull` converts a push-style `Seq` into a pull-style `(next, stop)` pair that you call manually, so you can consume any number of values — including just one — before deciding what to do next.

```
const lyrics = "Flowers\nWater\nSun and moon\nAin't it something\n"

next, stop := iter.Pull(strings.Lines(lyrics))
defer stop()

first, ok := next()         // pull the first line without a loop
if ok {
    fmt.Printf("opening line: %s", first)
}

for {                       // then drain the rest
    line, ok := next()
    if !ok {
        break
    }
```

```
    fmt.Print(line)
}
// opening line: Flowers
// Water
// Sun and moon
// Ain't it something
```

Always call `stop()` (typically via `defer`) even if you exhaust the iterator, so the underlying producer can release resources. `iter.Pull` is also useful when you need to step two iterators in lockstep, consuming one value from each per iteration.

Single-use Iterators

Some iterators are single-use, meaning you can iterate over them only once. A reusable iterator restarts from the beginning each time you `range` over it; a single-use iterator picks up where it left off. `strings.Lines` returns a single-use iterator. We know this because the documentation for `strings.Lines` has the phrase *It returns a single-use iterator*.

A single-use iterator allows you to combine `iter.Pull` and `range` to do something pretty cool. Let's say we want to iterate over the lines, but we want to process the first line before the `for` loop.

```
const lyrics = "Flowers\nWater\nSun and moon\nAin't it something\n"

lines := strings.Lines(lyrics)
next, stop := iter.Pull(lines)
defer stop()

first, ok := next()         // pull the first line without a loop
if ok {
    fmt.Printf("opening line: %s", first)
}

for line := range lines {                       // then drain the rest
    fmt.Print(line)
}
// opening line: Flowers
// Water
// Sun and moon
// Ain't it something
```

`encoding/base64` — Base64 Encoding

`encoding/base64` encodes and decodes binary data as printable ASCII — essential whenever you need to store or transmit raw bytes in a text context (HTTP headers, JSON fields, URLs).

```
func NewEncoder(enc *Encoding, w io.Writer) io.WriteCloser // streaming encoder
func NewDecoder(enc *Encoding, r io.Reader) io.Reader      // streaming decoder
func (enc *Encoding) EncodeToString(src []byte) string     // encode bytes to string
func (enc *Encoding) DecodeString(s string) ([]byte, error) // decode string to bytes
```

For the allocation-free path you write into a `[]byte` you own:

```
func (enc *Encoding) EncodedLen(n int) int                       // dst size to encode n bytes
func (enc *Encoding) DecodedLen(n int) int                       // max size to decode n bytes
func (enc *Encoding) Encode(dst, src []byte)                     // encode into pre-sized dst
func (enc *Encoding) Decode(dst, src []byte) (n int, err error)  // decode into pre-sized dst
```

```
func (enc *Encoding) AppendEncode(dst, src []byte) []byte          // append encoded src (1.22)
func (enc *Encoding) AppendDecode(dst, src []byte) ([]byte, error) // append decoded src (1.22)
```

The package provides four standard encodings:

Encoding	Use case
`base64.StdEncoding`	MIME (e-mail, PEM) — uses `+` and `/`, padded with `=`
`base64.URLEncoding`	URL and filename-safe — uses `-` and `_`, padded
`base64.RawStdEncoding`	Like StdEncoding but no padding
`base64.RawURLEncoding`	URL-safe, no padding — the most common choice for tokens

```
data := []byte("Bad Apple!!")
encoded := base64.StdEncoding.EncodeToString(data)
fmt.Println(encoded) // QmFkIEFwcGxlISE=

decoded, err := base64.StdEncoding.DecodeString(encoded)
fmt.Println(string(decoded)) // Bad Apple!!
```

`EncodeToString` and `DecodeString` allocate a fresh buffer on every call. When you want to avoid that — a hot loop, or a scratch slice you reuse — the lower-level `Encode` and `Decode` write straight into a `[]byte` you supply. The catch is that **you** are responsible for sizing that buffer first: `Encode` needs `EncodedLen(len(src))` bytes and `Decode` needs `DecodedLen(len(src))` bytes. Hand `Encode` a destination that is too short and it panics.

```
src := []byte("buenos días")
dst := make([]byte, base64.StdEncoding.EncodedLen(len(src)))
base64.StdEncoding.Encode(dst, src) // writes into the pre-sized dst
fmt.Println(string(dst))            // YnVlbm9zIGTDrWFz
```

Since Go 1.22, `AppendEncode` and `AppendDecode` take the sizing off your hands. They append the result to the destination slice and grow it if needed — exactly like the built-in `append` — so there is no `EncodedLen`/`DecodedLen` bookkeeping to get wrong:

```
buf := []byte("id=")
buf = base64.StdEncoding.AppendEncode(buf, []byte("buenos días"))
fmt.Println(string(buf)) // id=YnVlbm9zIGTDrWFz
```

Tip: Reach for `AppendEncode`/`AppendDecode` when you are building up a buffer; reach for `Encode`/`Decode` only when you already own a correctly sized destination and want zero allocations.

Trap: `StdEncoding` and `URLEncoding` produce different output for the same input. Always use the same encoding for encoding and decoding — a `+` in `StdEncoding` output becomes `%2B` in a URL, which a `URLEncoding` decoder will reject.

`crypto/rand` and `math/rand/v2` — Random Numbers

Go has two random number packages for two very different purposes.

`crypto/rand` produces **cryptographically secure** random bytes suitable for secrets, tokens, and keys. It reads from the OS entropy source (the `getrandom(2)` system call on Linux, `ProcessPrng` on Windows).

```
func Read(b []byte) (n int, err error) // fill b with random bytes; never errors in practice
```

```go
import "crypto/rand"
import "encoding/base64"

func generateToken(n int) (string, error) {
    b := make([]byte, n)
    if _, err := rand.Read(b); err != nil {
        return "", err
    }
    return base64.RawURLEncoding.EncodeToString(b), nil
}
```

`math/rand/v2` (Go 1.22+) is the **fast, non-cryptographic** package for simulations, shuffles, random sampling, and tests. It is not suitable for security-sensitive code.

```go
func N[I Integer](n I) I                // uniform integer in [0, n) for any integer I
func Float64() float64                  // uniform float in [0.0, 1.0)
func Shuffle(n int, swap func(i, j int)) // shuffle a slice
```

```go
import "math/rand/v2"

fmt.Println(rand.N(100))             // random int in [0, 100)
rand.Shuffle(len(tracks), func(i, j int) {
    tracks[i], tracks[j] = tracks[j], tracks[i]
})
```

Seeding and Reproducibility

The package-level helpers (`rand.N`, `rand.Float64`, `rand.Shuffle`) all draw from a single global generator that Go seeds automatically with a random value at startup. That means every run produces a different sequence — exactly what you want in production, but a problem when you need a test or a simulation to behave the same way twice.

A pseudo-random generator is deterministic: given the same starting **seed**, it emits the same stream of numbers forever. Coming from Java, this is the same idea as `new Random(42L)` versus the no-arg `new Random()` — a fixed seed gives a repeatable sequence, an unseeded one does not. When you want that repeatability, build your own generator from an explicit source instead of using the global:

```go
func New(src Source) *Rand            // wrap a source in a full-featured generator
func NewPCG(seed1, seed2 uint64) *PCG // a seedable PCG source (implements Source)

r := rand.New(rand.NewPCG(1, 2))                   // two uint64 seeds
fmt.Println(r.IntN(100), r.IntN(100), r.IntN(100)) // 76 61 78 --- every run, every machine
```

A `*rand.Rand` exposes the same methods as the package-level functions (`IntN`, `Float64`, `Shuffle`, and the rest); only the source of randomness differs. Feed it the same seeds and you get the same numbers; change a seed and you get a fresh but equally reproducible stream. `NewPCG` is the small, fast source; `rand.NewChaCha8(seed [32]byte) *ChaCha8` is the other built-in source, useful when you want a higher-quality stream from a 32-byte seed.

Wut: `math/rand/v2` removed the top-level `Seed` function. The original `math/rand` lets you pin its global generator with `rand.Seed` (deprecated since Go 1.20, which switched that global to auto-seed); `math/rand/v2` drops `Seed` entirely, so its global is always randomly seeded and **cannot** be pinned. If you need determinism, you must make your own `rand.New(rand.NewPCG(...))` — the global is off-limits.

Tip: `crypto/rand` has no seed at all. It is wired directly to the OS entropy source, so its output can never be reproduced — which is the whole point for secrets. Seeding belongs to `math/rand`, never `crypto/rand`.

Trap: Never use `math/rand` (or `math/rand/v2`) for passwords, tokens, session IDs, or encryption keys. Use `crypto/rand` for anything security-sensitive. The fast package is predictable given the seed — a bad actor who knows the seed can predict every value you generate.

`crypto/sha256`, `crypto/aes`, `crypto/cipher` — Hashing and Encryption

Go's `crypto` subtree provides production-grade primitives. Java programmers familiar with `javax.crypto` and `java.security.MessageDigest` will find the same patterns here.

Hashing with `crypto/sha256`

SHA-256 produces a 32-byte digest — useful for checksums, password hashing (with a proper KDF on top), and HMAC signatures.

```
import "crypto/sha256"

func Sum256(data []byte) [32]byte           // one-shot hash
func New() hash.Hash                        // streaming hash (implements io.Writer)
```

```
digest := sha256.Sum256([]byte("Bad Apple!!"))
fmt.Printf("%x\n", digest) // hex-encoded 64-character string
```

Trap: SHA-256 alone is not suitable for storing passwords. Use `golang.org/x/crypto/bcrypt` or `golang.org/x/crypto/argon2` which are designed to be intentionally slow and include a salt. SHA-256 is fast by design, which makes it easy to brute-force.

Symmetric Encryption with `crypto/aes` and `crypto/cipher`

AES-256-GCM is the standard choice for symmetric encryption in Go: authenticated, fast, and supported by hardware acceleration on most CPUs.

```
import (
    "crypto/aes"
    "crypto/cipher"
    "crypto/rand"
)

func encrypt(key, plaintext []byte) ([]byte, error) {
    block, err := aes.NewCipher(key)      // key must be 16, 24, or 32 bytes
    if err != nil {
        return nil, err
    }
    gcm, err := cipher.NewGCM(block)      // GCM mode: authenticated encryption
    if err != nil {
        return nil, err
    }
    nonce := make([]byte, gcm.NonceSize()) // 12 bytes; must be unique per message
    if _, err := rand.Read(nonce); err != nil {
        return nil, err
```

```
    }
    return gcm.Seal(nonce, nonce, plaintext, nil), nil // nonce prepended to ciphertext
}

func decrypt(key, ciphertext []byte) ([]byte, error) {
    block, _ := aes.NewCipher(key)
    gcm, _ := cipher.NewGCM(block)
    nonce, ct := ciphertext[:gcm.NonceSize()], ciphertext[gcm.NonceSize():]
    return gcm.Open(nil, nonce, ct, nil)  // authenticates and decrypts
}
```

The `gcm.Seal` / `gcm.Open` pair handles authentication automatically — if the ciphertext is tampered with, `Open` returns an error. This is called **authenticated encryption with associated data** (**AEAD**).

Tip: A 32-byte key gives you AES-256. Generate keys with `crypto/rand` and store them in environment variables or a secrets manager — never hard-code them. The nonce must never be reused with the same key; a fresh random nonce per message guarantees this.

Try It

The fastest way to make these packages stick is to wire several of them together in one small program. Type the following into a file and run it with `go run .` — it reads track titles from an in-memory reader with `bufio.Scanner`, sorts them with `slices.SortFunc` plus `cmp.Compare`, prints a numbered list, and logs a structured summary with `slog`.

```
package main

import (
    "bufio"
    "cmp"
    "fmt"
    "log/slog"
    "os"
    "slices"
    "strings"
    "time"
)

func main() {
    start := time.Now()
    logger := slog.New(slog.NewTextHandler(os.Stderr, nil))

    input := "Crazy Train\nCafé Del Mar\n\nZombie\nThe Sound of Silence\n"
    scanner := bufio.NewScanner(strings.NewReader(input))
    var titles []string
    for scanner.Scan() {
        if line := strings.TrimSpace(scanner.Text()); line != "" {
            titles = append(titles, line)
        }
    }
    if err := scanner.Err(); err != nil {
        logger.Error("scan failed", slog.Any("err", err))
        os.Exit(1)
    }
```

```
    slices.SortFunc(titles, func(a, b string) int {
        return cmp.Compare(a, b)
    })
    for i, t := range titles {
        fmt.Printf("%2d. %s\n", i+1, t)  // numbered, sorted playlist on stdout
    }

    logger.Info("scan complete",
        slog.Int("tracks", len(titles)),
        slog.Duration("elapsed", time.Since(start)),
    )
}
```

The numbered list goes to stdout in sorted order; the `slog` summary goes to stderr (with a timestamp and elapsed duration that vary per run).

Try these modifications:

- Switch `slog.NewTextHandler` to `slog.NewJSONHandler` and compare the output format.
- Sort by title length instead of alphabetically by returning `cmp.Compare(len(a), len(b))`.
- Replace the `strings.NewReader` source with `os.Stdin` and pipe a file in with `go run . < playlist.txt`.

Key Points

- `%+v` adds field names to struct output; `%#v` gives Go-syntax output you can paste back into code.
- `fmt.Fprintf` writes to any `io.Writer` — a file, a buffer, a network connection, a test recorder.
- `io.Copy` streams without buffering the entire input; `io.ReadAll` loads everything into memory.
- `io.TeeReader`, `io.MultiWriter`, `io.LimitReader`, and `io.Pipe` compose readers and writers without intermediate allocations.
- `bufio.Scanner` is the idiomatic line-by-line reader; always check `scanner.Err()` after the loop.
- Always `defer bw.Flush()` after creating a `bufio.Writer` or buffered data will be silently lost.
- Always `defer f.Close()` after opening a file — Go has no try-with-resources.
- `exec.Command` takes separate arguments, not a shell string; use `exec.Command("ls", "-la")`, not `exec.Command("ls -la")`.
- `flag.Parse()` must be called before reading any flag values; call it from `main`, not `init`.
- `time.Duration` is an `int64` nanoseconds; write durations as `30 * time.Second`, not `time.Duration(30)`.
- Go's `time.Format` uses reference-time constants (`2006-01-02`), not pattern tokens like Java's `yyyy-MM-dd`.
- Always `defer ticker.Stop()` to avoid goroutine leaks.
- `log/slog` (Go 1.21) provides structured logging; use `JSONHandler` in production, `TextHandler` in development.
- Call `slog.SetDefault` once in `main`; use `slog.LevelVar` for a runtime-adjustable level.
- Use `logger.With` to stamp per-request attributes on every log line without repeating them.
- Use `*Context` log methods so custom handlers can inject trace IDs automatically.
- Don't log and return the same error — pick one.
- Compile `regexp.MustCompile` once at package level; never inside a hot function.
- `base64.RawURLEncoding` is the most common choice for URL-safe tokens; always encode and decode with the same variant.
- Use `crypto/rand` for secrets and tokens; use `math/rand/v2` for simulations and shuffles — never the reverse.
- For a reproducible sequence, build your own `rand.New(rand.NewPCG(seed1, seed2))`; the auto-seeded global generator cannot be pinned, and `crypto/rand` has no seed at all.
- `crypto/sha256` produces a 32-byte digest; use `golang.org/x/crypto/bcrypt` for passwords.

- AES-256-GCM (`crypto/aes` + `cipher.NewGCM`) provides authenticated encryption; use a fresh random nonce per message.

Exercises

1. **Think about it:** In Java, `InputStream`, `OutputStream`, `Reader`, and `Writer` are four separate abstract class hierarchies. Go has two interfaces — `io.Reader` and `io.Writer` — and a set of composition functions. What design decision makes Go's two-interface model work where Java needed four base classes? What would be harder to express cleanly in Go's model?

2. **What does this print?**

```
package main

import (
    "bufio"
    "log/slog"
    "os"
    "strings"
    "time"
)

func main() {
    logger := slog.New(slog.NewTextHandler(os.Stdout, &slog.HandlerOptions{
        ReplaceAttr: func(groups []string, a slog.Attr) slog.Attr {
            if a.Key == slog.TimeKey {
                return slog.Attr{}  // suppress the timestamp
            }
            return a
        },
    }))

    input := "Café Del Mar\nZombie\nCrazy Train\n"
    scanner := bufio.NewScanner(strings.NewReader(input))
    count := 0
    for scanner.Scan() {
        count++
    }

    logger.Info("scan complete",
        slog.Int("lines", count),
        slog.Duration("elapsed", 0*time.Millisecond),
    )
}
```

3. **Calculation:** You open a 10 MiB file and read it in three ways:

   (a) `os.ReadFile` into a `[]byte`,
   (b) `bufio.NewScanner` reading line by line,
   (c) `io.Copy(io.Discard, f)` using the default 32 KiB copy buffer. For each approach, estimate the peak heap allocation in MiB, assuming the file contains 100,000 lines of 100 bytes each. Which approach is best for counting lines without storing the content?

4. **Where is the bug?**

```
package main
```

```
import (
    "fmt"
    "regexp"
)

func countMatches(texts []string, pattern string) int {
    total := 0
    for _, t := range texts {
        re := regexp.MustCompile(pattern)
        if re.MatchString(t) {
            total++
        }
    }
    return total
}

func main() {
    titles := []string{"Crazy Train", "Café Del Mar", "Zombie", "The Sound of Silence"}
    fmt.Println(countMatches(titles, `^[A-Z]`))
}
```

5. **Write a program:** Write a CLI tool that accepts three flags: `-dir` (a directory path, default `"."`), `-ext` (a file extension like `".go"`, default `".go"`), and `-verbose` (a boolean, default `false`). The tool should walk the directory tree, count files whose name ends with the given extension, and print the total. When `-verbose` is set, log each matching file path using `slog` at the `Info` level with a `"file"` attribute. Use `log/slog` with a text handler writing to `os.Stderr`, `path/filepath.WalkDir`, and `flag`.

Chapter 15

JSON, HTTP, and the Web

Go's standard library ships everything you need to build and consume web services — no Spring Boot, no Jackson, no Tomcat required. In the Java world, a production HTTP service usually drags in a web framework, a servlet container, and a JSON binding library, each with its own configuration files and version matrix. In Go, `encoding/json` and `net/http` cover the same ground from the standard library: a routed HTTP server with method matching, middleware, TLS, and JSON binding fits in a single file with no external dependencies. That keeps your `go.mod` short, your builds fast, and your deployment a single static binary. This chapter covers JSON encoding and decoding, making HTTP requests, building HTTP servers with the 1.22 mux, writing middleware, live profiling endpoints with `net/http/pprof`, XML encoding, raw TCP/UDP networking with the `net` package, and securing connections with `crypto/tls`.

Encoding JSON

Java programmers typically reach for Jackson or Gson to serialize objects to JSON. In Go, `encoding/json` is in the standard library and needs no configuration beyond struct tags.

The two core functions are:

```
func Marshal(v any) ([]byte, error)       // encode v as JSON bytes
func Unmarshal(data []byte, v any) error  // decode JSON bytes into v (non-nil pointer)
```

Here is a round trip:

```
package main

import (
    "encoding/json"
    "fmt"
)

type Song struct {
    Title  string `json:"title"`
    Artist string `json:"artist"`
    BPM    int    `json:"bpm"`
}

func main() {
    s := Song{Title: "Sandstorm", Artist: "Darude", BPM: 136}

    data, err := json.Marshal(s)
```

```
    if err != nil {
        panic(err)
    }
    fmt.Println(string(data))
    // {"title":"Sandstorm","artist":"Darude","bpm":136}

    var s2 Song
    if err := json.Unmarshal(data, &s2); err != nil {
        panic(err)
    }
    fmt.Printf("%+v\n", s2)
    // {Title:Sandstorm Artist:Darude BPM:136}
}
```

`Marshal` returns a `[]byte`. Convert it to `string` with `string(data)` when you need to print it or embed it in a larger string. `Unmarshal` takes a pointer so it can write into the struct; passing a non-pointer returns an error.

Tip: In Jackson you annotate fields with `@JsonProperty("title")`. In Go you use a struct tag: `` `json:"title"` ``. Both achieve the same mapping — Go's approach requires no annotation processor or reflection framework beyond what `encoding/json` already does at runtime.

Struct Tags for JSON

A struct tag is a raw string literal attached to a field declaration. `encoding/json` reads the `json` key in each tag to control marshaling.

```
type Track struct {
    Title    string  `json:"title"`                // rename to "title"
    Artist   string  `json:"artist"`               // rename to "artist"
    BPM      int     `json:"bpm,omitempty"`        // omit if BPM == 0
    Internal string  `json:"-"`                    // always skip
}
```

The options after the comma are:

- `omitempty` — skip the field when its value is empty: `false`, `0`, a `nil` pointer or interface, or any empty array, slice, map, or string.
- `-` — never include this field in JSON output, even if it has a value.

```
t1 := Track{Title: "Out Of The Blue", Artist: "System F", BPM: 138, Internal: "secret"}
data, _ := json.Marshal(t1)
fmt.Println(string(data))
// {"title":"Out Of The Blue","artist":"System F","bpm":138}
// Internal is gone; BPM present because it is non-zero

t2 := Track{Title: "Flaming June", Artist: "BT"}
data, _ = json.Marshal(t2)
fmt.Println(string(data))
// {"title":"Flaming June","artist":"BT"}
// BPM omitted because it is zero (omitempty)
```

Trap: `omitempty` has no effect on a struct-typed field — a zero-valued struct is never omitted. If you want a nested struct to be omittable, make the field a pointer: `` `json:"meta,omitempty"` `` on a `*Meta` field will omit `meta` when the pointer is `nil`. Since Go 1.24 there is a cleaner fix: the `omitzero` option (`` `json:"meta,omitzero"` ``) omits any field whose value is the zero value for its type — including zero-valued structs — no pointer required. If the type has an `IsZero() bool` method, `omitzero` uses it, which is how a zero `time.Time` gets omitted.

Tag syntax is strict: backtick string, `key:"value"` pairs separated by spaces, no commas between pairs. `go vet ./...` catches malformed tags.

Streaming with `Encoder` and `Decoder`

`Marshal` and `Unmarshal` work on in-memory byte slices. When the JSON is coming from or going to an `io.Reader` or `io.Writer` — an HTTP body, a file, a network connection — use `json.NewEncoder` and `json.NewDecoder` instead.

```
func NewEncoder(w io.Writer) *Encoder  // write JSON to w
func NewDecoder(r io.Reader) *Decoder  // read JSON from r
```

Key methods on those types:

```
func (enc *Encoder) Encode(v any) error      // marshal v and write to w, followed by a newline
func (dec *Decoder) Decode(v any) error      // read one JSON value from r and unmarshal into v
func (dec *Decoder) More() bool              // true if another value remains in the stream
```

Encoding directly to an `http.ResponseWriter`:

```
func songHandler(w http.ResponseWriter, r *http.Request) {
    s := Song{Title: "Saltwater", Artist: "Chicane", BPM: 138}
    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(s)
}
```

Decoding a stream of newline-delimited JSON records:

```
dec := json.NewDecoder(r.Body)
for dec.More() {
    var s Song
    if err := dec.Decode(&s); err != nil {
        break
    }
    fmt.Println(s.Title)
}
```

By default the decoder silently ignores JSON fields that have no matching struct field. To reject them instead, call `DisallowUnknownFields` before decoding:

```
func (dec *Decoder) DisallowUnknownFields()  // make Decode fail on unknown JSON fields
```

```
dec := json.NewDecoder(r.Body)
dec.DisallowUnknownFields()  // any extra field makes Decode return an error
```

This is the analog of Jackson's `FAIL_ON_UNKNOWN_PROPERTIES` for Java readers.

Tip: Prefer streaming `Encoder`/`Decoder` over `Marshal`/`Unmarshal` when working with `io.Reader`/`io.Writer`. Streaming avoids allocating the entire JSON payload as a byte slice, which matters for large payloads.

Making HTTP Requests

The `net/http` package is also the HTTP client. The simplest way to make a GET request is `http.Get`:

```
func Get(url string) (resp *Response, err error)  // GET url; caller must close resp.Body
```

```
resp, err := http.Get("https://api.example.com/songs/1")
if err != nil {
    log.Fatal(err)
}
defer resp.Body.Close()  // MUST close the body

body, err := io.ReadAll(resp.Body)
if err != nil {
    log.Fatal(err)
}
fmt.Println(string(body))
```

Trap: You **must** close `resp.Body`, even if you do not read it. Failing to close the body leaks the underlying TCP connection and eventually exhausts the connection pool. `defer resp.Body.Close()` immediately after checking `err` is the standard idiom.

For anything beyond a simple GET — custom headers, timeouts, POST bodies — use `http.Client` and `http.NewRequest`:

```
// http.Client --- reusable, configurable; zero value uses sensible defaults
type Client struct {
    Transport     RoundTripper  // how to make a single HTTP transaction
    CheckRedirect func(req *Request, via []*Request) error  // redirect policy
    Jar           CookieJar     // cookie storage
    Timeout       time.Duration // zero means no timeout
}

func NewRequest(method, url string, body io.Reader) (*Request, error)  // build a request
func (c *Client) Do(req *Request) (*Response, error)                    // execute the request
```

A POST with a JSON body and a timeout:

```
client := &http.Client{Timeout: 10 * time.Second}

song := Song{Title: "Out Of The Blue", Artist: "System F", BPM: 138}
buf := new(bytes.Buffer)
json.NewEncoder(buf).Encode(song)

req, err := http.NewRequest("POST", "https://api.example.com/songs/", buf)
if err != nil {
    log.Fatal(err)
}
req.Header.Set("Content-Type", "application/json")

resp, err := client.Do(req)
if err != nil {
    log.Fatal(err)
}
defer resp.Body.Close()
```

```
fmt.Println(resp.StatusCode)  // e.g. 201
```

Tip: Create one `http.Client` and reuse it across requests. `http.Client` manages a connection pool (`Transport`) internally. Creating a new client per request bypasses pooling and creates a new pool each time, which is wasteful and slower. The package-level `http.Get` and `http.Post` helpers use a shared default client (`http.DefaultClient`).

In Java you would use `HttpClient` (Java 11+), `RestTemplate`, or OkHttp. Go's `http.Client` fills the same role.

Building an HTTP Server

Go has a built-in HTTP server that calls out to an `http.Handler` to service the requests. The simplest way to start up an HTTP server is `http.ListenAndServe`, which takes a `host:port` to listen on and an `http.Handler` to service the requests.

```
func ListenAndServe(addr string, handler Handler) error  // nil handler = DefaultServeMux
```

`http.ListenAndServe` is really just for quick demos and prototypes. It spins up a server with all the default settings and gives you no handle to control it once it is running — there is no way to set timeouts, no way to shut it down gracefully, no way to do anything but kill the process. For production setups you instantiate an `http.Server` yourself, which gives you a handle to control the server's lifecycle (configure timeouts, shut down gracefully, drain in-flight requests). We start with `http.ListenAndServe` because it keeps the early examples short, then switch to `http.Server` once we have something worth running for real.

The foundation of Go's HTTP server is the `http.Handler` interface:

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)  // handles one HTTP request
}
```

Here is a super basic HTTP server:

```
package main

import (
    "fmt"
    "net/http"
)

func serveHttpRequest(w http.ResponseWriter, r *http.Request) {
    _, _ = w.Write([]byte("Hello from " + r.URL.Path))
}
func main() {
    err := http.ListenAndServe(":8888", http.HandlerFunc(serveHttpRequest))
    if err != nil {
        fmt.Printf("Problem starting http server %v", err)
    }
}
```

Before we jump into the HTTP API, it is worthwhile to take a moment to highlight how useful adapter functions are. `http.HandlerFunc` is a function type that has the same signature as `serveHttpRequest`. It also has a method, `func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request)`, so `http.HandlerFunc` implements the `http.Handler` interface, which is why we can pass it to `http.ListenAndServe`.

For nontrivial HTTP apps, we need better server management and request routing. In this simple example, we look at the path of the request in the handler function. Ideally, we would like to route requests to different

handler functions based on the type of request and the request path. Go has us covered! A **mux** (short for *multiplexer*, also called a *router*) is itself a handler whose whole job is to look at the request's method and path and dispatch to the right handler. `http.ServeMux` is the standard library's implementation. Think of it as `web.xml`/`@RequestMapping` routing, but as plain code you write in `main`.

When we use a mux, the request flow is: the network connection lands in the server, the server hands the request to the mux, and the mux dispatches it to the handler registered for that route. Because every layer is just an `http.Handler`, you can wrap one in another — which is exactly how middleware works, as you'll see later in this chapter.

In the `ServeMux` example below we will also see the `http.Server` type, which lets us construct a server that can be configured, started, and stopped.

`ServeMux` — Method Routing and Wildcards

`http.ServeMux` is a builtin handler that routes requests to `http.Handler`s or plain handler functions based on the request method and path, and can even parse out path elements. Before Go 1.22, `http.ServeMux` matched only on path prefixes and required external routers (Chi, Gorilla Mux, etc.) for method or wildcard routing. Go 1.22 upgraded the built-in mux significantly.

Method routing — prefix the pattern with an HTTP method and a space:

```
mux := http.NewServeMux()
mux.HandleFunc("GET /songs/", listSongs)    // only GET requests
mux.HandleFunc("POST /songs/", createSong)  // only POST requests
```

Path wildcards — `{name}` captures a path segment, `{name...}` captures the rest:

```
mux.HandleFunc("GET /songs/{id}/", getSong)  // /songs/42/ --- id = "42"
```

Retrieve the captured value with `r.PathValue`:

```
func getSong(w http.ResponseWriter, r *http.Request) {
    id := r.PathValue("id")                        // "42"
    fmt.Fprintf(w, "fetching song %s\n", id)
}
```

A complete example wiring up a small songs API:

```
package main

import (
    "encoding/json"
    "log"
    "net/http"
    "time"
)

type Song struct {
    ID     string `json:"id"`
    Title  string `json:"title"`
    Artist string `json:"artist"`
}

var catalog = map[string]Song{
    "1": {ID: "1", Title: "Sandstorm", Artist: "Darude"},
    "2": {ID: "2", Title: "Saltwater", Artist: "Chicane"},
}
```

```go
func getSong(w http.ResponseWriter, r *http.Request) {
    id := r.PathValue("id")
    song, ok := catalog[id]
    if !ok {
        http.Error(w, "not found", http.StatusNotFound)
        return
    }
    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(song)
}

func listSongs(w http.ResponseWriter, r *http.Request) {
    songs := make([]Song, 0, len(catalog))
    for _, s := range catalog {
        songs = append(songs, s)
    }
    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(songs)
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /songs/",      listSongs)
    mux.HandleFunc("GET /songs/{id}/", getSong)

    srv := &http.Server{
        Addr:         ":8080",
        Handler:      mux,
        ReadTimeout:  5 * time.Second,   // max time to read the whole request
        WriteTimeout: 10 * time.Second,  // max time to write the response
        IdleTimeout:  60 * time.Second,  // max keep-alive idle time
    }
    log.Fatal(srv.ListenAndServe())     // srv is the handle to the running server
}
```

Instead of calling the package-level `http.ListenAndServe`, we build an `http.Server` value and call its `ListenAndServe` method. The `srv` variable is now a handle to the running server: it carries the timeouts we configured and, as you will see in a moment, lets us shut the server down on our own terms.

```go
func (srv *Server) ListenAndServe() error               // listen on srv.Addr and serve
func (srv *Server) Shutdown(ctx context.Context) error  // graceful stop, drains in-flight
func (srv *Server) Close() error                        // immediately close all connections
```

Trap: The package-level `http.ListenAndServe` has no timeouts at all. A slow or malicious client that opens a connection and never finishes its request ties up a goroutine indefinitely (a *Slowloris* attack). Always set `ReadTimeout` and `WriteTimeout` on a real `http.Server`.

To stop cleanly, run the server in a goroutine and call `Shutdown` when you catch a termination signal. `Shutdown` stops accepting new connections and waits for in-flight requests to finish, up to the context's deadline:

```go
func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /songs/",      listSongs)
    mux.HandleFunc("GET /songs/{id}/", getSong)
```

```
    srv := &http.Server{Addr: ":8080", Handler: mux}

    go func() {
        if err := srv.ListenAndServe(); err != nil && err != http.ErrServerClosed {
            log.Fatalf("server error: %v", err)
        }
    }()

    // wait for SIGINT or SIGTERM
    stop := make(chan os.Signal, 1)
    signal.Notify(stop, os.Interrupt, syscall.SIGTERM)
    <-stop

    // give in-flight requests up to 10 seconds to finish
    ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)
    defer cancel()
    if err := srv.Shutdown(ctx); err != nil {
        log.Printf("graceful shutdown failed: %v", err)
    }
}
```

`ListenAndServe` returns `http.ErrServerClosed` once `Shutdown` is called, which is why we treat that error as the normal exit rather than a failure. This is the Go equivalent of a servlet container draining requests before stopping — but it is a few lines of explicit code rather than container lifecycle hooks.

Notice that `getSong` calls `http.Error` and immediately returns when the song is not found, so the success path only runs when the lookup succeeds. [*error-first-return-early*]

When both a wildcard pattern (`GET /songs/{id}/`) and a subtree pattern (`GET /songs/`) could match a request, the more-specific pattern wins, so `/songs/42/` goes to `getSong` rather than `listSongs`.

Wut: A trailing slash in the pattern makes it a subtree match: `GET /songs/` matches `/songs/`, `/songs/anything`, and `/songs/42/`. Without the trailing slash, `GET /songs` matches only the exact path `/songs`. This is the same behavior as pre-1.22 `ServeMux`.

Trap: A `{id}/` (trailing-slash) pattern requires the trailing slash, so a request to `/songs/1` is redirected (a 307 to `/songs/1/`), which can surprise clients that do not follow redirects.

Compared to Java: this is the rough equivalent of Spring MVC's `@GetMapping("/songs/{id}")` or JAX-RS's `@GET @Path("/songs/{id}")`, but built into the standard library with no framework dependency.

Middleware Chaining

Middleware in Go is a function that takes an `http.Handler` and returns an `http.Handler`. It wraps the inner handler with cross-cutting logic — logging, authentication, metrics, CORS — without modifying the handler itself.

The pattern:

```
func logging(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        log.Printf("%s %s", r.Method, r.URL.Path)  // before
        next.ServeHTTP(w, r)                       // call the inner handler
    })
}
```

Stack middleware by composing the wrappers:

```
func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /songs/{id}/", getSong)

    handler := logging(mux)   // outermost middleware runs first
    http.ListenAndServe(":8080", handler)
}
```

Add more layers by nesting calls:

```
handler := logging(auth(mux))  // logging wraps auth wraps mux
```

Tip: The middleware pattern is the Go equivalent of Java's servlet filters or Spring's `HandlerInterceptor`. Because `http.Handler` is a simple interface with one method, any function that wraps it composes cleanly — no framework annotation or XML configuration required.

You can also write a `chain` helper to apply a list of middleware in order:

```
func chain(h http.Handler, middleware ...func(http.Handler) http.Handler) http.Handler {
    for i := len(middleware) - 1; i >= 0; i-- {
        h = middleware[i](h)       // apply in reverse so the first listed runs first
    }
    return h
}

handler := chain(mux, logging, auth, metrics)
```

Live Profiling with `net/http/pprof`

Go's runtime can expose profiling data — CPU usage, heap allocations, goroutine stacks, mutex contention — as HTTP endpoints. All it takes is a blank import.

```
import _ "net/http/pprof"  // register pprof handlers with DefaultServeMux
```

The side-effect import registers several routes under `/debug/pprof/` on `http.DefaultServeMux`. If your server already calls `http.ListenAndServe(addr, nil)` (which uses `DefaultServeMux`), those routes are live the moment the process starts — no code changes beyond the import.

If you are using a custom `*http.ServeMux`, register the handlers explicitly:

```
import (
    "net/http"
    "net/http/pprof"
)

mux := http.NewServeMux()
mux.HandleFunc("GET /songs/{id}/", getSong)

// register pprof endpoints on the same mux
mux.HandleFunc("GET /debug/pprof/",        pprof.Index)
mux.HandleFunc("GET /debug/pprof/cmdline", pprof.Cmdline)
mux.HandleFunc("GET /debug/pprof/profile", pprof.Profile)
mux.HandleFunc("GET /debug/pprof/symbol",  pprof.Symbol)
mux.HandleFunc("GET /debug/pprof/trace",   pprof.Trace)
```

```
http.ListenAndServe(":8080", mux)
```

The key endpoints under `/debug/pprof/`:

Endpoint	What it shows
`/`	index of available profiles
`heap`	live heap allocations
`goroutine`	stack trace of every live goroutine
`allocs`	all past allocations, including freed ones
`mutex`	stack traces of code that held contended mutexes
`block`	stack traces of goroutines blocked on synchronization primitives (channel ops, mutex contention, `sync.WaitGroup`, etc.)
`profile?seconds=N`	CPU profile for N seconds (default 30)

The `block` and `mutex` profiles are off by default — call `runtime.SetBlockProfileRate(1)` and `runtime.SetMutexProfileFraction(1)` (typically at startup, behind a debug flag) to enable them.

Once the server is running, collect and view a profile with `go tool pprof`:

```
# collect a 30-second CPU profile and open the interactive browser UI
go tool pprof -http=:6060 http://localhost:8080/debug/pprof/profile?seconds=30

# inspect the live heap
go tool pprof -http=:6060 http://localhost:8080/debug/pprof/heap
```

`go tool pprof -http` opens a flame graph and call-graph view in your browser. The flame graph makes it easy to see which functions consume the most CPU or allocate the most memory.

Trap: Never expose `/debug/pprof/` on a public-facing port. CPU profiling adds measurable overhead to the whole process while it runs, the endpoints accept unbounded concurrent requests, and the profiles leak internals — source paths, goroutine stacks, even heap contents — to anyone who can reach the port. The standard pattern is to run pprof on a separate, internal-only port:

```
// internal admin server on a non-public port
go func() {
    // nil handler means DefaultServeMux, which has the pprof routes
    log.Println(http.ListenAndServe("localhost:6060", nil))
}()
// public server with a custom mux
http.ListenAndServe(":8080", mux)
```

In a container environment, expose port 6060 only within the cluster network.

Tip: In Java, runtime profiling typically requires attaching an external tool (VisualVM, JProfiler, YourKit) via the JVM attach mechanism or JMX. `net/http/pprof` is the Go equivalent built into the process — no agent, no attach, no separate daemon. The profile data is available over plain HTTP, which means you can script it with `curl` or integrate it into any monitoring pipeline.

Encoding XML

`encoding/xml` works exactly like `encoding/json` — same `Marshal`/`Unmarshal` functions, same streaming `Encoder`/`Decoder`, same struct tag mechanism. The tag key is `xml` instead of `json`.

```
func Marshal(v any) ([]byte, error)              // encode v as XML
func Unmarshal(data []byte, v any) error         // decode XML into v
func NewEncoder(w io.Writer) *Encoder            // streaming XML encoder
func NewDecoder(r io.Reader) *Decoder            // streaming XML decoder
```

```
import "encoding/xml"

type Song struct {
    XMLName xml.Name `xml:"song"`           // set the element name
    Title   string   `xml:"title"`
    Artist  string   `xml:"artist"`
    BPM     int      `xml:"bpm,omitempty"`
}

s := Song{Title: "Flaming June", Artist: "BT", BPM: 138}
data, _ := xml.Marshal(s)
fmt.Println(string(data))
// <song><title>Flaming June</title><artist>BT</artist><bpm>138</bpm></song>
```

The `xml.Name` field with tag `` `xml:"song"` `` sets the root element name. Without it, `encoding/xml` uses the struct type name, which is often not what you want.

XML tag options beyond the field name:

- `xml:"name,attr"` — encode the field as an XML attribute rather than a child element.
- `xml:",chardata"` — encode the field as the character data content of the parent element.
- `xml:",omitempty"` — same as JSON: omit when zero value.
- `xml:"-"` — always skip.

Tip: If your service needs both JSON and XML representations of the same struct, define both tag keys on each field:

```
type Song struct {
    Title  string `json:"title"  xml:"title"`
    Artist string `json:"artist" xml:"artist"`
}
```

Both `encoding/json` and `encoding/xml` read only their own tag key and ignore the other.

Raw Networking with net

HTTP sits on top of TCP. The `net` package gives you direct access to TCP and UDP connections — useful for non-HTTP protocols, custom servers, or low-level networking tools.

The two fundamental functions are:

```
func Dial(network, address string) (Conn, error)   // connect to a server
func Listen(network, address string) (Listener, error) // listen for incoming connections
```

`network` is `"tcp"`, `"tcp4"`, `"tcp6"`, `"udp"`, `"udp4"`, or `"udp6"` for `Dial`; `Listen` accepts only the stream-oriented networks (`"tcp"` variants and `"unix"`). `address` is `"host:port"` for TCP/UDP, e.g. `"localhost:8080"` or `":8080"` (all interfaces).

`net.Conn` implements both `io.Reader` and `io.Writer`, so you can use any `io` utility on it:

```
// net.Conn is an interface; these are its method signatures
type Conn interface {
    Read(b []byte) (n int, err error)        // receive bytes
    Write(b []byte) (n int, err error)       // send bytes
```

```
    Close() error                          // close the connection
    LocalAddr() Addr                       // this side's network address
    RemoteAddr() Addr                      // the peer's network address
    SetDeadline(t time.Time) error         // set read+write deadline
    SetReadDeadline(t time.Time) error     // set read deadline only
    SetWriteDeadline(t time.Time) error    // set write deadline only
}
```

`LocalAddr` and `RemoteAddr` return a `net.Addr`, a small interface with `Network() string` (the protocol, e.g. `"tcp"`) and `String() string` (the `host:port` text); `conn.RemoteAddr().String()` is the usual way to log who connected.

A simple TCP echo client:

```
package main

import (
    "fmt"
    "net"
)

func main() {
    conn, err := net.Dial("tcp", "localhost:9000")
    if err != nil {
        panic(err)
    }
    defer conn.Close()

    fmt.Fprintf(conn, "hola\n")     // send

    buf := make([]byte, 1024)
    n, _ := conn.Read(buf)          // receive
    fmt.Printf("echo: %s", buf[:n])
}
```

Tip: Plain `net.Dial` has no connection-setup timeout — if the host is unreachable the call can block for the operating system's default, often a minute or more. The connection deadlines below (`SetReadDeadline` and friends) do not help here, because they only apply *after* the connection exists. To bound the dial itself, reach for `net.DialTimeout`:
`func DialTimeout(network, address string, timeout time.Duration) (Conn, error)`
`conn, err := net.DialTimeout("tcp", "localhost:9000", 5*time.Second)`
When you need `context`-based cancellation instead of a fixed duration, use a `net.Dialer` and its `DialContext` method: `(&net.Dialer{}).DialContext(ctx, "tcp", addr)`.

A minimal TCP echo server:

```
package main

import (
    "io"
    "net"
)

func main() {
    ln, err := net.Listen("tcp", ":9000")
    if err != nil {
```

```
        panic(err)
    }
    defer ln.Close()

    for {
        conn, err := ln.Accept()    // block until a client connects
        if err != nil {
            break
        }
        go io.Copy(conn, conn)      // echo: copy reads back to writes
    }
}
```

`ln.Accept()` blocks until a new connection arrives and returns a `net.Conn` for that connection. Each connection is handled in its own goroutine so the server can accept the next connection immediately.

Wut: `io.Copy(conn, conn)` works because `net.Conn` implements both `io.Reader` and `io.Writer`. `io.Copy(dst, src)` reads from `src` and writes to `dst` until EOF or an error. Here `conn` is both — every byte received is immediately written back.

Deadlines

`conn.Read` and `conn.Write` block indefinitely by default — a silent peer can hang a goroutine forever. In Java, you bound each read with `socket.setSoTimeout(5000)`. After that call, any read that blocks for more than 5000 milliseconds returns a timeout. Go takes a different approach: instead of a *duration* you set an absolute *deadline* with `SetDeadline`, `SetReadDeadline`, or `SetWriteDeadline`, each taking a `time.Time`. Once that deadline has passed, all reads on that connection will fail. So instead of setting a 5 second timeout, we say 5 seconds from now any reads on the connection will fail:

```
conn.SetReadDeadline(time.Now().Add(5 * time.Second))   // fail if no data within 5s
n, err := conn.Read(buf)
```

When the deadline passes, the blocked `Read` or `Write` returns immediately with an error that satisfies `net.Error` and reports `Timeout() == true` (it wraps `os.ErrDeadlineExceeded`):

```
n, err := conn.Read(buf)
if err != nil {
    var ne net.Error
    if errors.As(err, &ne) && ne.Timeout() {
        // deadline hit --- handle the timeout
    }
}
```

Wut: Deadlines are *absolute*, not rolling. `SetReadDeadline` sets one fixed instant, not a per-read timer like Java's `setSoTimeout`. For an idle timeout that resets on every successful read, you have to call `SetReadDeadline(time.Now().Add(d))` again before *each* read.

A deadline applies to all future I/O on the connection, not just the next call, until you change it. Pass the zero value, `time.Time{}`, to clear a deadline and block indefinitely again:

```
conn.SetDeadline(time.Time{})   // no deadline
```

Tip: Unlike most blocking operations in Go, `net.Conn` I/O is *not* cancelled by a `context.Context`. If you have a `ctx` with a deadline, bridge it to the connection with `conn.SetDeadline(deadline)` where `deadline, ok := ctx.Deadline()`.

For UDP, dial with `net.Dial("udp", ...)` as before, but `net.Listen` does not accept UDP — it only handles stream-oriented networks. On the server side use `net.ListenPacket("udp", ...)`, which returns a `net.PacketConn` that tracks each client by address.

Tip: In Java, raw TCP/UDP programming uses `java.net.Socket`, `ServerSocket`, `DatagramSocket`, and `DatagramPacket`. Go's `net.Conn` is a simpler unified abstraction: it is an `io.Reader` and `io.Writer`, so you can layer a `bufio.Scanner` or a `json.Decoder` directly on top of it without any adapter code.

TLS with `crypto/tls`

The `crypto/tls` package provides TLS 1.2 and 1.3 support for both servers and clients. For HTTP, the standard library handles TLS transparently — you mostly just point it at your certificate files. For raw TCP, `crypto/tls` wraps a `net.Conn` into an encrypted connection that still satisfies `io.Reader` and `io.Writer`.

HTTPS Server

The simplest way to add TLS to an HTTP server is `http.ListenAndServeTLS`:

```
func ListenAndServeTLS(addr, certFile, keyFile string, handler Handler) error
```

```
mux := http.NewServeMux()
mux.HandleFunc("GET /songs/", listSongs)
http.ListenAndServeTLS(":443", "cert.pem", "key.pem", mux)
```

For more control, build a `tls.Config` and pass it to `http.Server`:

```
cert, err := tls.LoadX509KeyPair("cert.pem", "key.pem")  // load certificate and private key
if err != nil {
    log.Fatal(err)
}
srv := &http.Server{
    Addr:    ":443",
    Handler: mux,
    TLSConfig: &tls.Config{
        Certificates: []tls.Certificate{cert},  // one or more certificates
        MinVersion:   tls.VersionTLS12,          // reject TLS 1.0 and 1.1
    },
}
srv.ListenAndServeTLS("", "")  // cert and key already in TLSConfig; pass empty strings
```

TLS Client Configuration

The default `http.Client` already validates server certificates against the system trust store — you get HTTPS for free just by using an `https://` URL. To customize TLS behavior, set `TLSClientConfig` on `http.Transport`:

```
client := &http.Client{
    Transport: &http.Transport{
        TLSClientConfig: &tls.Config{
            MinVersion: tls.VersionTLS12,  // require TLS 1.2+
        },
    },
}
```

To trust a custom certificate authority (common in internal services):

```
pool := x509.NewCertPool()
caCert, _ := os.ReadFile("internal-ca.pem")
pool.AppendCertsFromPEM(caCert)

client := &http.Client{
    Transport: &http.Transport{
        TLSClientConfig: &tls.Config{
            RootCAs: pool,  // trust this CA in addition to the system store
        },
    },
}
```

Trap: `tls.Config{InsecureSkipVerify: true}` disables all certificate validation. It is occasionally used in local development against self-signed certs, but it silently makes man-in-the-middle attacks undetectable. Never use it in production code, and never commit it to a shared repository.

Raw TLS Connections

`crypto/tls` mirrors the `net` package: `tls.Dial` and `tls.Listen` replace `net.Dial` and `net.Listen` and return a `*tls.Conn`, which implements `net.Conn` — and therefore `io.Reader` and `io.Writer`.

```
func Dial(network, addr string, config *tls.Config) (*tls.Conn, error)        // TLS client
func Listen(network, laddr string, config *tls.Config) (net.Listener, error)  // TLS listener
```

A TLS client connecting to a server:

```
conn, err := tls.Dial("tcp", "api.example.com:443", &tls.Config{
    MinVersion: tls.VersionTLS12,
})
if err != nil {
    log.Fatal(err)
}
defer conn.Close()

fmt.Fprintf(conn, "GET / HTTP/1.0\r\nHost: api.example.com\r\n\r\n")
io.Copy(os.Stdout, conn)
```

`tls.NewListener` wraps an existing `net.Listener` to add TLS to any protocol:

```
func NewListener(inner net.Listener, config *tls.Config) net.Listener

ln, _ := net.Listen("tcp", ":9443")
tlsLn := tls.NewListener(ln, &tls.Config{Certificates: []tls.Certificate{cert}})
for {
    conn, _ := tlsLn.Accept()   // conn is a *tls.Conn wrapping a net.Conn
    go handleConn(conn)
}
```

Tip: In Java, TLS requires `SSLContext`, `KeyManagerFactory`, `TrustManagerFactory`, and an `SSLSocketFactory` or `SSLServerSocketFactory` — a ceremony that takes dozens of lines even for the common case. Go's `crypto/tls` collapses this to a `tls.Config` struct and a single function call. For HTTPS, `http.ListenAndServeTLS` reduces it further to two file paths.

Try It

Type this in and run it. It wires together the four headline APIs of this chapter — a `ServeMux` with method routing and a path wildcard, a logging middleware, JSON encoding to the response, and a JSON-decoding HTTP client — in one self-contained program. It uses `httptest.NewServer` so the whole round trip runs in a single process with no port to pick or `curl` to type.

```
package main

import (
    "encoding/json"
    "fmt"
    "log"
    "net/http"
    "net/http/httptest"
)

type Song struct {
    ID     string `json:"id"`
    Title  string `json:"title"`
    Artist string `json:"artist"`
}

var catalog = map[string]Song{
    "1": {ID: "1", Title: "Si Antes Te Hubiera Conocido", Artist: "Karol G"},
    "2": {ID: "2", Title: "Luna", Artist: "Feid"},
}

func logging(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        log.Printf("%s %s", r.Method, r.URL.Path)  // log every request
        next.ServeHTTP(w, r)                        // then call the handler
    })
}

func getSong(w http.ResponseWriter, r *http.Request) {
    song, ok := catalog[r.PathValue("id")]
    if !ok {
        http.Error(w, "no existe", http.StatusNotFound)
        return
    }
    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(song)
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /songs/{id}/", getSong)

    srv := httptest.NewServer(logging(mux))  // start a real server on a random port
    defer srv.Close()

    resp, err := http.Get(srv.URL + "/songs/1/")
    if err != nil {
```

```
        log.Fatal(err)
    }
    defer resp.Body.Close()

    var s Song
    json.NewDecoder(resp.Body).Decode(&s)
    fmt.Printf("%d %s by %s\n", resp.StatusCode, s.Title, s.Artist)
    // 200 Si Antes Te Hubiera Conocido by Karol G
}
```

Try these modifications:

- Request `/songs/99/` instead of `/songs/1/` and confirm you get a `404` with the body `no existe`.
- Add a second middleware that sets a `X-Powered-By` header, and chain it inside `logging`.
- Add a `POST /songs/{id}/` route that uses `json.NewDecoder(r.Body)` to decode a `Song` from the request body and stores it in `catalog`.

Key Points

- `json.Marshal` encodes a Go value to `[]byte`; `json.Unmarshal` decodes `[]byte` into a pointer.
- Struct tags control JSON field names, `omitempty` (skip zero values), and `-` (always skip).
- Use the streaming `json.NewEncoder` and `json.NewDecoder` when reading from or writing to an `io.Reader`/`io.Writer` to avoid allocating the entire payload.
- `http.Handler` is the core interface; `http.HandlerFunc` adapts a plain function to satisfy it (Chapter 6 pattern).
- Go 1.22 `ServeMux` supports method routing (`GET /path/`) and path wildcards (`/songs/{id}/`) in the standard library — no third-party router needed for most services.
- Middleware is a function `func(http.Handler) http.Handler`; compose layers by wrapping.
- Always close `resp.Body` after an HTTP client response — `defer resp.Body.Close()` immediately after the error check.
- Reuse `http.Client` across requests; it manages a connection pool internally.
- `import _ "net/http/pprof"` registers live profiling endpoints on `DefaultServeMux`; use `go tool pprof -http` to analyze; never expose these on a public port.
- `encoding/xml` mirrors `encoding/json` exactly — same `Marshal`/`Unmarshal` API, same tag mechanism, different key (`xml` instead of `json`).
- `net.Dial` and `net.Listen` give raw TCP/UDP access; `net.Conn` implements `io.Reader` and `io.Writer`, so all `io` utilities compose with it.
- `http.ListenAndServeTLS` adds TLS to an HTTP server with two file paths; `tls.Config` controls certificate loading, minimum version, and custom CA trust.
- `tls.Dial` and `tls.Listen` mirror `net.Dial`/`net.Listen` for raw TLS; `tls.Conn` satisfies `net.Conn` so existing `io` code needs no changes.
- Never use `InsecureSkipVerify: true` in production — it disables all certificate validation.

Exercises

1. **Think about it:** In Java with Spring MVC or JAX-RS, you annotate a class method with `@GetMapping("/songs/{id}")` or `@GET @Path("/songs/{id}")` and the framework discovers handlers via reflection and classpath scanning. In Go, you call `mux.HandleFunc("GET /songs/{id}/", getSong)` explicitly in `main`. What are the tradeoffs of each approach? Consider startup time, debuggability, IDE navigation, and what happens when two handlers are registered for the same pattern.

2. **What does this print?**

```go
package main

import (
    "encoding/json"
    "fmt"
)

type Artist struct {
    Name    string `json:"name"`
    Country string `json:"country,omitempty"`
    Secret  string `json:"-"`
}

func main() {
    a := Artist{Name: "Chicane", Country: "", Secret: "UK"}
    data, _ := json.Marshal(a)
    fmt.Println(string(data))

    var b Artist
    json.Unmarshal([]byte(`{"name":"Darude","secret":"Finland"}`), &b)
    fmt.Printf("Name: %s, Secret: %q\n", b.Name, b.Secret)
}
```

3. **Calculation:** Consider the following `ServeMux` registration and the three incoming requests below. For each request, state which handler function is called. If no handler runs, give the HTTP error status the mux returns, and say whether the mux failed to match the path at all (`404 Not Found`) or matched the path pattern but not the request method (`405 Method Not Allowed`).

```go
mux := http.NewServeMux()
mux.HandleFunc("GET /tracks/",       listTracks)
mux.HandleFunc("GET /tracks/{id}/",  getTrack)
mux.HandleFunc("POST /tracks/",      createTrack)
```

   a. `GET /tracks/`
   b. `GET /tracks/42/`
   c. `DELETE /tracks/7/`

4. **Where is the bug?**

```go
package main

import (
    "fmt"
    "io"
    "net/http"
)

func fetchLyrics(url string) (string, error) {
    resp, err := http.Get(url)
    if err != nil {
        return "", err
    }
    body, err := io.ReadAll(resp.Body)
    if err != nil {
        return "", err
    }
```

```
    return string(body), nil
}

func main() {
    lyrics, err := fetchLyrics("https://api.example.com/lyrics/sandstorm")
    if err != nil {
        fmt.Println("error:", err)
        return
    }
    fmt.Println(lyrics)
}
```

5. **Write a program:** Build a small in-memory HTTP server that manages a list of songs. Define a `Song` struct with fields `ID int`, `Title string`, and `Artist string`, all with appropriate `json` tags. Store songs in a package-level `map[int]Song`. Register two routes using a `http.NewServeMux()`:
   - `GET /songs/` returns all songs as a JSON array.
   - `GET /songs/{id}/` returns the single song with that ID, or HTTP 404 if not found. Pre-populate the map with two songs (use Darude or Chicane tracks). Start the server on `:8080`.

Chapter 16

gRPC

The previous chapter built services that speak JSON over HTTP — the lingua franca of the public web. That works, but it leaves a lot on the table for service-to-service traffic inside your own systems. JSON is text, so every request pays to serialize numbers into strings and parse them back. The contract lives in documentation (or in your head), so a typo in a field name fails silently at runtime rather than at compile time. And every endpoint reinvents pagination, streaming, deadlines, and error codes by hand.

gRPC fixes all three. You describe your service once in a `.proto` file — the messages and the methods — and a code generator produces strongly typed Go (and Java, and Python, and...) for both sides. Messages travel as compact binary over HTTP/2, which also gives you multiplexed streams for free. If you have used Java's gRPC stack (the `protobuf-maven-plugin`, `StreamObserver`, blocking and async stubs), the concepts here are identical — the contract is the same `.proto` — but Go's generated code is far less ceremonious. Think of it as the typed, binary, streaming counterpart to the REST services from Chapter 15: same network, very different ergonomics.

RPC stands for Remote Procedure Call. As programmers we are very comfortable with function calls — also known as procedure calls. They are one of the early concepts that we learn in programming, so RPC endeavors to make calling a remote service as easy as calling a local function. Unfortunately, network failures, object serialization, remote failures, and delays are all challenges that crop up with RPCs that aren't present with local functions. gRPC — Google's version of RPC — is still a useful tool to use, but it is far from simple.

This chapter gives a brief introduction to Protocol Buffers and the `.proto` language, the `protoc` toolchain that generates Go code, implementing and running a gRPC server, calling it from a client, all four kinds of RPC (unary and the three streaming flavors), propagating deadlines and metadata, returning typed errors with status codes, interceptors (gRPC's middleware), and securing the wire with TLS. We encourage you to learn more from the official gRPC documentation (The gRPC Authors 2025).

Protocol Buffers

A gRPC service starts with a schema written in **Protocol Buffers** (protobuf), Google's language-neutral interface definition language. We can only use types that gRPC knows about or that we have defined with its definition language. A `.proto` file declares the *messages* — gRPC lingo for *types* — and the *services* — the methods that use those types. Code for both the client and server is generated from it, so they can consistently use field names and types when communicating — even using differing programming languages.

Here is the schema we will use for the rest of the chapter — a small song catalog:

```
syntax = "proto3";                      // always proto3 for modern gRPC

package music.v1;                       // protobuf namespace
```

```
option go_package = "example/musicpb";   // Go import path for generated code

message Song {
  string id     = 1;                     // field numbers, not values
  string title  = 2;
  string artist = 3;
  int32  bpm    = 4;
}

message GetSongRequest {
  string id = 1;
}

service Jukebox {
  rpc GetSong(GetSongRequest) returns (Song);  // one request, one response
}
```

Notice that if you change **message** to **class** the message definitions look almost like Java classes, except for the numbers after each field. The numbers after each field (`= 1`, `= 2`) are **field tags**, not default values. They are how protobuf identifies a field on the wire. The *name* is used to generate methods and fields, and the *field tags* are used to encode data to send over the network — the field *name* never travels, only its tag. This is why protobuf payloads are small, and why you can rename a field freely but must never reuse or change a tag number.

Just like Go, Protocol Buffers have built-in types. The scalar types map onto Go types predictably:

proto type	Go type	Java type
`string`	`string`	`String`
`bool`	`bool`	`boolean`
`int32`, `sint32`	`int32`	`int`
`int64`, `sint64`	`int64`	`long`
`double`	`float64`	`double`
`float`	`float32`	`float`
`bytes`	`[]byte`	`ByteString`
`repeated T`	`[]T`	`List<T>`
`map<K,V>`	`map[K]V`	`Map<K,V>`

Wut: In proto3 every scalar field has a zero-value default and there is no built-in "was it set?" bit for scalars. A `bpm` of `0` is indistinguishable from "bpm not provided." When that distinction matters, either wrap the field (`optional int32 bpm = 4;`, which generates a `*int32` in Go) or use a dedicated message. This is the protobuf analog of the `int` vs `Integer` boxing decision in Java.

The `protoc` Toolchain

Go code does not read `.proto` files at runtime. You run a compiler, `protoc`, ahead of time to generate `.go` files. Two plugins do the work: `protoc-gen-go` generates the message structs, and `protoc-gen-go-grpc` generates the client and server stubs.

Install the plugins (they are ordinary Go tools) and make sure `protoc` itself is on your `PATH`:

```
# the two code generators
go install google.golang.org/protobuf/cmd/protoc-gen-go@latest
```

```
go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@latest
```

The `protoc` compiler itself does not come from `go install` — you get it from your platform's package manager:

```
# macOS (Homebrew)
brew install protobuf

# Debian/Ubuntu
apt install protobuf-compiler

# Windows (Chocolatey)
choco install protoc
```

On Windows you can also use Scoop (`scoop install protobuf`). Whichever you pick, make sure the resulting `protoc` is on your `PATH` — and that the Go tool directory (`go env GOBIN`, or `$(go env GOPATH)/bin`) is too, so the two plugins are found.

Generate the Go code from `music.proto`:

```
protoc --go_out=. --go_opt=paths=source_relative \
       --go-grpc_out=. --go-grpc_opt=paths=source_relative \
       music.proto
```

This writes two files next to the proto: `music.pb.go` (structs representing the message types `Song`, `GetSongRequest`, ...) and `music_grpc.pb.go` (the `JukeboxServer` interface, the `JukeboxClient` stub that performs the gRPC calls, and the `RegisterJukeboxServer` registration function). You commit the generated files alongside your hand-written code and regenerate whenever the `.proto` changes.

The plugins you installed with `go install` are *host tools* — they run on your machine to produce code and never ship with your program. The generated code, however, imports two *runtime libraries* that your program does link against: `google.golang.org/protobuf` (the wire encoding) and `google.golang.org/grpc` (the client and server machinery). Those have to be real dependencies in your module's `go.mod`, so add them with `go get`:

```
go get google.golang.org/grpc
go get google.golang.org/protobuf
```

In practice a single `go mod tidy` after generating will discover both from the new imports and pin them for you. The Java analog is the split between the `protoc` plugin in your build file and the `grpc-stub`/`protobuf-java` artifacts on your classpath — Go just keeps the build tools out of `go.mod` entirely.

Tip: Memorizing that `protoc` invocation is nobody's idea of fun. Most teams use `buf` (`buf.build`) instead: a `buf.gen.yaml` describes the plugins once, and `buf generate` runs them — plus `buf lint` and `buf breaking` catch style problems and wire-incompatible changes before they ship. It is the protobuf equivalent of replacing a hand-rolled `protobuf-maven-plugin` block with a single linted config.

Trap: Never hand-edit the generated `*.pb.go` files. They are overwritten on every regeneration. Put your logic in separate files that *use* the generated types.

Building a gRPC Server

There are two parts of the gRPC server that get generated for us: the interface that we need to implement and an embedded type.

The interface is named after the gRPC service — with a `Server` suffix added — and each of the RPCs for the service shows up as a method. For example, the generated `music_grpc.pb.go` declares the interface:

```
// generated --- one method per rpc in the service
type JukeboxServer interface {
    GetSong(context.Context, *GetSongRequest) (*Song, error)
    mustEmbedUnimplementedJukeboxServer()  // forward-compatibility guard
}
```

That last unexported method is deliberate: you cannot satisfy the interface by accident. Instead you **embed** the generated `UnimplementedJukeboxServer` (Chapter 6 embedding) into your type, which supplies a default "unimplemented" body for every method. Then you override only the methods you actually handle. The `Unimplemented` prefix may sound awkward, but the name comes from the fact that it provides an implementation of every method that just returns the `codes.Unimplemented` error.

In the code below, notice that `jukeboxServer` does not need to explicitly say that it implements the `JukeboxServer` interface, but it does need to embed the `musicpb.UnimplementedJukeboxServer` type.

```
package main

import (
    "context"
    "sync"

    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/status"
    "example/musicpb"
)

type jukeboxServer struct {
    musicpb.UnimplementedJukeboxServer            // embed for forward compatibility
    mu      sync.Mutex                            // guards catalog (Chapter 11)
    catalog map[string]*musicpb.Song
}

func (s *jukeboxServer) GetSong(
    ctx context.Context, req *musicpb.GetSongRequest,
) (*musicpb.Song, error) {
    s.mu.Lock()
    defer s.mu.Unlock()

    song, ok := s.catalog[req.GetId()]
    if !ok {
        return nil, status.Errorf(codes.NotFound, "no song with id %q", req.GetId())
    }
    return song, nil
}
```

Notice `req.GetId()` rather than `req.Id`. The generated code gives every field a getter that is `nil`-safe: calling `GetId()` on a `nil *GetSongRequest` returns the zero value instead of panicking. Prefer the getters when reading request fields.

Wiring it up looks a lot like the raw `net.Listen` server from Chapter 15, except a `grpc.Server` sits between the listener and your handler:

```
func main() {
    lis, err := net.Listen("tcp", ":50051")  // gRPC convention: port 50051
```

```
    if err != nil {
        log.Fatal(err)
    }

    s := grpc.NewServer()
    musicpb.RegisterJukeboxServer(s, &jukeboxServer{
        catalog: map[string]*musicpb.Song{
            "1": {Id: "1", Title: "Monaco", Artist: "Bad Bunny", Bpm: 130},
            "2": {Id: "2", Title: "Despecha", Artist: "Rosalia", Bpm: 130},
        },
    })

    log.Printf("jukebox listening on %s", lis.Addr())
    log.Fatal(s.Serve(lis))  // blocks until the server stops
}
```

The key functions that you will use:

```
func NewServer(opt ...ServerOption) *Server     // new server; opts add interceptors, TLS
func (s *Server) Serve(lis net.Listener) error  // accept connections until Stop/GracefulStop
func (s *Server) GracefulStop()                 // stop accepting, drain in-flight RPCs
func (s *Server) Stop()                         // stop immediately, cancelling in-flight RPCs
```

Just like the `http.Server` from Chapter 15, prefer `GracefulStop` on shutdown so in-flight RPCs finish instead of being severed mid-call.

Calling a gRPC Server

The client side is generated too. `NewJukeboxClient` wraps a connection and returns a typed stub whose methods look exactly like local function calls — the network is hidden behind the generated code.

```
func NewClient(target string, opts ...DialOption) (*ClientConn, error)  // lazy connection
func NewJukeboxClient(cc grpc.ClientConnInterface) JukeboxClient        // generated typed stub
```

```
package main

import (
    "context"
    "log"
    "time"

    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials/insecure"
    "example/musicpb"
)

func main() {
    conn, err := grpc.NewClient(
        "localhost:50051",
        grpc.WithTransportCredentials(insecure.NewCredentials()),  // plaintext; see TLS section
    )
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()
```

```
    client := musicpb.NewJukeboxClient(conn)

    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()

    song, err := client.GetSong(ctx, &musicpb.GetSongRequest{Id: "1"})
    if err != nil {
        log.Fatal(err)
    }
    log.Printf("%s by %s (%d BPM)", song.GetTitle(), song.GetArtist(), song.GetBpm())
    // Monaco by Bad Bunny (130 BPM)
}
```

Trap: `grpc.Dial` is the old constructor you will see in pre-2024 examples and tutorials. It is deprecated in favor of `grpc.NewClient`. The big behavioral difference: `Dial` connected eagerly (and `WithBlock` waited for the handshake), while `NewClient` connects lazily on the first RPC. Do not block startup waiting for a connection — let the first call establish it, and rely on deadlines to bound failures.

Tip: A `*grpc.ClientConn` is expensive to create and safe for concurrent use. Create one per backend at startup and share it across all goroutines, exactly as you reuse an `http.Client` (Chapter 15) or a `*sql.DB`. Do not open a new connection per request.

The Four Kinds of RPC

HTTP/2 carries multiple message frames over a single connection, so gRPC supports streaming in either direction — not just one-shot request/response. There are four method shapes, declared in the `.proto` with the `stream` keyword:

```
service Jukebox {
  rpc GetSong(GetSongRequest) returns (Song);              // unary
  rpc ListSongs(ListRequest) returns (stream Song);        // server streaming
  rpc AddSongs(stream Song) returns (AddSummary);          // client streaming
  rpc Party(stream SongRequest) returns (stream Song);     // bidirectional
}
```

You have already seen unary above. The streaming variants generate strongly typed stream handles backed by Go generics (Chapter 18): `grpc.ServerStreamingServer[T]`, `grpc.ClientStreamingServer[Req, Res]`, and so on. A stream is the RPC analog of a Go channel (Chapter 10): you `Send` and `Recv` values until `io.EOF`.

Server Streaming

The server sends many messages for one request — think "list" or "subscribe." On the server, you receive the request plus a stream and call `Send` in a loop:

```
func (s *jukeboxServer) ListSongs(
    req *musicpb.ListRequest,
    stream grpc.ServerStreamingServer[musicpb.Song],
) error {
    s.mu.Lock()
    defer s.mu.Unlock()
    for _, song := range s.catalog {
        if err := stream.Send(song); err != nil {  // one frame per song
            return err
```

```
        }
    }
    return nil  // returning ends the stream cleanly
}
```

The client receives until `io.EOF`:

```
stream, err := client.ListSongs(ctx, &musicpb.ListRequest{})
if err != nil {
    log.Fatal(err)
}
for {
    song, err := stream.Recv()
    if err == io.EOF {
        break  // server closed the stream --- normal end
    }
    if err != nil {
        log.Fatal(err)
    }
    log.Println(song.GetTitle())
}
```

Client Streaming

The client sends many messages and gets one response back — think "bulk upload." The client calls `Send` repeatedly, then `CloseAndRecv` to signal the end and read the single reply:

```
stream, err := client.AddSongs(ctx)
if err != nil {
    log.Fatal(err)
}
for _, song := range incoming {
    if err := stream.Send(song); err != nil {
        log.Fatal(err)
    }
}
summary, err := stream.CloseAndRecv()  // close the send side, get the response
if err != nil {
    log.Fatal(err)
}
log.Printf("added %d songs", summary.GetCount())
```

On the server, you `Recv` in a loop and `SendAndClose` once at the end:

```
func (s *jukeboxServer) AddSongs(
    stream grpc.ClientStreamingServer[musicpb.Song, musicpb.AddSummary],
) error {
    var count int32
    for {
        song, err := stream.Recv()
        if err == io.EOF {
            return stream.SendAndClose(&musicpb.AddSummary{Count: count})
        }
        if err != nil {
            return err
        }
```

```
        s.mu.Lock()
        s.catalog[song.GetId()] = song
        s.mu.Unlock()
        count++
    }
}
```

Bidirectional Streaming

Both sides stream independently over the one connection. Because the directions are independent, a common pattern is to read in one goroutine (Chapter 10) and write in another. Here the server simply echoes a matching song for every request it receives:

```
func (s *jukeboxServer) Party(
    stream grpc.BidiStreamingServer[musicpb.SongRequest, musicpb.Song],
) error {
    for {
        req, err := stream.Recv()
        if err == io.EOF {
            return nil  // client closed its send side
        }
        if err != nil {
            return err
        }
        s.mu.Lock()
        song := s.catalog[req.GetId()]
        s.mu.Unlock()
        if song != nil {
            if err := stream.Send(song); err != nil {
                return err
            }
        }
    }
}
```

The stream handle methods worth memorizing:

```
// server side
func (x ServerStreamingServer[T]) Send(*T) error                      // push to client
func (x ClientStreamingServer[Req, Res]) Recv() (*Req, error)         // pull from client
func (x ClientStreamingServer[Req, Res]) SendAndClose(*Res) error     // final reply + close
// client side
func (x ServerStreamingClient[T]) Recv() (*T, error)                  // pull from server
func (x ClientStreamingClient[Req, Res]) Send(*Req) error             // push to server
func (x ClientStreamingClient[Req, Res]) CloseAndRecv() (*Res, error) // close + read reply
func (x BidiStreamingClient[Req, Res]) CloseSend() error              // close send side only
```

Wut: A streaming RPC's `Send` does not guarantee the peer received the message — only that it was handed to the transport. Ordering is guaranteed by HTTP/2, but delivery is not — the only way to know the server *processed* your messages is to read the response (or, for server streams, to reach `io.EOF` without error). If the call fails with a network error instead — the connection drops, say — the server may have processed your messages and the response was lost, or it may never have received them at all. Always check the error returned by `CloseAndRecv`/`Recv`, not just the per-`Send` errors.

Deadlines and Metadata

Every RPC takes a `context.Context` (Chapter 12), and gRPC propagates its deadline *across the network*. When the client sets `context.WithTimeout`, the server's handler receives a context with the same deadline — so a slow handler can bail out early by watching `ctx.Done()` instead of doing work nobody is waiting for anymore. This is a big upgrade over plain HTTP, where you have to plumb timeouts through headers yourself.

```
ctx, cancel := context.WithTimeout(context.Background(), 200*time.Millisecond)
defer cancel()

_, err := client.GetSong(ctx, &musicpb.GetSongRequest{Id: "1"})
// if the server takes longer than 200ms, err has code DeadlineExceeded
```

Metadata is gRPC's equivalent of HTTP headers: a string-keyed multimap carried alongside the request, used for things like auth tokens, request IDs, and tracing. The client attaches it to the context; the server reads it back out.

```
import "google.golang.org/grpc/metadata"

// client: attach an auth token
ctx = metadata.AppendToOutgoingContext(ctx, "authorization", "Bearer hunter2")

// server: read it
func (s *jukeboxServer) GetSong(
    ctx context.Context, req *musicpb.GetSongRequest,
) (*musicpb.Song, error) {
    md, ok := metadata.FromIncomingContext(ctx)
    if ok {
        tokens := md.Get("authorization")  // []string
        _ = tokens
    }
    // ...
}
```

Errors and Status Codes

A gRPC handler returns an ordinary Go `error` (Chapter 9), but gRPC wants that error to carry a **status code** — a small enum that both sides understand regardless of language. Return `nil` for success; return a `status.Error` (or `status.Errorf`) with a code for failure.

```
func Errorf(c codes.Code, format string, a ...any) error  // build an error with a status code
func FromError(err error) (*Status, bool)                 // extract the status from an error

// server
return nil, status.Errorf(codes.InvalidArgument, "id must not be empty")

// client: inspect the code
song, err := client.GetSong(ctx, req)
if err != nil {
    st, _ := status.FromError(err)
    switch st.Code() {
    case codes.NotFound:
        log.Println("no existe esa cancion")
    case codes.DeadlineExceeded:
        log.Println("el servidor tardo demasiado")
    default:
```

```
        log.Printf("rpc failed: %v", st.Message())
    }
}
```

The codes you will reach for most, and their rough equivalents:

gRPC code	When to use	HTTP analog	Java analog
`OK`	success (return `nil`)	200	normal return
`InvalidArgument`	caller sent bad input	400	`IllegalArgumentException`
`NotFound`	resource does not exist	404	`NoSuchElementException`
`AlreadyExists`	create conflict	409	duplicate-key exception
`PermissionDenied`	authenticated but not allowed	403	`SecurityException`
`Unauthenticated`	missing/invalid credentials	401	`AuthenticationException`
`DeadlineExceeded`	RPC ran past its deadline	504	`TimeoutException`
`Unavailable`	backend down; safe to retry	503	`ConnectException`
`Internal`	a bug on the server	500	unchecked `RuntimeException`

Trap: Do not return a bare `errors.New("...")` or `fmt.Errorf("...")` from a handler. gRPC wraps any error without a status code as `codes.Unknown`, which tells the client nothing useful and defeats retry logic. Always wrap failures in `status.Errorf` with the most specific code that fits.

Interceptors

An **interceptor** is gRPC's middleware: a function that wraps every RPC so you can add logging, authentication, metrics, or recovery in one place. It is the same idea as the `func(http.Handler) http.Handler` middleware from Chapter 15, just with a gRPC-shaped signature. There are two flavors — one for unary RPCs and one for streams.

```
// unary server interceptor
func(ctx context.Context, req any, info *grpc.UnaryServerInfo,
     handler grpc.UnaryHandler) (resp any, err error)
```

A logging-plus-timing interceptor:

```
func logging(ctx context.Context, req any, info *grpc.UnaryServerInfo,
    handler grpc.UnaryHandler) (any, error) {

    start := time.Now()
    resp, err := handler(ctx, req)  // call the actual RPC
    log.Printf("%s took %s (err=%v)", info.FullMethod, time.Since(start), err)
    return resp, err
}

func main() {
    s := grpc.NewServer(grpc.ChainUnaryInterceptor(logging))  // chain runs left to right
    // ... register and serve
}
```

`grpc.ChainUnaryInterceptor(a, b, c)` composes several so the first listed runs outermost — the same ordering as the `chain` helper for HTTP middleware in Chapter 15. The client side has the mirror-image `grpc.WithChainUnaryInterceptor` for adding outbound auth headers, retries, and the like.

TLS

The `insecure.NewCredentials()` we used above sends everything in plaintext — fine for local development, never for production. gRPC rides on the same `crypto/tls` machinery as HTTPS (Chapter 15), exposed through the `credentials` package.

On the server, load a certificate and pass it as a `ServerOption`:

```
import "google.golang.org/grpc/credentials"

creds, err := credentials.NewServerTLSFromFile("cert.pem", "key.pem")
if err != nil {
    log.Fatal(err)
}
s := grpc.NewServer(grpc.Creds(creds))  // every connection is now TLS
```

On the client, trust the server's CA and dial with transport credentials:

```
creds, err := credentials.NewClientTLSFromFile("ca.pem", "")  // "" = use cert's hostname
if err != nil {
    log.Fatal(err)
}
conn, err := grpc.NewClient("jukebox.example.com:443", grpc.WithTransportCredentials(creds))
```

Tip: For service-to-service calls inside a cluster, most teams let a service mesh (Istio, Linkerd) terminate mTLS transparently, so application code can dial with `insecure` while the sidecar encrypts the wire. When you do TLS in-process, the rules from Chapter 15 apply unchanged — set a minimum version and never disable verification with `InsecureSkipVerify` in production.

Try It

Type this in and run it. gRPC needs a generated stub, so this is a three-step program: write the `.proto`, generate the Go, then run a single file that starts the server in a goroutine and calls it as a client — a full round trip in one process, no separate terminals.

First, `music.proto`:

```
syntax = "proto3";
package music.v1;
option go_package = "example/musicpb";

message GetSongRequest { string id = 1; }
message Song {
  string id     = 1;
  string title  = 2;
  string artist = 3;
}

service Jukebox {
  rpc GetSong(GetSongRequest) returns (Song);
}
```

Generate the code (see the toolchain section for installing the plugins):

```
protoc --go_out=. --go_opt=paths=source_relative \
       --go-grpc_out=. --go-grpc_opt=paths=source_relative \
       music.proto
```

Then `main.go`:

```
package main

import (
    "context"
    "log"
    "net"
    "time"

    "google.golang.org/grpc"
    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/credentials/insecure"
    "google.golang.org/grpc/status"
    "example/musicpb"
)

type server struct {
    musicpb.UnimplementedJukeboxServer
}

func (server) GetSong(_ context.Context, req *musicpb.GetSongRequest) (*musicpb.Song, error) {
    if req.GetId() != "1" {
        return nil, status.Errorf(codes.NotFound, "no song %q", req.GetId())
    }
    return &musicpb.Song{Id: "1", Title: "Todo De Ti", Artist: "Rauw Alejandro"}, nil
}

func main() {
    lis, _ := net.Listen("tcp", "localhost:0")  // :0 = pick any free port
    s := grpc.NewServer()
    musicpb.RegisterJukeboxServer(s, server{})
    go s.Serve(lis)            // run the server in the background
    defer s.GracefulStop()

    conn, err := grpc.NewClient(lis.Addr().String(),
        grpc.WithTransportCredentials(insecure.NewCredentials()))
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()
    client := musicpb.NewJukeboxClient(conn)

    ctx, cancel := context.WithTimeout(context.Background(), 2*time.Second)
    defer cancel()

    song, err := client.GetSong(ctx, &musicpb.GetSongRequest{Id: "1"})
    if err != nil {
        log.Fatal(err)
    }
    log.Printf("%s by %s", song.GetTitle(), song.GetArtist())
    // Todo De Ti by Rauw Alejandro
}
```

Try these modifications:

- Request id `"2"` and confirm the error code is `NotFound` via `status.FromError`.
- Add a `ListSongs(ListRequest) returns (stream Song)` RPC, regenerate, and stream two songs back to the client.
- Add a unary interceptor that logs `info.FullMethod` and the latency of each call.

Key Points

- gRPC is contract-first: you define messages and services in a `.proto` file and generate typed Go for both client and server, so the two cannot disagree about the wire format.
- Protocol Buffers encode data as compact binary; field *tag numbers*, not names, travel on the wire — never reuse or renumber a tag.
- `protoc` with `protoc-gen-go` and `protoc-gen-go-grpc` generates `*.pb.go` files; commit them and regenerate when the `.proto` changes. `buf` is the friendlier modern front end.
- Embed `UnimplementedXxxServer` in your server type for forward compatibility, then override the methods you handle.
- Use field getters (`req.GetId()`) when reading messages — they are `nil`-safe.
- `grpc.NewServer` + `Serve` runs a server (prefer `GracefulStop` on shutdown); `grpc.NewClient` + the generated `NewXxxClient` make calls. `grpc.Dial` is deprecated.
- Reuse a single `*grpc.ClientConn` across goroutines, like an `http.Client` or `*sql.DB`.
- Four RPC shapes: unary, server streaming, client streaming, bidirectional — streams expose `Send`/`Recv` and end at `io.EOF`.
- Deadlines from the client's `context` propagate across the network; metadata is gRPC's header map.
- Return failures as `status.Errorf(code, ...)` with a specific `codes` value — a bare `error` becomes the useless `codes.Unknown`.
- Interceptors are gRPC's middleware; TLS comes from the `credentials` package over the same `crypto/tls` stack as HTTPS.

Exercises

1. **Think about it:** Chapter 15 built a JSON-over-HTTP service for the same song catalog. For a public API consumed by third-party web and mobile clients you do not control, and for high-volume internal traffic between your own microservices, which would you reach for — REST/JSON or gRPC — and why? Consider browser support, human debuggability with `curl`, payload size, schema evolution, and streaming.

2. **What does this do?** A teammate writes this proto and regenerates, then is surprised the change "broke" old clients:

   ```
   // before
   message Song {
     string id    = 1;
     string title = 2;
   }
   // after
   message Song {
     string id     = 2;
     string title  = 1;
   }
   ```

   They only swapped the tag numbers, not the field names or types. What happens when a new server sends a `Song` to a client built from the *old* proto, and why?

3. **Calculation:** A unary RPC handler does 80 ms of work. A client calls it with `context.WithTimeout(ctx, 50*time.Millisecond)`. The server-side handler does not check `ctx.Done()` and runs to completion.

(a) What status code does the *client* observe?
(b) Does the server's handler still finish its 80 ms of work?
(c) What single change makes the server stop early when the deadline passes?

4. **Where is the bug?**

```go
func (s *jukeboxServer) ListSongs(
    req *musicpb.ListRequest,
    stream grpc.ServerStreamingServer[musicpb.Song],
) error {
    for _, song := range s.catalog {
        stream.Send(song)
    }
    return errors.New("done sending")
}
```

There are two distinct problems in this handler. Identify both and say what the client sees for each.

5. **Write a program:** Define a `.proto` with a `Library` service exposing `AddSongs(stream Song) returns (Summary)` where `Summary` has an `int32 count` and an `int32 total_bpm`. Implement the server so it accumulates the count and the sum of all `bpm` fields across the streamed songs, then returns the summary with `SendAndClose`. Write a client that streams three songs and prints the returned count and average BPM. Use `status.Errorf(codes.InvalidArgument, ...)` if any streamed song has an empty `id`.

Chapter 17

Database Access

Almost every real-world application reads and writes a database. Go's `database/sql` package plays the same role as Java's JDBC: it gives you a standard interface that works with any database driver, so your application code stays independent of the specific database engine underneath it. If you already know JDBC, you will recognize most of the ideas here — the idioms are just different enough to matter.

Coming from Java, you may expect a database layer to mean either JDBC's verbose `try/catch/finally` ceremony or a heavyweight ORM like Hibernate with its sessions, lazy-loading proxies, and dialect configuration. Go takes neither extreme: `database/sql` is a thin, standard-library layer that hands you rows and lets you write SQL directly, with no annotations, no entity manager, and no XML. The result is less magic and less to learn — you write the queries, you `Scan` the results, and connection pooling comes built in rather than bolted on.

The `sql.DB` Connection Pool

In JDBC you open a `Connection` to represent a single database connection, and you typically manage a `DataSource` connection pool separately. Go combines these two concepts: `sql.DB` **is** the connection pool.

```
import (
    "database/sql"
    "log"

    _ "github.com/lib/pq" // register the Postgres driver; see the Driver Registration section
)

func main() {
    db, err := sql.Open("postgres", "postgres://user:pass@localhost/music?sslmode=disable")
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()
    // db is ready to use; no actual connection has been made yet
}
```

Wut: `sql.Open` does **not** open a connection. It validates the driver name (and, for most drivers, the DSN format) and returns a pool handle. The first real connection is made lazily on the first query. To verify connectivity at startup, call `db.PingContext(ctx)` immediately after `sql.Open`.

`sql.DB` manages a pool of underlying connections and is **safe for concurrent use by multiple goroutines**. You should create one `sql.DB` per database and share it across your whole application — creating a new `sql.DB` per request is a common mistake that exhausts connection limits quickly.

You can tune the pool with a handful of methods:

```
db.SetMaxOpenConns(25)      // maximum number of open connections
db.SetMaxIdleConns(25)      // maximum number of idle connections kept in the pool
db.SetConnMaxLifetime(time.Hour) // maximum age of a connection before it is recycled
```

The Java JDBC analogy:

JDBC	Go `database/sql`
`DataSource`	`*sql.DB`
`Connection`	internal, managed by the pool
`PreparedStatement`	`*sql.Stmt`
`ResultSet`	`*sql.Rows`
`SQLException`	`error` (idiomatic Go)

Querying the Database

Every query method has a plain form (`Query`, `QueryRow`, `Exec`) and a context-aware form (`QueryContext`, `QueryRowContext`, `ExecContext`). Always use the context-aware forms in production code; they respect cancellation and deadlines (see Chapter 12).

```
// Always prefer these:
db.QueryContext(ctx, query, args...)     // multiple rows
db.QueryRowContext(ctx, query, args...)  // exactly one row
db.ExecContext(ctx, query, args...)      // INSERT / UPDATE / DELETE
```

Trap: The plain `db.Query`, `db.QueryRow`, and `db.Exec` methods use `context.Background()` internally and cannot be cancelled. Using them in a web handler or an RPC server means the query runs to completion even after the client disconnects. Always pass a context.

Querying Multiple Rows

`db.QueryContext` returns `*sql.Rows`, which you iterate with `rows.Next()`.

```
func listSongs(ctx context.Context, db *sql.DB, artist string) ([]Song, error) {
    rows, err := db.QueryContext(ctx,
        "SELECT id, title, artist FROM songs WHERE artist = $1", artist)
    if err != nil {
        return nil, err
    }
    defer rows.Close() // always close rows when done

    var songs []Song
    for rows.Next() {
        var s Song
        if err := rows.Scan(&s.ID, &s.Title, &s.Artist); err != nil {
            return nil, err
        }
        songs = append(songs, s)
    }
```

```
    // rows.Err() reports any error that stopped iteration early
    if err := rows.Err(); err != nil {
        return nil, err
    }
    return songs, nil
}
```

Trap: If you forget `defer rows.Close()` and the function returns early — a `Scan` error, a mid-loop `return` — the connection borrowed from the pool is never returned. (Fully iterating to the end closes the rows automatically, but you should not rely on every path reaching the end.) Under load, the pool is exhausted and new queries block forever. Always `defer rows.Close()` immediately after checking the error from `QueryContext`.

Wut: The placeholder syntax for query parameters is **driver-specific**, not part of `database/sql`. Postgres (`lib/pq`, `pgx`) uses `$1`, `$2`, ...; MySQL and SQLite use `?`; Oracle uses `:name`. The examples in this section use `$1`-style Postgres placeholders; switch to `?` if your driver is MySQL or SQLite, and keep each query consistent with the driver you actually opened.

The Java JDBC equivalent pattern is:

```
try (PreparedStatement ps = conn.prepareStatement(sql)) {
    ps.setString(1, artist);
    try (ResultSet rs = ps.executeQuery()) {
        while (rs.next()) { ... }
    }
}
```

Go's `defer rows.Close()` replaces the `try-with-resources` block.

Querying a Single Row

When you expect exactly one row, use `QueryRowContext`. It returns `*sql.Row`, and you call `Scan` on it directly — no loop needed.

```
func getSong(ctx context.Context, db *sql.DB, id int) (Song, error) {
    var s Song
    err := db.QueryRowContext(ctx,
        "SELECT id, title, artist FROM songs WHERE id = $1", id).
        Scan(&s.ID, &s.Title, &s.Artist)
    if err != nil {
        return Song{}, err
    }
    return s, nil
}
```

If no row matches the query, `Scan` returns the sentinel error `sql.ErrNoRows`. Use `errors.Is` to test for it (Chapter 9):

```
if errors.Is(err, sql.ErrNoRows) {
    return Song{}, fmt.Errorf("song %d not found", id)
}
```

Executing Non-Query Statements

`ExecContext` is for statements that do not return rows: `INSERT`, `UPDATE`, `DELETE`, `CREATE TABLE`, and so on. It returns `sql.Result`, which tells you how many rows were affected and the last inserted auto-increment ID.

```go
func addSong(ctx context.Context, db *sql.DB, title, artist string) (int64, error) {
    result, err := db.ExecContext(ctx,
        "INSERT INTO songs (title, artist) VALUES ($1, $2)", title, artist)
    if err != nil {
        return 0, err
    }
    return result.LastInsertId() // not all drivers support this
}
```

Tip: Postgres does not support `LastInsertId`. Use `RETURNING id` in your `INSERT` statement and `QueryRowContext` + `Scan` to read the generated key back.

```go
var id int64
err := db.QueryRowContext(ctx,
    "INSERT INTO songs (title, artist) VALUES ($1, $2) RETURNING id",
    title, artist).Scan(&id)
```

Scanning Results

`rows.Scan` maps database column values into Go variables. You pass a pointer to each destination variable, in the same order as the columns in your `SELECT` list.

```go
var id    int
var title string
var plays int64
rows.Scan(&id, &title, &plays)
```

`Scan` handles type conversions between the database wire type and the Go type. For example, a SQL `INT` column can scan into `int`, `int32`, `int64`, or `string`. If the conversion is not possible, `Scan` returns an error.

Trap: The number of arguments to `Scan` must match the number of columns returned by your query. A mismatch causes a runtime error, not a compile-time error. If you `SELECT *` and the schema changes, `Scan` will fail — prefer explicit column lists.

Nullable Column Values with `sql.Null[T]`

SQL columns can be `NULL`. If you scan a `NULL` value into a plain Go variable such as `string` or `int`, `Scan` returns an error. Before Go 1.22 you used type-specific helpers like `sql.NullString` and `sql.NullInt64`. Go 1.22 introduced the generic `sql.Null[T]`:

```go
type Null[T any] struct {
    V     T    // the value; zero value of T if NULL
    Valid bool // true if the column is not NULL
}
```

Use it when a column may be `NULL`:

```go
type Song struct {
    ID     int
    Title  string
    Artist string
    Album  sql.Null[string] // NULL when the song is a single
    Plays  sql.Null[int64]  // NULL when plays are not tracked
}
```

```
var s Song
rows.Scan(&s.ID, &s.Title, &s.Artist, &s.Album, &s.Plays)

if s.Album.Valid {
    fmt.Println("Album:", s.Album.V)
} else {
    fmt.Println("Single (no album)")
}
```

Tip: `sql.Null[T]` works with any type `T` accepted by `driver.Value`. For older code (pre-1.22) you will see `sql.NullString`, `sql.NullInt64`, `sql.NullFloat64`, and `sql.NullBool`. They are still valid and work the same way; `sql.Null[T]` just generalizes them.

Transactions

A transaction groups multiple statements into an atomic unit: either all succeed or all are rolled back. In JDBC you call `conn.setAutoCommit(false)` and then `conn.commit()` or `conn.rollback()`. In Go you call `db.BeginTx` to get a `*sql.Tx`, then use `tx.Commit()` or `tx.Rollback()`.

`*sql.Tx` has the same query methods as `*sql.DB`: `QueryContext`, `QueryRowContext`, `ExecContext`, and `PrepareContext`. All statements executed on a `*sql.Tx` run inside the same database transaction.

The Deferred Rollback Pattern

The idiomatic Go pattern for transactions uses `defer` to guarantee a rollback if anything goes wrong:

```
func transferPlay(ctx context.Context, db *sql.DB, fromID, toID int, count int64) error {
    tx, err := db.BeginTx(ctx, nil) // nil uses the default isolation level
    if err != nil {
        return err
    }
    defer tx.Rollback() // no-op if tx.Commit() has already been called

    _, err = tx.ExecContext(ctx,
        "UPDATE songs SET plays = plays - $1 WHERE id = $2", count, fromID)
    if err != nil {
        return err // defer fires tx.Rollback()
    }

    _, err = tx.ExecContext(ctx,
        "UPDATE songs SET plays = plays + $1 WHERE id = $2", count, toID)
    if err != nil {
        return err // defer fires tx.Rollback()
    }

    return tx.Commit() // defer fires tx.Rollback(), but Rollback after Commit is a no-op
}
```

The key insight: `tx.Rollback()` is a no-op if the transaction has already been committed. So you can `defer tx.Rollback()` unconditionally right after `BeginTx`, and then call `tx.Commit()` at the end of the happy path. If anything returns early with an error, the deferred rollback fires and cleans up.

Tip: This pattern is the Go equivalent of Java's `try { ... conn.commit(); } catch (Exception e) { conn.rollback(); }` boilerplate. It is shorter and harder to forget because `defer` always runs, even on panics.

Trap: After `tx.Commit()` or `tx.Rollback()` is called, the `*sql.Tx` is no longer usable. Any further operations on it return `sql.ErrTxDone`.

Prepared Statements

A prepared statement is a pre-compiled SQL template that can be executed multiple times with different parameter values. In JDBC, `PreparedStatement` is the standard way to parameterize queries. In Go, use `db.PrepareContext` to get a `*sql.Stmt`:

```
func bulkInsert(ctx context.Context, db *sql.DB, songs []Song) error {
    stmt, err := db.PrepareContext(ctx,
        "INSERT INTO songs (title, artist) VALUES ($1, $2)")
    if err != nil {
        return err
    }
    defer stmt.Close()

    for _, s := range songs {
        if _, err := stmt.ExecContext(ctx, s.Title, s.Artist); err != nil {
            return err
        }
    }
    return nil
}
```

`*sql.Stmt` has the same execution methods as `*sql.DB`: `QueryContext`, `QueryRowContext`, and `ExecContext`.

Tip: Prepare the statement once and execute it many times. Preparing a statement on every iteration of a loop defeats the purpose — the database has to parse and plan the query each time anyway.

Wut: Go's `database/sql` transparently re-prepares statements when a connection from the pool is replaced. The `*sql.Stmt` handle is associated with the pool, not a single connection. Under the hood, Go may prepare the same statement on multiple connections as needed. This is different from JDBC, where a `PreparedStatement` is tied to a single `Connection`.

You can also prepare a statement inside a transaction:

```
tx, err := db.BeginTx(ctx, nil)
if err != nil { ... }
defer tx.Rollback()

stmt, err := tx.PrepareContext(ctx, "INSERT INTO songs (title, artist) VALUES ($1, $2)")
if err != nil { ... }
defer stmt.Close()
```

Statements prepared on a `*sql.Tx` are scoped to that transaction.

Driver Registration

`database/sql` is driver-neutral. It defines the API; the actual wire protocol to a specific database is provided by a third-party driver package. Drivers register themselves with `database/sql` in their `init()` function (see Chapter 5). You import the driver package for its side effects — you never call it directly.

```
import _ "github.com/lib/pq"              // PostgreSQL driver
import _ "github.com/mattn/go-sqlite3"    // SQLite driver
import _ "github.com/go-sql-driver/mysql" // MySQL driver
```

The `_` alias discards the package name so the compiler does not complain about an unused import. The `init()` function inside the driver package calls `sql.Register("postgres", &Driver{})` (or equivalent), making the driver available to `sql.Open`.

This is the same blank import pattern introduced in Chapter 1 (for side-effect-only imports). The driver package has a valuable side effect — driver registration — but no exported API that your code calls directly.

Tip: `github.com/mattn/go-sqlite3` wraps the C SQLite library, so it requires cgo — you need `CGO_ENABLED=1` and a working C toolchain to build it. If you want to avoid cgo entirely, use the pure-Go driver `modernc.org/sqlite` (registered under the name `"sqlite"`), which builds anywhere Go does.

Tip: The driver name string you pass to `sql.Open` (e.g., `"postgres"`, `"sqlite3"`, `"mysql"`) must match the name the driver registers in its `init()`. Check the driver's documentation if `sql.Open` returns `unknown driver` — you may be using the wrong name or missing the blank import.

`pgx` — The PostgreSQL Driver

For PostgreSQL specifically, `github.com/jackc/pgx/v5` is the dominant driver and most new Go projects use it directly rather than through `database/sql`. `pgx` exposes the full Postgres wire protocol — named parameters, `COPY`, `LISTEN/NOTIFY`, batch queries — none of which are available through the `database/sql` interface.

You can use `pgx` in two modes:

Mode 1: as a `database/sql` driver (drop-in, no API change):

```
import (
    "database/sql"
    _ "github.com/jackc/pgx/v5/stdlib" // registers "pgx" driver name
)

db, err := sql.Open("pgx", os.Getenv("DATABASE_URL"))
```

Mode 2: native `pgx` API (recommended for new Postgres projects):

```
import "github.com/jackc/pgx/v5/pgxpool"

pool, err := pgxpool.New(ctx, os.Getenv("DATABASE_URL"))
```

`pgxpool.Pool` is a connection pool with the same context-aware query methods as `database/sql` but with access to Postgres-specific features. `pgx` also provides `pgx.CollectRows` and `pgx.RowToStructByName` helpers that scan results directly into structs without manual `Scan` calls.

Tip: Prefer the native `pgx` API for new PostgreSQL projects. Use the `stdlib` wrapper only when you need to share code with a `database/sql`-based library (e.g., `sqlx`, `goose`).

Try It: A Small Music Store

Type this in and run it. It is a complete, self-contained program that opens an in-memory SQLite database, creates a table, inserts a few songs, and queries them back, demonstrating the core APIs from this chapter in one place. SQLite uses `?` placeholders, so every query below matches the driver it opens.

```
package main

import (
    "context"
    "database/sql"
    "fmt"
    "log"

    _ "github.com/mattn/go-sqlite3" // register the SQLite driver
)

type Song struct {
    ID     int
    Title  string
    Artist string
    Album  sql.Null[string]
}

func main() {
    db, err := sql.Open("sqlite3", ":memory:")
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()

    ctx := context.Background()

    if err := db.PingContext(ctx); err != nil { // verify connectivity
        log.Fatal(err)
    }

    _, err = db.ExecContext(ctx, `
        CREATE TABLE songs (
            id     INTEGER PRIMARY KEY AUTOINCREMENT,
            title  TEXT    NOT NULL,
            artist TEXT    NOT NULL,
            album  TEXT
        )`)
    if err != nil {
        log.Fatal(err)
    }

    // bulk insert using a prepared statement
```

```
stmt, err := db.PrepareContext(ctx,
    "INSERT INTO songs (title, artist, album) VALUES (?, ?, ?)")
if err != nil {
    log.Fatal(err)
}
defer stmt.Close()

inserts := []Song{
    {
        Title:  "Sounds of Slashdot",
        Artist: "San Mehat",
        Album:  sql.Null[string]{V: "DJ Essentials: Trance", Valid: true},
    },
    {
        Title:  "Gamemaster",
        Artist: "Matt Darey & Lost Tribe",
        Album:  sql.Null[string]{V: "DJ Essentials: Trance", Valid: true},
    },
    {
        Title:  "J'ai pas vingt ans !",
        Artist: "Alizée",
        Album:  sql.Null[string]{V: "Mes Courants Électriques...", Valid: true},
    },
    {
        Title:  "Gouryella",
        Artist: "Gouryella",
        Album:  sql.Null[string]{}, // NULL album
    },
}
for _, s := range inserts {
    if _, err := stmt.ExecContext(ctx, s.Title, s.Artist, s.Album); err != nil {
        log.Fatal(err)
    }
}

// query all songs for San Mehat and Matt Darey & Lost Tribe (DJ Essentials: Trance)
rows, err := db.QueryContext(ctx,
    "SELECT id, title, artist, album FROM songs WHERE album = ?", "DJ Essentials: Trance")
if err != nil {
    log.Fatal(err)
}
defer rows.Close()

for rows.Next() {
    var s Song
    if err := rows.Scan(&s.ID, &s.Title, &s.Artist, &s.Album); err != nil {
        log.Fatal(err)
    }
    album := "single"
    if s.Album.Valid {
        album = s.Album.V
    }
    fmt.Printf("%d: %s --- %s (%s)\n", s.ID, s.Title, s.Artist, album)
}
```

```
    if err := rows.Err(); err != nil {
        log.Fatal(err)
    }
}
```

Output:

```
1: Sounds of Slashdot --- San Mehat (DJ Essentials: Trance)
2: Gamemaster --- Matt Darey & Lost Tribe (DJ Essentials: Trance)
```

Once it runs, try these modifications:

- Wrap the four inserts in a transaction using `db.BeginTx` and the deferred rollback pattern, then commit at the end.
- Change the `WHERE album = ?` query to select every song and print `single` for the rows where `s.Album.Valid` is `false`.
- Delete the `db.PingContext` call and the table-creation step, then observe the exact error `QueryContext` returns when the `songs` table does not exist.

Key Points

- `sql.DB` is a **connection pool**, not a single connection; create one instance per database and share it across your application.
- Always use the context-aware methods — `QueryContext`, `QueryRowContext`, `ExecContext` — so that queries respect cancellation and timeouts (Chapter 12).
- `rows.Scan` maps column values into Go variables by position; always `defer rows.Close()` and check `rows.Err()` after the loop. [*no-discard-error*]
- `sql.ErrNoRows` is the sentinel error from `QueryRowContext` when no row matches; test with `errors.Is`.
- `sql.Null[T]` (Go 1.22) wraps nullable column values; `Valid` is `true` when the column is not `NULL`.
- `sql.Tx` groups statements into a transaction; the **deferred rollback pattern** guarantees cleanup on any error path.
- `db.PrepareContext` returns a `*sql.Stmt`; prepare once, execute many times.
- Driver packages (e.g., `github.com/lib/pq`) register themselves via `init()` and must be blank-imported to take effect (Chapter 1).
- For PostgreSQL, prefer `pgx/v5` (`github.com/jackc/pgx/v5`) over `lib/pq`; use the native `pgxpool.Pool` API for full Postgres feature access, or `pgx/v5/stdlib` for a `database/sql`-compatible drop-in.

Exercises

1. **Think about it:** JDBC requires explicit transaction management and connection pooling through a `DataSource`, usually provided by an application server or a library like HikariCP. Go's `database/sql` builds connection pooling directly into `sql.DB`. What are the tradeoffs of each approach? In what situations might you still want an external connection pool in a Go application?

2. **What does this print?**

```
package main

import (
    "database/sql"
    "fmt"
)

func main() {
    a := sql.Null[string]{V: "J'ai pas vingt ans !", Valid: true}
```

```
    b := sql.Null[string]{V: "Gouryella", Valid: false}
    c := sql.Null[int64]{V: 0, Valid: false}

    fmt.Println(a.Valid, a.V)
    fmt.Println(b.Valid, b.V)
    fmt.Println(c.Valid, c.V)
}
```

3. **Calculation:** Trace the following transaction sequence and state whether the database ends up with the row inserted or not, and why.

```
tx, _ := db.BeginTx(ctx, nil)
defer tx.Rollback()

_, err := tx.ExecContext(ctx, "INSERT INTO songs (title, artist) VALUES (?, ?)",
    "Sounds of Slashdot", "San Mehat")
if err != nil {
    return err
}

return tx.Commit()
```

   (Error handling on `BeginTx` is elided to keep the trace short.)

   Case A: `ExecContext` succeeds and `Commit` succeeds. Case B: `ExecContext` succeeds but `Commit` returns an error. Case C: `ExecContext` returns an error.

4. **Where is the bug?**

```
func getArtistSongs(ctx context.Context, db *sql.DB, artist string) ([]string, error) {
    rows, err := db.QueryContext(ctx,
        "SELECT title FROM songs WHERE artist = ?", artist)
    if err != nil {
        return nil, err
    }

    var titles []string
    for rows.Next() {
        var title string
        if err := rows.Scan(&title); err != nil {
            return nil, err
        }
        titles = append(titles, title)
    }
    return titles, nil
}
```

5. **Write a program:** Using `database/sql` and the `github.com/mattn/go-sqlite3` driver, write a program that:

   - Opens an in-memory SQLite database.
   - Creates a `playlists` table with columns `id` (integer primary key autoincrement), `name` (text, not null), and `owner` (text, not null).
   - Inserts at least three rows inside a single transaction using the deferred rollback pattern.
   - Queries and prints all rows using `QueryContext` and `Scan`.
   - Uses `sql.Null[string]` for at least one nullable column (add a `description` column that is nullable).

Chapter 18

Generics

Java programmers are no strangers to generics — you have been writing `List<T>` and `Map<K,V>` for years. Go added generics in version 1.18, and while the surface syntax looks familiar, the semantics differ in important ways: Go uses **monomorphization** rather than type erasure, introduces **constraints** as a first-class concept, and ships a constraint system powerful enough to express both simple and nuanced type requirements. This chapter covers type parameters, constraints, the tilde syntax, the standard packages that generics unlocked (`slices`, `maps`, `cmp`, `iter`, `unique`), and — critically — when you should reach for a concrete type instead.

Type Parameters

In Java, a generic method looks like:

```
public static <T, U> List<U> map(List<T> list, Function<T, U> f) { ... }
```

In Go, the type parameters move to a **type parameter list** in square brackets immediately after the function name:

```
func Map[T, U any](s []T, f func(T) U) []U {
    result := make([]U, len(s))
    for i, v := range s {
        result[i] = f(v)  // call f on each element and store the result
    }
    return result
}
```

`[T, U any]` declares two type parameters. `any` is the **constraint** — it means "T and U may be any type whatsoever." At the call site, the compiler usually infers the type arguments from the arguments you pass:

```
titles := []string{"Escape", "$100 Bills", "J'ai pas vingt ans !", "J'en ai marre !"}
lengths := Map(titles, func(s string) int { return len(s) })
fmt.Println(lengths) // [6 10 20 15]
```

You can supply them explicitly when inference fails:

```
lengths := Map[string, int](titles, func(s string) int { return len(s) })
```

Generic Types

Type parameters work on types as well as functions. A generic `Stack` is a classic example:

```
type Stack[T any] struct {
    items []T
```

```
}

func (s *Stack[T]) Push(v T) {
    s.items = append(s.items, v)  // append v to the top of the stack
}

func (s *Stack[T]) Pop() (T, bool) {
    if len(s.items) == 0 {
        var zero T              // zero value of T
        return zero, false
    }
    top := s.items[len(s.items)-1]
    s.items = s.items[:len(s.items)-1]
    return top, true
}
```

Instantiate it with a concrete type:

```
var playlist Stack[string]
playlist.Push("Escape")
playlist.Push("J'ai pas vingt ans !")
v, ok := playlist.Pop()
fmt.Println(v, ok) // J'ai pas vingt ans ! true
```

Tip: Unlike Java, Go generic types are instantiated with the concrete type written at the declaration site: `Stack[string]`, not `new Stack<>()`. There is no diamond operator because Go does not use constructors.

Constraints

A **constraint** is an interface that limits which types may be used as a type argument. Every type parameter has exactly one constraint. You cannot omit the constraint — if you want a type parameter to accept any type, you must write `any` explicitly; there is no implicit "unconstrained."

`any`

`any` is the least restrictive constraint: it permits every type. A type parameter constrained to `any` supports only the operations that every type supports — assignment and passing as a function argument. You cannot call methods on it, compare it with `==`, or use it as a map key.

`comparable`

`comparable` is a built-in constraint that permits any type that supports `==` and `!=`. It is the constraint required for map keys:

```
func Contains[T comparable](s []T, v T) bool {
    for _, elem := range s {
        if elem == v {  // only valid because T is comparable
            return true
        }
    }
    return false
}
```

Trap: `any` and `comparable` are not interchangeable. If you write `func NewCache[K any, V any]() map[K]V`, the compiler rejects it because map keys must be `comparable`. Change `K any` to `K comparable`.

Custom Constraint Interfaces

A constraint can be any interface. The interface may include method requirements, type union elements, or both.

A method constraint requires the type to have a specific method:

```
type Stringer interface {
    String() string  // the type must have a String() string method
}

func PrintAll[T Stringer](items []T) {
    for _, item := range items {
        fmt.Println(item.String())  // safe because T is constrained to Stringer
    }
}
```

A type union constraint lists the exact types permitted:

```
type Number interface {
    int | int32 | int64 | float32 | float64  // any of these five types
}

func Sum[T Number](s []T) T {
    var total T
    for _, v := range s {
        total += v  // + is defined for all types in Number
    }
    return total
}
```

You can combine union elements with method requirements in a single constraint interface, though this is uncommon in practice.

Tip: If you are hunting for Go's equivalent of Java's wildcards, stop — there isn't one. Java has use-site variance (`List<? extends Number>`, `List<? super Integer>`) and bounded type parameters (`<T extends Comparable<T>>`). Go has no wildcards and no variance at all, declaration-site or use-site. The constraint interface is the whole story: it plays the role that both bounded type parameters and wildcards play in Java. Where a Java API might take `List<? extends T>` to stay flexible about the element type, idiomatic Go just writes a generic function with a type parameter and a constraint, e.g. `func Sum[T Number](s []T) T`.

The Tilde Syntax

A type union like `int | float64` is too narrow: it excludes user-defined types whose **underlying type** is `int` or `float64`. Consider:

```
type BPM int
```

`BPM` is not `int` — it is a distinct named type. A function constrained to `int` will not accept `BPM`, even though `BPM` behaves exactly like an integer.

The **tilde** prefix `~T` means "any type whose underlying type is T":

```
type Numeric interface {
    ~int | ~int32 | ~int64 | ~float32 | ~float64
}

func Max[T Numeric](a, b T) T {
    if a > b {
        return a
    }
    return b
}
```

Now `Max` works with `BPM` as well as `int`:

```
type BPM int

a := BPM(128)
b := BPM(140)
fmt.Println(Max(a, b)) // 140
```

Tip: Standard library constraints always use `~T` rather than bare `T` in union elements. `cmp.Ordered` (see below) is defined with `~int | ~int8 | ... | ~string` so that user-defined types work automatically.

Wut: Java generics use **type erasure** — the type parameter is replaced by `Object` (or the bound) at compile time and checked only at the boundaries. At runtime, a `List<String>` and a `List<Integer>` are the same `List`. Go uses **monomorphization** — the compiler generates a distinct instantiation for each unique set of type arguments (or uses a shared representation for pointer-shaped types). The upshot: Go generics have no hidden boxing cost for value types like `int`, and there is no "unchecked cast" at runtime.

`comparable` vs `any` — Map Keys

Map keys must be `comparable`. If you write a generic function that creates or accepts a map, the key type parameter must be constrained to `comparable`.

```
// MapFromSlice builds a map from a slice of keys and a transform function.
func MapFromSlice[K comparable, V any](keys []K, f func(K) V) map[K]V {
    m := make(map[K]V, len(keys))
    for _, k := range keys {
        m[k] = f(k)  // build the map entry
    }
    return m
}

songs := []string{"Escape", "$100 Bills", "J'ai pas vingt ans !", "J'en ai marre !"}
lengths := MapFromSlice(songs, func(s string) int { return len(s) })
fmt.Println(lengths["J'ai pas vingt ans !"]) // 20
```

`any` is deliberately weaker than `comparable`. A `[]int` satisfies `any` but does not satisfy `comparable`. This distinction is enforced at compile time — you cannot accidentally use a non-comparable type as a map key.

The `slices` Package (Go 1.21)

Chapter 7 introduced the `slices` package. Now that you understand type parameters and constraints, the signatures make sense:

```
func Sort[S ~[]E, E cmp.Ordered](x S)                    // sort x in place; E must be ordered
func SortFunc[S ~[]E, E any](x S, cmp func(a, b E) int)  // sort using a custom comparator
func Contains[S ~[]E, E comparable](s S, v E) bool       // true if v appears in s
func Index[S ~[]E, E comparable](s S, v E) int           // first index of v, or -1
func Compact[S ~[]E, E comparable](s S) S                // remove consecutive duplicates
```

`S ~[]E` uses the tilde to accept any slice type whose element is `E`, including user-defined slice types like `type Playlist []string`. `E cmp.Ordered` requires elements that support `<`, `>`, and `==`. `E comparable` requires only `==`.

The iterator-based helpers landed later, in Go 1.23, once the `iter` package existed:

```
func Collect[E any](seq iter.Seq[E]) []E         // collect an iterator into a slice (Go 1.23)
func Sorted[E cmp.Ordered](seq iter.Seq[E]) []E  // collect and sort (Go 1.23)
```

Tip: `slices.Sort` works without a comparator because `cmp.Ordered` guarantees the `<` operator. `slices.SortFunc` works on any element type because it delegates ordering to the comparator you supply. Use `Sort` for simple value types, `SortFunc` for structs.

The `maps` Package (Go 1.21)

The `maps` package complements the maps you met in Chapter 7, just as `slices` does for slices. The full signatures reveal the constraints:

```
func Clone[M ~map[K]V, K comparable, V any](m M) M  // shallow copy of m (Go 1.21)
```

`Map ~map[K]V` uses the tilde to accept any named map type, not just `map[K]V` directly. `K comparable` is required because map keys must be comparable.

The iterator-returning `Keys` and `Values` are newer: the standard library originally exposed slice-returning versions in the experimental `golang.org/x/exp/maps`, but the versions promoted into the standard `maps` package in Go 1.23 return `iter.Seq` so they can be ranged over directly:

```
func Keys[Map ~map[K]V, K comparable, V any](m Map) iter.Seq[K]    // iterator over keys
func Values[Map ~map[K]V, K comparable, V any](m Map) iter.Seq[V]  // iterator over values
```

`Keys` and `Values` return iterators — see the `iter` section below.

The `cmp` Package (Go 1.21)

Chapter 7 introduced `cmp.Compare` and `cmp.Ordered`. The `cmp.Ordered` constraint is defined as:

```
// package cmp
type Ordered interface {
    ~int | ~int8 | ~int16 | ~int32 | ~int64 |
        ~uint | ~uint8 | ~uint16 | ~uint32 | ~uint64 | ~uintptr |
        ~float32 | ~float64 |
        ~string
}

func Compare[T Ordered](x, y T) int // returns -1, 0, or +1
func Less[T Ordered](x, y T) bool   // true if x < y
```

Every element in `Ordered` uses `~` so that user-defined types like `type BPM int` or `type Title string` satisfy the constraint automatically.

The `iter` Package (Go 1.23)

Go 1.23 introduced the `iter` package and **range-over-func** — the ability to `range` over a function value rather than a slice or map.

Iterator Types

```
// package iter
type Seq[V any] func(yield func(V) bool)        // single-value iterator
type Seq2[K, V any] func(yield func(K, V) bool) // two-value iterator (key and value)
```

An iterator is just a function that accepts a **yield callback**. The iterator calls `yield(v)` for each element. If `yield` returns `false`, the iteration should stop — this is how `break` works inside a `range` loop over an iterator.

Writing an Iterator

```
// ArtistTitles yields "Title (Artist)" strings for a slice of tracks.
func ArtistTitles(tracks []Track) iter.Seq[string] {
    return func(yield func(string) bool) {
        for _, t := range tracks {
            if !yield(t.Title + " (" + t.Artist + ")") {  // stop if yield returns false
                return
            }
        }
    }
}
```

Range Over Func

Once you have an `iter.Seq[V]` or `iter.Seq2[K,V]`, you can loop over it with `range`:

```
type Track struct {
    Title  string
    Artist string
}

func main() {
    tracks := []Track{
        {Title: "Escape",              Artist: "Jaroslav Beck"},
        {Title: "$100 Bills",          Artist: "Jaroslav Beck"},
        {Title: "J'ai pas vingt ans !", Artist: "Alizée"},
        {Title: "J'en ai marre !",     Artist: "Alizée"},
    }

    for s := range ArtistTitles(tracks) {
        fmt.Println(s)
    }
    // Escape (Jaroslav Beck)
    // $100 Bills (Jaroslav Beck)
    // J'ai pas vingt ans ! (Alizée)
```

```
    // J'en ai marre ! (Alizée)
}
```

`break` inside the loop causes `yield` to return `false`, which signals the iterator to stop.

`iter.Seq2`

`iter.Seq2[K,V]` is for iterators that yield two values, like index and element. `maps.Keys` and `maps.Values` return `iter.Seq[K]` and `iter.Seq[V]` respectively; `slices.All` (Go 1.23) returns `iter.Seq2[int, E]` yielding index–element pairs.

```
// package slices (Go 1.23)
func All[S ~[]E, E any](s S) iter.Seq2[int, E]  // yields (index, element) pairs
```

```
for i, title := range slices.All([]string{
    "Escape", "J'ai pas vingt ans !", "J'en ai marre !",
}) {
    fmt.Printf("%d: %s\n", i, title)
}
// 0: Escape
// 1: J'ai pas vingt ans !
// 2: J'en ai marre !
```

Tip: The `yield` function is the Go equivalent of Java's `Consumer<T>` in `Iterable.forEach`, but with a crucial difference: returning `false` from `yield` is how early termination (i.e., `break`) is communicated back to the iterator. Always check the return value of `yield` and return immediately when it is `false`.

The `unique` Package (Go 1.23)

The `unique` package provides **value interning** — it canonicalizes equal values so that identical values share the same memory address. This is useful for reducing memory pressure when the same strings (or other comparable values) appear many times.

```
// package unique
func Make[T comparable](v T) Handle[T]  // intern v; equal values return the same Handle

type Handle[T comparable] struct { /* opaque */ }

func (h Handle[T]) Value() T  // retrieve the interned value
```

Two `Handle` values are equal (via `==`) if and only if their underlying values are equal. This means handles can be used as map keys to efficiently deduplicate values:

```
import "unique"

func main() {
    a := unique.Make("Alizée")
    b := unique.Make("Alizée")
    c := unique.Make("Jaroslav Beck")

    fmt.Println(a == b) // true  --- same underlying string
    fmt.Println(a == c) // false --- different strings

    fmt.Println(a.Value()) // Alizée
}
```

Tip: `unique` is particularly useful for reducing allocations when a program repeatedly interns the same short strings (e.g., artist names or tag values from a high-volume stream). Handles are pointer-sized and comparable, so they can serve as efficient map keys in place of full string values.

Generic Type Aliases (Go 1.24)

Go 1.24 added support for **generic type aliases**. Before 1.24, a type alias could not have its own type parameters. Now it can:

```
// A Pair is an alias for a two-element struct.
type Pair[A, B any] = struct {
    First  A
    Second B
}

// A StringMap is an alias for a map with string keys.
type StringMap[V any] = map[string]V
```

Generic type aliases are useful when you want a shorter or domain-specific name for a parameterized type from another package:

```
type Result[T any] = struct {
    Value T
    Err   error
}
```

Wut: A generic type alias uses `=` in the definition, just like a non-generic alias. Without `=` you are defining a new named type, not an alias — the difference matters for method sets and assignability.

When NOT to Use Generics

Generics add expressive power, but they also add complexity. The Go team's guidance, reinforced by experience with the 1.18 release, is to prefer concrete types unless you genuinely need the abstraction.

Use generics when:

- You are writing a utility function that operates uniformly on multiple types with a shared structure (e.g., `Map`, `Filter`, `Reduce` over slices).
- You are building a container type (`Stack[T]`, `Queue[T]`) that does not care what it holds.
- You are writing library code that needs to work across a wide range of user-defined types.

Do not use generics when:

- The function only ever needs one concrete type. Write `func Sum(s []int) int`, not `func Sum[T ~int](s []T) T`.
- An `interface{}` / `any` parameter is simpler and the type is checked at runtime anyway (e.g., encoding libraries).
- The constraint is so complex that callers struggle to satisfy it.
- You are chasing "zero duplication" in application code where two concrete implementations are clearer than one generic one.

Trap: A common mistake when learning Go generics is over-constraining type parameters. If your function only calls `String()` on `T`, constrain `T` to `fmt.Stringer`, not to a union of every concrete type you think callers might use. Narrow constraints keep the function flexible; wide unions tie it to a fixed list.

Tip: Java generics are primarily a **type safety** mechanism — they prevent `ClassCastException` at runtime. Go generics serve the additional role of enabling **static dispatch**: the compiler can inline and optimize generic code in ways that `any` + type assertions cannot. But if you do not need the optimization and the types are few, a simple `interface` with a method set is often more idiomatic Go than a type-parameterized function.

Try It

Type this in and run it. It pulls together a generic `Set[T comparable]`, the iterator-based `maps.Keys`, and `slices.Sorted/slices.SortFunc` with `cmp.Compare` — the generics-powered standard library you met above, working as one program.

```
package main

import (
    "cmp"
    "fmt"
    "maps"
    "slices"
)

// Set is a generic set backed by a map; equal values are stored once.
type Set[T comparable] struct {
    m map[T]struct{}
}

func NewSet[T comparable]() *Set[T] {
    return &Set[T]{m: make(map[T]struct{})}
}

func (s *Set[T]) Add(v T) {
    s.m[v] = struct{}{}  // empty struct is a zero-byte placeholder
}

func (s *Set[T]) Contains(v T) bool {
    _, ok := s.m[v]
    return ok
}

func main() {
    plays := map[string]int{
        "Escape":     12,
        "$100 Bills": 7,
        "Legend":     3,
    }

    // maps.Keys returns an iter.Seq[string]; slices.Sorted collects and sorts it.
```

```
    titles := slices.Sorted(maps.Keys(plays))
    fmt.Println("titles:", titles)

    set := NewSet[string]()
    for _, t := range titles {
        set.Add(t)
    }
    set.Add("Escape")  // duplicate, silently ignored

    fmt.Println("has Escape:", set.Contains("Escape"))

    // slices.SortFunc with cmp.Compare orders entries by play count, descending.
    type entry struct {
        title string
        plays int
    }
    entries := make([]entry, 0, len(plays))
    for t, n := range plays {
        entries = append(entries, entry{t, n})
    }
    slices.SortFunc(entries, func(a, b entry) int {
        return cmp.Compare(b.plays, a.plays)
    })
    fmt.Println("most played:", entries[0].title)
}
```

It prints `titles: [$100 Bills Escape Legend]`, `has Escape: true`, and `most played: Escape`.

Try these modifications:

- Add a `Values() []T` method to `Set[T]` and a `Len() int` method, then print the set's size.
- Add a second `comparable` type parameter and make `Set` hold structs instead of strings.
- Swap `cmp.Compare(b.plays, a.plays)` for `cmp.Compare(a.plays, b.plays)` and see the order flip to ascending.

Key Points

- Type parameters are declared in square brackets after the function or type name: `func Foo[T any](...) ...`.
- Every type parameter requires a constraint; `any` permits all types, `comparable` permits types that support `==`.
- Constraint interfaces may contain method requirements, type union elements (`int | string`), or both.
- The tilde prefix `~T` means "any type whose underlying type is T" — essential for user-defined types like `type BPM int`.
- Map key type parameters must be constrained to `comparable`, not `any`.
- Go uses **monomorphization** (concrete code per instantiation), not type erasure; value types like `int` are never boxed.
- The `slices`, `maps`, and `cmp` packages arrived in Go 1.21 as the standard library's first-class use of generics; their `~T` constraints accept user-defined slice and map types. The iterator-based helpers (`slices.Collect`, `maps.Keys`, `maps.Values`) came later, in Go 1.23, once the `iter` package existed.
- `iter.Seq[V]` and `iter.Seq2[K,V]` (Go 1.23) are the iterator types; write an iterator by returning a function that calls a `yield` callback and checks its return value.
- `unique.Make[T]` (Go 1.23) interns comparable values; equal values share one canonical `Handle`.
- Generic type aliases (`type Alias[T any] = OtherType[T]`) are supported from Go 1.24.

- Prefer concrete types and interfaces over generics in application code; reach for generics when writing containers or utilities that must work uniformly across many types.

Exercises

1. **Think about it:** Java generics use **type erasure**: at runtime, `List<String>` and `List<Integer>` are both just `List`. Generic type information is only available at compile time. Go generics use **monomorphization** (or a shared pointer-shaped representation): the compiler may generate distinct code for each instantiation. Describe one concrete advantage and one concrete disadvantage of each approach. How does type erasure affect what you can do with a Java generic type at runtime (e.g., `instanceof List<String>`)? Does Go have the same limitation?

2. **What does this print?**

```go
package main

import "fmt"

func Filter[T any](s []T, keep func(T) bool) []T {
    var out []T
    for _, v := range s {
        if keep(v) {
            out = append(out, v)
        }
    }
    return out
}

type BPM int

func main() {
    beats := []BPM{72, 128, 96, 140, 80}
    fast := Filter(beats, func(b BPM) bool { return b >= 120 })
    fmt.Println(fast)

    words := []string{"Escape", "J'ai pas vingt ans !", "J'en ai marre !", "$100 Bills"}
    long := Filter(words, func(s string) bool { return len(s) > 7 })
    fmt.Println(long)
}
```

3. **Calculation:** A function with the signature `func Reduce[T, U any](s []T, init U, f func(U, T) U) U` folds a slice into a single value. Trace the execution of `Reduce([]int{1, 2, 3, 4}, 0, func(acc, v int) int { return acc + v })`. What is the concrete type bound to `T`? What is the concrete type bound to `U`? What value does the function return, and what are the intermediate values of `acc` after each call to `f`?

4. **Where is the bug?**

```go
package main

import "fmt"

type Playlist []string

func Dedupe[T any](s []T) []T {
    seen := make(map[T]bool)
```

```
    var out []T
    for _, v := range s {
        if !seen[v] {
            seen[v] = true
            out = append(out, v)
        }
    }
    return out
}

func main() {
    p := Playlist{
        "Escape", "J'ai pas vingt ans !", "Escape",
        "J'en ai marre !", "J'ai pas vingt ans !",
    }
    fmt.Println(Dedupe(p))
}
```

5. **Write a program:** Implement a generic `Set[T comparable]` type backed by a `map[T]struct{}`. It should support three methods: `Add(v T)` (add an element), `Contains(v T) bool` (membership test), and `Values() []T` (return all elements as a slice in any order). In `main`, create a `Set[string]`, add the four song titles "Escape", "$100 Bills", "J'ai pas vingt ans !", and "J'en ai marre !", add "J'ai pas vingt ans !" a second time, and print the length of the set and whether it contains "Escape" and "Legend".

Chapter 19

Testing

Go's `testing` package is deliberately minimal: no annotations, no test runner configuration files, no assertion library. What it lacks in ceremony it makes up for in power — table-driven tests, subtests, benchmarks, and a built-in fuzzer all ship with the standard library. In Java, testing is a stack of dependencies and configuration: you add JUnit to your `pom.xml` or `build.gradle`, wire up Surefire or the Gradle test task, and lean on annotations like `@Test`, `@BeforeEach`, and `@ParameterizedTest` to tell the runner what to do. In Go, testing is first-class tooling that is already installed: `go test` discovers any `TestXxx` function in a `_test.go` file and runs it, with no build-tool plugin, no runner config, and no annotations. The same toolchain you use to build also benchmarks, fuzzes, measures coverage, and detects data races — so writing a test is as cheap as writing the function it covers. If you are used to JUnit 5, most concepts will map cleanly; the idioms are just different.

The `testing.T` Type

Every test function takes a single argument of type `*testing.T`. The naming rule is strict: a test function must be named `TestXxx` where `Xxx` does not begin with a lowercase letter (an uppercase letter by convention), and it must live in a file whose name ends in `_test.go`.

```
package music

import "testing"

func TestBadApple(t *testing.T) {
    got := normalize("bad apple!!")
    want := "Bad Apple!!"
    if got != want {
        t.Errorf("normalize(%q) = %q, want %q", "bad apple!!", got, want)
    }
}
```

The test lives in `package music` — the same package as the code — so it can call the unexported `normalize`. A test file may instead declare `package music_test` (an *external* test package); it then sees only exported names, which keeps the test honest about the public API.

Go discovers and runs test files automatically with `go test`. There is no `@Test` annotation, no test class — just a naming convention.

`t.Error`, `t.Fatal`, and `t.Log`

The three methods you will reach for most often are:

```
func (t *T) Error(args ...any)                 // mark test failed; continue running
func (t *T) Errorf(format string, args ...any) // like Error, with a format string
func (t *T) Fatal(args ...any)                 // mark test failed; stop this test immediately
func (t *T) Fatalf(format string, args ...any) // like Fatal, with a format string
func (t *T) Log(args ...any)                   // log a message; shown only on failure or -v
func (t *T) Logf(format string, args ...any)   // record a formatted message
```

The key difference between `Error` and `Fatal` mirrors the difference between a recoverable and an unrecoverable condition. `t.Error` marks the test as failed but lets it keep running, which is useful when you want to report multiple independent failures in one pass. `t.Fatal` marks the test as failed and stops the current test function immediately.

Tip: Use `t.Fatal` when subsequent checks depend on a previous one passing. If you set up a database connection in a test and it fails, there is no point running the 20 assertions that follow — use `t.Fatal` to stop early. Use `t.Error` when each check is independent and you want a full picture of all failures.

Trap: `t.Fatal` calls `runtime.Goexit()` under the hood, which unwinds deferred functions in the current goroutine. If you call `t.Fatal` from a goroutine that is not the test's own goroutine, it will **not** stop the test — it will stop only that goroutine. Call `t.Fatal` only from the goroutine that received `t` directly (the test function itself or a helper called from it, not a spawned goroutine).

In Java with JUnit 5, you use `Assertions.assertEquals`, `Assertions.assertTrue`, and `Assertions.assertAll`. Go has no assertion library in the standard library. The idiomatic style is a plain `if` that calls `t.Error` or `t.Fatal`. Third-party libraries like `github.com/stretchr/testify/assert` exist, but many Go teams prefer the standard approach. Whichever you use, your failure messages should state the input, the actual result, and the expected result so that a reader can diagnose the failure without re-running the test. [*test-failure-describes-wrong*]

Table-Driven Tests

Table-driven tests are Go's answer to JUnit 5's `@ParameterizedTest`. Instead of writing one test function per case, you define a slice of structs — each struct is a test case — and loop over them. [*table-driven-tests*]

```
package music

import (
    "testing"
)

func TestBetterOffAlone(t *testing.T) {
    cases := []struct {
        name  string
        input int
        want  string
    }{
        {name: "zero",     input: 0,   want: "zero stars"},
        {name: "one star", input: 1,   want: "one star"},
        {name: "max",      input: 5,   want: "five stars"},
        {name: "negative", input: -1,  want: "zero stars"},
```

```
    }

    for _, tc := range cases {
        t.Run(tc.name, func(t *testing.T) {
            got := starRating(tc.input)
            if got != tc.want {
                t.Errorf("starRating(%d) = %q, want %q", tc.input, got, tc.want)
            }
        })
    }
}
```

The outer test function iterates over the cases and calls `t.Run` for each one. `t.Run` creates a **subtest**: a named, independently tracked test run.

`t.Run` Subtests

`t.Run(name, func(t *testing.T))` registers and runs a named subtest.

```
func (t *T) Run(name string, f func(t *T)) bool // run f as a named subtest; true if f passes
```

Each subtest:

- Has its own pass/fail state.
- Can be run individually with `-run TestFoo/subtest_name`.
- Appears in failure output as `TestFoo/subtest_name`, making it easy to identify which case failed.

Tip: Name your test cases with a `name` field and pass it as the first argument to `t.Run`. When a case fails, the output shows `--- FAIL: TestBetterOffAlone/negative (0.00s)` so you immediately know which row broke. Without subtests you would only see `--- FAIL: TestBetterOffAlone` and have to dig through the output to find which input caused it.

In JUnit 5, parameterized tests use `@ParameterizedTest` with `@MethodSource` or `@CsvSource`. Go's table-driven approach with `t.Run` is more explicit — you write a slice literal and a loop — but it gives you the same per-case naming and isolation. Format each `t.Errorf` call with the actual result before the expected result: `got %q, want %q`. [***actual-before-expected***]

`t.Helper()`

When you extract repeated assertion logic into a helper function, Go's test output points to the **helper** as the failure site rather than the call site in the test. That is rarely what you want. `t.Helper()` fixes this: calling it at the top of a helper function tells the testing framework to attribute any failure in that function to the **caller**.

Without `t.Helper()`:

```
// checkEqual reports whether got == want, but failure points to this line, not the caller.
func checkEqual(t *testing.T, got, want string) {
    if got != want {
        t.Errorf("got %q, want %q", got, want)  // file:line points here
    }
}
```

With `t.Helper()`:

```
// checkEqual reports whether got == want.
func checkEqual(t *testing.T, got, want string) {
    t.Helper()  // attribute failures to the caller, not this function
```

```
    if got != want {
        t.Errorf("got %q, want %q", got, want)  // file:line now points to the caller
    }
}
```

Now when `checkEqual` fails, the reported line is the line in your test function that called `checkEqual`, not the line inside `checkEqual` itself.

```
func TestCrazyTrain(t *testing.T) {
    checkEqual(t, normalize("crazy train"), "Crazy Train")                    // failure here
    checkEqual(t, normalize("THE SOUND OF SILENCE"), "The Sound of Silence") // failure here
}
```

Tip: Any function that calls `t.Error`, `t.Fatal`, `t.Log`, or another helper should call `t.Helper()` as its first statement. Forgetting `t.Helper` is a common oversight that makes failure output point to the wrong file and line. [***t-helper-for-helpers***]

`t.Cleanup`: Teardown the Idiomatic Way

JUnit 5 has `@AfterEach` to undo whatever `@BeforeEach` set up. Go's analog is `t.Cleanup`, which registers a function to run when the test (or subtest) finishes.

```
func (t *T) Cleanup(f func())  // register f to run when the test ends (LIFO order)
```

You register the cleanup right next to the code that needs it, so the setup and its teardown live together:

```
func TestPlaylistFile(t *testing.T) {
    f, err := os.CreateTemp("", "playlist-*.txt")
    if err != nil {
        t.Fatal(err)
    }
    t.Cleanup(func() {
        os.Remove(f.Name())  // runs when the test ends, pass or fail
    })

    // ... use f ...
}
```

Cleanups run in **last-in, first-out** order, just like deferred functions, and they run whether the test passes, fails, or calls `t.Fatal`.

Tip: `t.Cleanup` beats a bare `defer` for teardown that lives inside a helper. A `defer` in a helper fires when the *helper* returns, which is too early. `t.Cleanup` registered inside that helper fires when the *test* ends, so a `newTestServer(t)` helper can register its own shutdown and the caller never has to remember to close anything.

`t.Parallel`: Running Tests Concurrently

By default tests in a package run one after another. Calling `t.Parallel` signals that a test is safe to run alongside other parallel tests.

```
func (t *T) Parallel()  // mark this test to run in parallel with other parallel tests
```

It pairs naturally with `t.Run`: each subtest calls `t.Parallel` as its first statement, the runner pauses it, and once the loop finishes registering subtests it runs the paused ones together.

```go
func TestNormalizeParallel(t *testing.T) {
    titles := []string{"Monaco", "La Bachata", "Bad Apple!!"}
    for _, title := range titles {
        t.Run(title, func(t *testing.T) {
            t.Parallel()  // run this subtest concurrently with its siblings
            if normalize(title) == "" {
                t.Errorf("normalize(%q) was empty", title)
            }
        })
    }
}
```

Trap: Before Go 1.22, the loop variable was shared across iterations, so a parallel subtest that captured `title` would see the *last* value by the time it actually ran. Go 1.22 changed loop variables to be per-iteration, so the capture above is safe. If you target older toolchains, copy the variable inside the loop: `title := title`.

Benchmarks

A benchmark measures the performance of a piece of code. Benchmarks follow the same file convention as tests — `_test.go` — but the function name starts with `Benchmark` and the argument is `*testing.B`.

```go
func BenchmarkNormalize(b *testing.B) {
    for range b.N {
        normalize("bad apple!!")
    }
}
```

`b.N` is the number of iterations the framework chooses. The benchmark runner starts with a small `N` and increases it until the total run time is stable enough to report a reliable per-operation time. You do not set `b.N`; the framework sets it for you.

Run benchmarks with `-bench`:

```
go test -bench=. -benchmem ./...
```

The `-benchmem` flag adds allocation counts to the output.

`b.Loop`: the Modern Idiom

Go 1.24 added `b.Loop`, which is now the preferred way to write the benchmark loop.

```go
func (b *B) Loop() bool  // true until enough iterations have run; drives the benchmark loop
```

Instead of `for range b.N`, you write `for b.Loop()`:

```go
func BenchmarkNormalize(b *testing.B) {
    for b.Loop() {
        normalize("bad apple!!")
    }
}
```

`b.Loop` returns `true` until the framework has run enough iterations, then returns `false` to end the loop. It has two advantages over `b.N`. First, any setup that runs **before** the loop and any teardown **after** it are automatically excluded from the timing — you no longer need `b.ResetTimer` for the common case. Second, `b.Loop` keeps the benchmarked call alive so the compiler cannot optimize it away, a footgun that `b.N` loops sometimes hit.

```go
func BenchmarkTopTrack(b *testing.B) {
    data := buildLargePlaylist(10_000)  // setup outside the loop, not timed
    for b.Loop() {
        _ = findTopTrack(data)
    }
}
```

`b.N` still works and you will see it in older code, but reach for `b.Loop` in new benchmarks.

Tip: With `b.Loop`, put expensive setup before the loop and teardown after it; both are excluded from the measurement automatically. Only fall back to `b.ResetTimer` (next section) when you are still writing a `b.N`-style loop or need to reset the timer partway through.

`b.ResetTimer`

If your benchmark has expensive setup before the measured loop, use `b.ResetTimer()` to exclude the setup time from the measurement.

```go
func BenchmarkSoundOfSilence(b *testing.B) {
    data := buildLargePlaylist(10_000)  // expensive setup
    b.ResetTimer()                      // start timing only from here
    for range b.N {
        _ = findTopTrack(data)
    }
}
```

`b.ResetTimer` zeroes the elapsed time and allocation counters so that only the measured loop is included in the reported numbers.

Tip: Always call `b.ResetTimer()` after any setup that you do not want included in the benchmark. Without it, a slow setup inflates the per-operation time, making the benchmark misleading.

Trap: A plain `go test ./...` does not run benchmarks. Benchmarks only run when you pass `-bench=<pattern>`; without it, benchmark functions are compiled but never executed.

In JUnit 5, there is no built-in benchmarking; you need JMH or similar. Go ships a benchmarking harness in the standard library.

Fuzzing

Fuzzing automatically generates inputs to find crashes and panics. A fuzz test is a function named `FuzzXxx` that takes `*testing.F`.

```go
func FuzzBetterOffAlone(f *testing.F) {
    f.Add("Better Off Alone")   // seed corpus: start from known inputs
    f.Add("")
    f.Add("BAD APPLE!!")

    f.Fuzz(func(t *testing.T, s string) {
        result := normalize(s)
        if len(result) > 0 && result[0] >= 'a' && result[0] <= 'z' {
            t.Errorf("normalize(%q) starts with lowercase: %q", s, result)
```

```
        }
    })
}
```

`f.Add` seeds the fuzzer with known inputs. The fuzzer uses those seeds as a starting point and then mutates them to generate new inputs.

`f.Fuzz` registers the function that is called for each generated input. The first argument is always `*testing.T`; the remaining arguments match the types passed to `f.Add`.

Run fuzzing with `-fuzz`:

```
go test -fuzz=FuzzBetterOffAlone -fuzztime=30s .
```

Unlike `-run` and `-bench`, `-fuzz` accepts only a single package, so point it at one directory rather than `./...`. Without `-fuzz`, a fuzz test runs only the seed corpus — just like a regular test. With `-fuzz`, the engine runs indefinitely (or until `-fuzztime` elapses) mutating inputs until it finds a failure. When a failure is found, the engine writes the failing input to `testdata/fuzz/FuzzXxx/` so you can reproduce it.

Tip: Fuzz tests double as regression tests. The seed corpus (`f.Add` calls) runs every time `go test` runs — no `-fuzz` flag needed. Add the `testdata/fuzz/` directory to version control so that previously found failures are always checked.

Wut: Fuzzing is not available as a built-in in JUnit 5; you need a separate library. Go 1.18 added native fuzzing to the standard toolchain.

Example Tests

An example test is a function named `ExampleXxx` whose body ends in an `// Output:` comment. `go test` runs the function, captures what it prints to standard output, and fails if the captured output does not match the comment.

```
func ExampleNormalize() {
    fmt.Println(Normalize("bad apple!!"))
    // Output: Bad Apple!!
}
```

The payoff is double duty: the function is a verified test *and* it shows up in the package's generated documentation as runnable sample code. If someone changes `Normalize` so the output drifts, the example test fails like any other test — your docs cannot rot.

The name links the example to what it documents: `ExampleNormalize` attaches to the `Normalize` function, `ExampleClient_Get` to the `Get` method on `Client`, and a bare `Example` to the package itself. For output whose line order is not guaranteed (for example, ranging over a map), use `// Unordered output:` instead, which compares lines as a set.

Wut: An `ExampleXxx` function only runs if it has an `// Output:` (or `// Unordered output:`) comment. Without that comment, `go test` still compiles the example — so it must build — but never executes it, and it never fails.

JUnit 5 has no real analog here; Javadoc snippets are not executed or verified by the test runner.

Testing HTTP Handlers with `httptest`

Chapter 15 built HTTP handlers; the `net/http/httptest` package tests them without binding a real port or making you guess at a free one. There are two main styles.

The lightweight style uses `httptest.NewRecorder`, an `http.ResponseWriter` that records the status, headers, and body so you can assert on them. You call the handler directly — no network involved.

```
func NewRecorder() *httptest.ResponseRecorder                          // records the response
func NewRequest(method, target string, body io.Reader) *http.Request   // a request for tests
```

```
func greet(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hola, %s", r.URL.Query().Get("name"))
}

func TestGreet(t *testing.T) {
    req := httptest.NewRequest(http.MethodGet, "/?name=Mundo", nil)
    rec := httptest.NewRecorder()

    greet(rec, req)  // call the handler directly

    resp := rec.Result()
    if resp.StatusCode != http.StatusOK {
        t.Errorf("status = %d, want %d", resp.StatusCode, http.StatusOK)
    }
    body, _ := io.ReadAll(resp.Body)
    if string(body) != "Hola, Mundo" {
        t.Errorf("body = %q, want %q", body, "Hola, Mundo")
    }
}
```

The full-stack style uses `httptest.NewServer`, which starts a real HTTP server on a random local port and hands you its URL. This is the right tool when you need to exercise a client, middleware, or routing — anything that depends on a genuine round trip.

```
func NewServer(handler http.Handler) *httptest.Server // start a real server on a random port
```

```
func TestGreetServer(t *testing.T) {
    ts := httptest.NewServer(http.HandlerFunc(greet))
    defer ts.Close()  // shut the server down at the end

    resp, err := http.Get(ts.URL + "/?name=Mundo")
    if err != nil {
        t.Fatal(err)
    }
    defer resp.Body.Close()
    body, _ := io.ReadAll(resp.Body)
    if string(body) != "Hola, Mundo" {
        t.Errorf("body = %q, want %q", body, "Hola, Mundo")
    }
}
```

Tip: Prefer `httptest.NewRecorder` for unit-testing a single handler — it is faster and needs no port. Reach for `httptest.NewServer` only when something under test must make a real HTTP request, such as a client library or a reverse proxy.

Race Detector

Chapter 11 introduced the race detector briefly. Here is how to use it in tests.

Run `go test -race` to enable the race detector:

```
go test -race ./...
```

The race detector instruments the binary to track every memory access. If two goroutines access the same variable concurrently and at least one of them is a write, the detector prints a detailed report showing the goroutine stacks at both access sites.

Tip: Run `go test -race` in CI on every push. The race detector has a noticeable runtime cost (typically a 2–20x slowdown and 5–10x more memory), which is acceptable in CI but may be too slow for production. The cost of finding a race in production is far higher than the cost of running a slower test suite.

Trap: The race detector only catches races that **actually occur at runtime**. It cannot prove the absence of races. A test suite with low concurrency coverage might miss a race that the detector would catch under higher load. Write tests that exercise concurrent code paths.

Here is a concise example of a race the detector will catch:

```
func TestCounterRace(t *testing.T) {
    var count int
    var wg sync.WaitGroup

    for i := 0; i < 100; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            count++  // data race: concurrent unsynchronized write
        }()
    }
    wg.Wait()
    t.Log("count:", count)
}
```

This test imports `sync` for `sync.WaitGroup` (see Chapter 11) alongside `testing`. Run this with `go test -race` and the detector reports the race immediately.

Goroutine Leak Detection

Chapter 12 covered goroutine leaks in depth: what causes them, how to prevent them with context cancellation and `done` channels, and how `go.uber.org/goleak` detects them in tests. Every goroutine you spawn should have a clear exit path; if it has none, it is a leak. [*goroutine-must-exit*] The testing-specific summary: place `goleak.VerifyTestMain(m)` in `TestMain` to check every test in the package, or `defer goleak.VerifyNone(t)` in individual tests.

```
// Check all tests in the package --- preferred.
func TestMain(m *testing.M) {
    goleak.VerifyTestMain(m)
}

// Check a single test --- use when you cannot modify TestMain.
func TestBadAppleStream(t *testing.T) {
```

```
    defer goleak.VerifyNone(t)
    // ...
}
```

Tip: `TestMain` is Go's equivalent of JUnit 5's `@BeforeAll` / `@AfterAll` at the package level. It runs once per test binary rather than once per test function — the right place for global setup such as opening a shared database connection or installing goleak.

Wut: `goleak.VerifyTestMain` calls `m.Run()` internally. Do not call `m.Run()` yourself before passing `m` to it — the tests would run twice.

Integration Tests and Build Tags

Unit tests run quickly and need no external dependencies. Integration tests talk to real databases, real HTTP servers, or real message queues — they are slower and require infrastructure. Build tags let you separate the two so that `go test ./...` runs only the fast unit tests by default.

A build tag is a comment at the very top of a file, before the package declaration:

```
//go:build integration

package music_test
```

This file is excluded from the build unless the `integration` tag is provided. To run integration tests:

```
go test -tags=integration ./...
```

To run unit tests only:

```
go test ./...
```

Tip: Use separate CI pipeline stages: one stage runs `go test ./...` on every commit (fast, no infrastructure), and a second stage runs `go test -tags=integration ./...` before merging to main (slower, requires a test database or test container).

Wut: In Go 1.16 and earlier, build tags used a different syntax: `// +build integration` (a comment, not a `//go:build` directive). Go 1.17 introduced the `//go:build` form, which is now the standard. `gofmt` will add the `//go:build` form for you if you only write the old form. Both can coexist in the same file for backward compatibility.

Running Tests

The `go test` command is the entry point for all test-related tasks.

`go test ./...`

Run all tests in the current module:

```
go test ./...
```

The `./...` pattern matches the current directory and all subdirectories.

`-count=1` (Disable Caching)

Go caches test results. If the test sources and dependencies have not changed, `go test` reuses the cached result rather than re-running the tests. This is usually helpful, but sometimes you want to force a fresh run — for example, when testing code that depends on external state.

```
go test -count=1 ./...
```

`-count=1` tells the test runner to run each test exactly once and bypass the cache.

Tip: Use `-count=1` in CI to guarantee that tests always run, even when nothing has changed in the source. Cached test results in CI can hide flaky tests.

`-timeout`

By default `go test` applies a 10-minute timeout to the entire test binary. You can override this:

```
go test -timeout=30s ./...
```

Set a tight timeout in CI so that a hanging test (for example, a goroutine waiting on a channel that is never closed) fails fast rather than blocking your pipeline for 10 minutes.

`-run` and `-bench`

`-run` filters which test functions run by matching against a regular expression. `-bench` does the same for benchmarks.

```
go test -run=TestBetterOffAlone ./...           # run only tests matching "TestBetterOffAlone"
go test -run=TestBetterOffAlone/zero ./...      # run only the "zero" subtest
go test -bench=BenchmarkNormalize ./...     # run only this benchmark
go test -bench=. -benchmem ./...           # run all benchmarks with allocation stats
```

A Typical CI Invocation

A minimal, correct CI test command:

```
go test -race -count=1 -timeout=120s ./...
```

This runs all tests with the race detector, bypasses the cache, and fails if anything takes more than two minutes.

Try It

Time to type something in. The program below pairs a tiny `normalizeTitle` function with a table-driven test, a subtest per case, and a `t.Helper()` assertion — the three things you will use in almost every Go test you write. Save both files in the same directory, then run `go test -v` and watch each subtest report on its own line.

```
// normalize.go
package main

import "strings"

// normalizeTitle trims surrounding spaces and title-cases each word.
func normalizeTitle(s string) string {
    fields := strings.Fields(s)
```

```
    for i, w := range fields {
        fields[i] = strings.ToUpper(w[:1]) + strings.ToLower(w[1:])
    }
    return strings.Join(fields, " ")
}

func main() {}
```

```
// normalize_test.go
package main

import "testing"

// checkTitle reports a mismatch at the caller's line, not this helper's.
func checkTitle(t *testing.T, got, want string) {
    t.Helper()
    if got != want {
        t.Errorf("got %q, want %q", got, want)
    }
}

func TestNormalizeTitle(t *testing.T) {
    cases := []struct {
        name  string
        input string
        want  string
    }{
        {name: "padded",     input: "  paint the town red  ", want: "Paint The Town Red"},
        {name: "all caps",   input: "MONACO",                 want: "Monaco"},
        {name: "mixed case", input: "la BACHATA",             want: "La Bachata"},
        {name: "empty",      input: "",                       want: ""},
    }

    for _, tc := range cases {
        t.Run(tc.name, func(t *testing.T) {
            checkTitle(t, normalizeTitle(tc.input), tc.want)
        })
    }
}
```

Running `go test -v` reports `PASS` for `TestNormalizeTitle` and each named subtest below it. Then try these modifications:

- Add a case that you expect to fail (say, `want: "monaco"`) and confirm the failure output points to the `t.Run` line, not inside `checkTitle` — that is `t.Helper()` at work.
- Add a `BenchmarkNormalizeTitle` and run `go test -bench=. -benchmem` to see ns/op and allocation counts.
- Delete the `t.Helper()` line and re-run the failing case to see the reported line number move into the helper.

Key Points

- Test functions are named `TestXxx` and live in `_test.go` files; no annotations required.
- `t.Error` marks a failure and continues; `t.Fatal` marks a failure and stops the test immediately.
- Table-driven tests use a slice of anonymous structs; `t.Run` creates named subtests for each case.

- `t.Helper()` must be called at the top of any helper function so that failure output points to the call site, not the helper.
- `t.Cleanup(f)` registers teardown that runs when the test ends (LIFO), the idiomatic analog of JUnit's `@AfterEach`; `t.Parallel()` marks a (sub)test to run concurrently.
- Benchmarks are named `BenchmarkXxx` and receive `*testing.B`; prefer the `for b.Loop() { ... }` idiom (Go 1.24), which excludes setup and teardown from timing automatically. `b.N` still works.
- Call `b.ResetTimer()` after setup when you are still writing a `b.N`-style loop.
- Example tests (`ExampleXxx` with a trailing `// Output:` comment) are verified, runnable documentation.
- Test HTTP handlers with `httptest.NewRecorder` (call the handler directly) or `httptest.NewServer` (a real server on a random port).
- Fuzz tests are named `FuzzXxx` and receive `*testing.F`; `f.Add` seeds the corpus, `f.Fuzz` registers the test body.
- Without `-fuzz`, a fuzz test runs only the seed corpus, acting as a standard test.
- Run `go test -race` in CI on every push to catch data races; the race detector only reports races that actually occur.
- `goleak.VerifyTestMain(m)` in `TestMain` detects goroutine leaks after the test suite finishes.
- Use `//go:build integration` to gate slow integration tests; run them with `-tags=integration`.
- `go test ./...` runs all tests; `-count=1` disables caching; `-timeout` caps the run time.

Exercises

1. **Think about it:** JUnit 5's `@ParameterizedTest` with `@CsvSource` and Go's table-driven tests with `t.Run` both let you run the same logic against many inputs. Describe two concrete advantages that Go's table-driven approach gives you over `@CsvSource`. Then explain the key behavioral difference between `t.Fatal` and `t.Error` inside a subtest, and describe a scenario where you would deliberately choose `t.Error` over `t.Fatal`.

2. **What does this print?** Trace the output when this test is run with `go test -v`.

```go
package music_test

import "testing"

func checkPositive(t *testing.T, n int) {
    if n <= 0 {
        t.Errorf("expected positive, got %d", n)
    }
}

func TestSoundOfSilence(t *testing.T) {
    checkPositive(t, 1)
    t.Log("checked 1")
    checkPositive(t, -1)
    t.Log("checked -1")
    checkPositive(t, 2)
    t.Log("checked 2")
}
```

3. **Calculation:** A benchmark function has the following structure:

```go
func BenchmarkCrazyTrain(b *testing.B) {
    for range b.N {
        _ = processTrack("Crazy Train")
    }
}
```

On the first probe the framework sets `b.N = 1` and measures elapsed time. It then sets `b.N = 100`, then `b.N = 10_000`, then `b.N = 1_000_000`. The framework stops when the total elapsed time exceeds one second. If `processTrack` takes exactly 2 µs per call, at which value of `b.N` does the total elapsed time first exceed one second? What is the reported ns/op value?

4. **Where is the bug?** The following test helper is supposed to make failure output point to the call site in `TestBadApple`, but it does not. Identify the bug and show the fix.

```
package music

import "testing"

func assertNormalized(t *testing.T, input, want string) {
    got := normalize(input)
    if got != want {
        t.Fatalf("normalize(%q): got %q, want %q", input, got, want)
    }
}

func TestBadApple(t *testing.T) {
    assertNormalized(t, "bad apple!!", "Bad Apple!!")
    assertNormalized(t, "better off alone", "Better Off Alone")
}
```

5. **Write a program:** Write a table-driven test for the following function. Your test must include at least five cases covering normal input, empty string, all-caps input, and a multi-word title. Use `t.Run` for each case and `t.Helper` in any helper you write.

```
// TitleCase converts a string to title case.
// Each word's first letter is uppercased; the rest are lowercased.
// Words are separated by spaces.
func TitleCase(s string) string
```

Appendix A

Go Proverbs

Rob Pike delivered a talk called *Go Proverbs* at Gopherfest in November 2015. He listed a set of short, pithy sayings that capture the philosophy and idioms of Go — not language rules, but ways of thinking that experienced Go programmers internalize over time. If you search for "Rob Pike Go Proverbs" you will find the original talk and the companion site at go-proverbs.github.io.

The proverbs are memorable precisely because they are terse. Each one encodes a design decision, a tradeoff, or a lesson learned from years of building large systems. For Java programmers, several of them describe habits you need to actively unlearn. This appendix takes the proverbs most relevant to a Java programmer, unpacks what each means in practice, and connects it explicitly to the Java thinking it pushes back against.

Don't communicate by sharing memory; share memory by communicating

This is the most famous Go proverb and the one that most directly contradicts the Java approach to concurrency.

In Java, the default concurrency model is shared mutable state. Two threads access the same `ArrayList` or the same field, and you protect it with `synchronized`, `ReentrantLock`, or a `volatile` field. The data lives in one place; the threads fight to access it; the locks prevent corruption.

Go's preferred model flips this around. Instead of giving both goroutines a reference to the same piece of data and then locking it, you give the data to one goroutine and send it through a channel when another goroutine needs it. At any moment, only one goroutine owns the data — ownership transfers through communication, not through locking.

```
// Java-style thinking in Go: don't do this
var counter int
var mu sync.Mutex

func increment() {
    mu.Lock()
    counter++
    mu.Unlock()
}

// Go-style thinking: send the work, not the shared state
jobs := make(chan int, 100)
done := make(chan struct{})
```

```go
go func() {
    total := 0
    for n := range jobs {
        total += n
    }
    fmt.Println(total)
    close(done)
}()

for i := 0; i < 100; i++ {
    jobs <- i
}
close(jobs)
<-done // wait for the consumer to finish before exiting
```

The second version has no lock, no shared variable visible to multiple goroutines, and no data race. The accumulator `total` is private to one goroutine; the channel is the communication mechanism.

Tip: The Go memory model guarantees that a send on a channel *happens before* the corresponding receive. This means the channel itself acts as synchronization — you do not need a separate lock when you use channels correctly.

Trap: This proverb does not mean "never use `sync.Mutex`." Mutexes are the right tool when a small critical section protects a shared counter or cache. The proverb is about *default thinking*: reach for channels first when goroutines need to coordinate.

Concurrency is not parallelism

Java programmers often use "concurrent" and "parallel" as synonyms. Go draws a sharp distinction.

Concurrency is a property of program structure: you decompose a problem into independent pieces that *could* run simultaneously. **Parallelism** is a property of execution: those pieces *actually* run simultaneously on multiple CPU cores.

A concurrent program written for a single-core machine will not run in parallel, but it is still concurrent — the pieces are structured to be independent. Conversely, you can run a non-concurrent program on many cores and get no benefit.

Tip: Rob Pike's memorable formulation is: "Concurrency is about *dealing with* lots of things at once. Parallelism is about *doing* lots of things at once." Write concurrent code first. The runtime decides how much parallelism you get based on `GOMAXPROCS`.

Goroutines give you concurrency cheaply. How much parallelism you get is controlled by `runtime.GOMAXPROCS`, which defaults to the number of CPU cores. Since Go 1.25 it also respects the container's cgroup CPU limit, so a process pinned to two cores in a container no longer spins up a goroutine scheduler sized for the whole host. You almost never set it manually — the default is right.

The practical lesson for Java programmers: the goal of channels and goroutines is clean problem decomposition, not raw throughput. A pipeline of three goroutines connected by channels is concurrent design even if it runs on one core.

The bigger the interface, the weaker the abstraction

In Java, interfaces commonly carry dozens of methods. `java.util.List` declares roughly two dozen methods; `java.util.Collection` more than a dozen. Implementing them requires either a large amount of code or inheriting a skeletal implementation.

Go's standard library tells a different story. `io.Reader` has one method. `io.Writer` has one method. `fmt.Stringer` has one method. `error` has one method.

A one-method interface describes exactly one capability. Anything that satisfies it — a file, a network connection, a byte buffer, a custom type — can be used wherever that interface is expected. The abstraction is maximally general.

When an interface grows to ten methods, only a much narrower set of types can satisfy it. The interface describes a specific *kind of thing* rather than a specific *capability*. It has become less of an abstraction and more of a blueprint for a single concrete type.

```
// Weak abstraction: only one type will ever implement this
type DatabaseService interface {
    Connect(dsn string) error
    Query(sql string, args ...any) (*Rows, error)
    Exec(sql string, args ...any) (Result, error)
    Begin() (*Tx, error)
    Rollback() error
    Commit() error
    Close() error
    Ping() error
    Stats() DBStats
}

// Strong abstraction: anything that can store a byte slice
type Storer interface {
    Store(key string, value []byte) error
}
```

Tip: When designing a Go interface, ask: "What is the *minimum* set of methods a caller actually needs?" Often the answer is one or two. Define the small interface at the call site rather than defining a large interface in the package that owns the implementation.

Accept interfaces, return concrete types

This one is not actually from Pike's *Go Proverbs* talk — it is a closely related community idiom that gets repeated alongside the canonical proverbs, and it fits right in here. It describes the idiomatic direction of abstraction in Go function signatures.

Accept interfaces because it makes your function flexible. A function that accepts `io.Reader` works with a file, a byte buffer, a network socket, a gzip stream, or a test mock — anything that implements `Read`. The caller is not locked into a specific type.

Return concrete types because it gives the caller maximum information. If your function returns `*os.File`, the caller can call `Stat`, `Seek`, `Sync`, and `Close`. If you return `io.ReadCloser`, the caller can only `Read` and `Close`. Returning an interface hides capability the caller might need.

```
// Good: accepts interface (flexible), returns concrete type (informative)
func NewLoggedReader(r io.Reader) *LoggedReader {
    return &LoggedReader{r: r, count: 0}
```

```
}

// Avoid: returning an interface when the concrete type is more useful
func NewLoggedReader(r io.Reader) io.Reader { // loses access to Count()
    return &LoggedReader{r: r, count: 0}
}
```

Trap: Java programmers coming from "program to an interface, not an implementation" often want to return interface types everywhere. In Go that habit hides the concrete type's full API from callers. Return the concrete type; let the caller decide which interface to assign it to.

The exception is the `error` interface: always return `error`, not a concrete error type, so callers use `errors.Is` and `errors.As` instead of type-asserting against a specific struct.

Make the zero value useful

In Java, objects do not exist until you call a constructor. Using an uninitialized field (null) causes a `NullPointerException`. Java programmers habitually write defensive constructors and factory methods to ensure objects are always in a valid state.

In Go, every type has a **zero value** — the value you get when you declare a variable without initializing it. Numeric types zero to `0`, booleans to `false`, pointers to `nil`, strings to `""`, and struct fields to their respective zero values.

The proverb asks you to design your types so the zero value is *already useful*, not broken.

```
var b bytes.Buffer        // zero value: empty buffer, ready to use
b.WriteString("Sandstorm") // no constructor needed

var mu sync.Mutex         // zero value: unlocked mutex, ready to use
mu.Lock()

var wg sync.WaitGroup     // zero value: counter at zero, ready to use
wg.Add(1)
```

None of these required a constructor call. `bytes.Buffer`, `sync.Mutex`, and `sync.WaitGroup` are all designed so the zero value is fully operational.

Tip: When designing a new type, ask: "Is the zero value of this struct a sensible default state?" If you can arrange for it to be, callers get a simpler API — no constructor required, and no possibility of an "uninitialized" state.

Trap: Pointers in Go zero to `nil`, just like Java references. Calling a method on a nil pointer panics. If your type relies on a pointer field being non-nil, either initialize it lazily on first use or document that the zero value is not usable and require a constructor.

The empty interface says nothing

In Go, `interface{}` (written `any` since Go 1.18) is an interface with no methods. Every type satisfies it. A function that accepts `any` can receive literally anything — an `int`, a `string`, a goroutine channel, a function value, a `nil`.

This flexibility is a warning sign, not a feature. If you accept `any`, you have told the compiler nothing about what you intend to do with the value. You will need a type assertion or a type switch to extract something

useful, and that work happens at runtime with no compile-time safety.

```
// Loses type safety; requires runtime type assertions
func PrintValue(v any) {
    switch x := v.(type) {
    case int:
        fmt.Println("int:", x)
    case string:
        fmt.Println("string:", x)
    default:
        fmt.Printf("unknown: %T\n", v)
    }
}

// Better: use a specific interface or generics (Go 1.18+)
func PrintStringer(v fmt.Stringer) {
    fmt.Println(v.String())
}
```

Trap: Java programmers who are used to `Object` as the universal base class reach for `any` when they want "any type." In Go that is rarely the right answer. Use a concrete type, a narrow interface, or a generic type parameter instead. Reserve `any` for truly generic infrastructure like JSON serialization or logging frameworks where the type is genuinely unknown.

Errors are values

In Java, error signaling means throwing an exception — an interruption of the normal control flow that unwinds the call stack. The exception is caught somewhere above and the normal path and the error path are syntactically separate (`try` vs `catch`).

In Go, a function that can fail returns an `error` as an ordinary return value. There is no stack unwinding, no separate exception handling syntax, and no checked-versus-unchecked distinction. The error is just a value you assign to a variable and inspect with `if err != nil`.

Because errors are values, you can do everything with them that you can do with any other Go value: store them in a slice, pass them to a function, wrap them, compare them, build pipelines around them.

```
// Errors as values enable this pattern: chain calls, accumulate the first error
type Writer struct {
    w   io.Writer
    err error
}

func (ew *Writer) write(p []byte) {
    if ew.err != nil {
        return
    }
    _, ew.err = ew.w.Write(p)
}

// Callers write without checking each time; check once at the end
ew := &Writer{w: os.Stdout}
ew.write(header)
ew.write(body)
ew.write(footer)
```

```go
if ew.err != nil {
    log.Fatal(ew.err)
}
```

Tip: The Go blog post "Errors are Values" by Rob Pike (2015) shows this pattern in detail and demonstrates that "if err != nil" repetition is not inherent — it is a consequence of how you structure the code.

Don't just check errors, handle them gracefully

This proverb follows directly from the previous one. Because errors are values, you are responsible for deciding what to do with them.

Java programmers sometimes carry over two habits that do not translate well to Go:

1. Catching an exception and logging it before re-throwing, which adds noise without adding information.
2. Swallowing an error silently (catching `Exception` and doing nothing).

In Go, "handle gracefully" means: add context, decide whether to retry or abort, clean up resources, and propagate with enough information for the caller to make a decision.

```go
// Bad: ignore the error
data, _ := os.ReadFile("config.json")

// Bad: check but do nothing useful
data, err := os.ReadFile("config.json")
if err != nil {
    fmt.Println(err) // log and fall through; data is nil
}

// Good: add context, propagate clearly
data, err := os.ReadFile("config.json")
if err != nil {
    return fmt.Errorf("loading config: %w", err)
}
```

Wrapping with `%w` preserves the original error so callers can use `errors.Is` or `errors.As` to inspect it. Each layer adds its own context to the chain without hiding what happened.

Trap: Do not log an error *and* return it. That causes the same error to be logged multiple times at different layers. Either handle the error fully at one level (log it and stop propagating) or wrap and return it for the caller to handle.

A little copying is better than a little dependency

In the Java ecosystem, adding a Maven or Gradle dependency is almost zero-friction. The build system downloads transitive dependencies automatically, and the culture encourages reuse at the package level. The result is dependency trees hundreds of packages deep for trivial utilities.

Go takes a more cautious position. The module system (`go.mod`) makes dependencies explicit, and the community norm is to pull in a dependency only when the benefit clearly outweighs the cost of tracking it, auditing it for security, and waiting for it to update.

When you need ten lines of utility code from a package, consider copying those ten lines instead of adding the whole package as a dependency. The copied code is stable, auditable, and has no surprise transitive

dependencies.

```
// Instead of importing a string utilities package for one function,
// copy the five-line helper you actually need:

// containsAny reports whether s contains any Unicode code point in chars.
func containsAny(s, chars string) bool {
    for _, c := range chars {
        if strings.ContainsRune(s, c) {
            return true
        }
    }
    return false
}
```

Tip: The Go standard library is intentionally rich so that most common tasks do not require third-party dependencies. Before adding a dependency, scan `strings`, `bytes`, `slices`, `maps`, `sort`, `strconv`, and `unicode` — the answer is often already there.

Wut: This proverb applies to *small* utilities. For a full TLS implementation, a database driver, or an HTTP client framework, copying is obviously not the answer. The proverb targets the temptation to add a 50-package dependency to get one helper function.

Clear is better than clever

Go was designed for large teams and long-lived codebases. The person who reads your code in two years might be you after you have forgotten what it does, or a colleague who joined the project last week.

"Clever" code — one-liners that pack multiple side effects into a single expression, or intricate use of the type system to eliminate a few lines of boilerplate — imposes a cognitive tax on every future reader. "Clear" code says what it does, even if it takes a few extra lines.

```
// Clever: builds a throwaway map and indexes it in one expression
x := map[string]int{"a": 1, "b": 2}["b"]

// Clear: two lines, obvious intent
m := map[string]int{"a": 1, "b": 2}
x := m["b"]
```

Tip: Go's `gofmt` enforces a uniform style so that no one spends mental energy on formatting. The remaining style decisions — naming, structure, abstraction level — should all favor the reader. If you find yourself writing a comment to explain what a line of code does, consider rewriting the code so the comment is unnecessary.

Java programmers who have spent time with lambda chains, streams, and optional-chaining sometimes find Go's explicit loops and `if err != nil` checks verbose. Over time most find the verbosity pays for itself in debuggability and readability.

gofmt's style is no one's favorite, yet gofmt is everyone's favorite

`gofmt` is Go's canonical formatter. It rewrites your source code to match the one true style: tab indentation, specific brace placement, aligned spacing, consistent blank lines. You have no configuration options.

No one gets exactly the style they would choose if left to their own preferences — that is the point. Because the style is universal and non-negotiable, no one argues about it. Code review time is not spent on formatting. Merge conflicts over whitespace disappear. Every Go file in every repository in the world looks structurally the same.

```
// You write:
func add(a int,b int)int{return a+b}

// gofmt produces:
func add(a int, b int) int { return a + b }
```

Tip: Run `gofmt -w .` before committing, or configure your editor to run it on save. Most Go projects enforce `gofmt` compliance in CI and reject code that is not formatted. The `goimports` tool does everything `gofmt` does and also manages import blocks.

Java programmers accustomed to negotiating Checkstyle or Google Java Format rules with their team will appreciate that in Go the negotiation never happens.

Cgo is not Go

Cgo is the mechanism that lets Go code call C libraries. It is powerful and sometimes necessary — for system-level APIs or existing C codebases — but it comes with a significant cost.

Code that uses Cgo does not compile with the same toolchain as pure Go code. Cross-compilation becomes complicated. Build times increase. The garbage collector and the C memory allocator coexist awkwardly. Goroutine stacks cannot grow through Cgo frames. Debugging is harder.

The proverb is a reminder that reaching for Cgo means stepping outside the Go ecosystem and accepting all of its constraints. For most application code, a pure Go alternative exists.

Tip: Java programmers are familiar with JNI, which carries similar complexity costs. The Go community's attitude toward Cgo mirrors the Java community's attitude toward JNI: use it when you must, avoid it when you can.

Cgo must always be guarded with build constraints

(The proverb is originally stated as "Cgo must always be guarded with build tags"; the mechanism is the same thing modern Go calls a build constraint.)

When you do use Cgo, you must tell the Go build system which platforms your C code supports. Without build constraints, your package will fail to compile on any platform where the C toolchain is absent or where the C code does not compile.

Build constraints live at the top of the file:

```
//go:build linux && amd64

package mypackage

// #include <sys/mman.h>
import "C"
```

This constraint restricts the file to Linux on AMD64. The rest of your package can still be compiled everywhere; only this file is gated.

Tip: Use `//go:build` (the modern syntax, Go 1.17+) rather than the older `// +build` comment. Run `go build ./...` for a representative set of target platforms in CI to catch constraint gaps early.

Syscall must always be guarded with build constraints

(As with Cgo, the proverb is originally phrased "Syscall must always be guarded with build tags"; "build constraint" is the modern term for the same mechanism.)

The `syscall` package exposes operating-system system calls directly. System calls are platform-specific: the numbers, arguments, and available calls differ between Linux, macOS, Windows, and other platforms.

A file that calls `syscall.Mmap` or `syscall.Kill` will fail to compile on a platform where that call does not exist or has a different signature. Build constraints ensure the file is only compiled on the platforms it supports.

```
//go:build linux || darwin

package fileutil

import "syscall"

func lockFile(fd uintptr) error {
    return syscall.Flock(int(fd), syscall.LOCK_EX)
}
```

Trap: Java programmers are used to the JVM abstracting away platform differences. In Go, once you use `syscall` or `golang.org/x/sys`, you own the platform compatibility problem. Prefer higher-level standard library packages (`os`, `net`, `io`) that handle platform differences internally.

With the unsafe package there are no guarantees

The `unsafe` package lets you escape Go's type system: convert between pointer types, read the size of a value, perform pointer arithmetic. Using it bypasses the memory safety guarantees that make Go programs reliable.

There is no specification for what `unsafe` code will do across Go versions, architectures, or runtime implementations. Code that works today may break on the next Go release, on a different OS, or with a different garbage collector.

```
import "unsafe"

// This compiles, but violates Go's memory model.
// The garbage collector may move objects; this pointer may become invalid.
p := (*int)(unsafe.Pointer(uintptr(unsafe.Pointer(&x)) + unsafe.Sizeof(x)))
```

Trap: Java programmers who have used `sun.misc.Unsafe` know this territory. The pattern is the same: you get power, you lose guarantees, and you own all the consequences. In Go, `unsafe` is almost never needed in application code. The standard library and well-maintained packages handle the rare cases — binary serialization, memory-mapped I/O — where direct memory access is genuinely required.

Reflection is never clear

The `reflect` package lets you inspect and manipulate values at runtime without knowing their types at compile time. It powers `encoding/json`, `fmt`, and `text/template`. It is essential infrastructure. It is also notoriously difficult to read and debug.

Reflection code trades compile-time type checking for runtime flexibility. Errors that a compiler would catch become panics at runtime. The code path through a reflective call is opaque to the reader.

```
// Clear: the compiler knows the type, the reader knows the type
n := len(mySlice)

// Unclear: works, but why? what type is v? what if Kind() is wrong?
v := reflect.ValueOf(mySlice)
n := v.Len()
```

Tip: Before reaching for `reflect`, check whether generics (Go 1.18+) solve your problem. A generic function preserves type safety and is readable. Reflection is appropriate for cases where the types are genuinely unknown at compile time — serialization, testing frameworks, dependency injection containers.

Trap: Java programmers with a Spring or Hibernate background are accustomed to frameworks that do heavy annotation processing and reflection under the hood. That is fine in frameworks you use but do not read. In Go application code, reflective paths are yours to own and debug. Keep them small, well-tested, and isolated from the rest of the codebase.

Appendix B

Tooling

Go ships with an unusually complete set of first-party tools, and the ecosystem adds a few more that every Go developer uses daily. This appendix is a quick reference for all of them.

gofmt and goimports

Go formatting is not a style guide suggestion — it is enforced. `gofmt` is the official formatter, and idiomatic Go code is always `gofmt`-clean. There is no equivalent debate in the Go community about brace placement or indentation because `gofmt` makes every decision for you.

The Java world has tools like Checkstyle and Google Java Format, but they are optional and configurable. `gofmt` has almost no configuration on purpose: one canonical style for all Go code everywhere.

Run the formatter in place:

```
gofmt -w .          # rewrite all .go files under the current directory
gofmt -l .          # list files that differ from the canonical format
```

`goimports` extends `gofmt` by also adding missing imports and removing unused ones. It is a strict superset of `gofmt` and is preferred in editor integrations.

```
goimports -w .      # format and fix imports in place
```

Install it once:

```
go install golang.org/x/tools/cmd/goimports@latest
```

Tip: Configure your editor to run `goimports -w` on every save. In VS Code this is the default when the Go extension is installed. In IntelliJ IDEA with the Go plugin, enable **Reformat code on save** and set the formatter to `goimports`. You should never need to run either tool manually once your editor is configured.

Trap: Submitting un-formatted code in a Go project is a red flag. Most Go CI pipelines run `gofmt -l .` and fail the build if any file differs. Don't rely on reviewers to catch it.

go vet

`go vet` is a static analysis tool that catches mistakes the compiler deliberately ignores. The compiler only rejects code that is syntactically or type-invalid; `go vet` catches things that compile fine but are almost

certainly bugs.

Java programmers will recognize this role from SpotBugs or ErrorProne, but `go vet` is built in and requires no configuration.

```
go vet ./...        # vet all packages in the module
```

Common mistakes `go vet` catches:

- `Printf`-family format string mismatches (`fmt.Printf("%d", "hello")`)
- Passing a mutex by value (copying a `sync.Mutex` breaks its invariants)
- Unreachable code after `return`
- Misuse of `sync/atomic`, like `x = atomic.AddUint64(&x, 1)` (the assignment races with the atomic update)
- Calling `t.Fatal` or `t.FailNow` from a goroutine inside a test (`t.Error` is goroutine-safe)

Tip: Run `go vet ./...` as part of your normal build step, not just on CI. It is fast enough that there is no reason to skip it locally. Many Makefiles include a `vet` target that runs before tests.

Wut: `go vet` is not a linter. It reports only definite bugs or misuses, not style violations or suspicious patterns. For broader coverage, use `golangci-lint` (see below).

golangci-lint

`golangci-lint` is the standard linter aggregator for Go. It runs dozens of individual linters in parallel, merges their output, and is fast enough to use in CI. The closest Java equivalent is a combination of Checkstyle, SpotBugs, PMD, and ErrorProne — all configured in one place.

Install it:

```
# macOS / Linux via the official installer (preferred)
curl -sSfL https://raw.githubusercontent.com/golangci/golangci-lint/master/install.sh \
    | sh -s -- -b $(go env GOPATH)/bin v2.12.2

# or via go install (slower, but always available)
go install github.com/golangci/golangci-lint/v2/cmd/golangci-lint@latest
```

Run it:

```
golangci-lint run ./...             # run all enabled linters
golangci-lint run --fix ./...       # auto-fix what can be auto-fixed
golangci-lint linters               # list all available linters and their status
```

Configuration

Configuration lives in `.golangci.yml` at the module root. A minimal starting config:

```
version: "2"

linters:
  enable:
    - govet        # go vet findings
    - errcheck     # unchecked errors
    - staticcheck  # advanced static analysis (now includes the old gosimple checks)
    - ineffassign  # assignments that are never read
```

```
    - unused       # unused code
  settings:
    govet:
      enable-all: true
```

In golangci-lint v2 the built-in exclusion presets are off by default, so there is no key to disable them. If you *want* the common false-positive exclusions, opt in under `linters.exclusions.presets` (for example `common-false-positives` or `legacy`).

Tip: Start with a small, agreed-upon set of linters and expand over time. Enabling every linter at once on an existing codebase produces a wall of noise that discourages use. The linters `govet`, `errcheck`, `staticcheck`, and `unused` are a solid first set.

Trap: Do not install `golangci-lint` with `go install` in CI. The resulting binary depends on which Go version is in the environment, and linter behavior can change between releases. Use the pinned version from the official install script or a pinned Docker image instead.

go doc and godoc

Go uses plain comments immediately above declarations as documentation — no annotation syntax, no XML tags. The Java equivalent is Javadoc; the Go equivalent is much simpler.

A documented function looks like this:

```go
// Greet returns a greeting for the named person.
// If name is empty, it returns a generic greeting.
func Greet(name string) string {
    if name == "" {
        return "Hello, stranger."
    }
    return "Hello, " + name + "."
}
```

The `go doc` command reads these comments from source and displays them in the terminal:

```
go doc fmt                    # show package-level docs for fmt
go doc fmt.Println            # show docs for a specific function
go doc -all fmt               # show all exported names and their docs
go doc -src fmt.Println       # show the source of the function
```

Since Go 1.25, `go doc` can also serve documentation as HTML in the same format as pkg.go.dev — no extra install needed:

```
go doc -http=:6060            # serve docs at http://localhost:6060
```

(The deprecated `golang.org/x/tools/cmd/godoc` command was the way to do this before Go 1.25; you may still see it referenced.)

Tip: Write doc comments for every exported name. `golangci-lint` includes the `godot` linter, which enforces that doc comments end with a period. Treat a missing doc comment on an exported symbol the same way you would treat a missing Javadoc block.

Wut: Unlike Javadoc, Go doc comments are plain text — no HTML, no `@param`, no `@return` tags. Since Go 1.19, the `go doc` format supports a lightweight markup (lists, code fences, links) but it is still far simpler than Javadoc.

gopls

`gopls` (pronounced "go please") is the official Go language server. It implements the Language Server Protocol (LSP), the same protocol that powers Java support in VS Code via the Eclipse JDT Language Server and in IntelliJ via its built-in engine. Any editor that speaks LSP — VS Code, Neovim, Emacs, Helix, and others — can use `gopls` for Go support.

`gopls` provides:

- Code completion
- Go-to-definition and find-references
- Inline type information and documentation
- Rename refactoring
- Auto-import via `goimports`
- Real-time `go vet` and some `staticcheck` diagnostics
- Inlay hints for parameter names and return types

You rarely invoke `gopls` directly; your editor or its Go plugin manages it. Install or update it:

```
go install golang.org/x/tools/gopls@latest
```

Tip: Keep `gopls` updated. It improves rapidly and newer versions understand newer Go language features. Running an old `gopls` against a module that uses a new Go version can produce confusing false errors.

Trap: If `gopls` is slow or consuming too much memory on a large module, check that your `GOPATH` module cache is not inside a Dropbox or similar sync folder. File-watcher conflicts with sync tools are a common source of `gopls` instability.

Delve

Delve is the standard debugger for Go. The Java equivalent is `jdb` on the command line, or the debugger built into IntelliJ IDEA and Eclipse. Delve understands goroutines, deferred functions, and Go's calling conventions — things that a generic C debugger like GDB cannot handle correctly.

Install Delve:

```
go install github.com/go-delve/delve/cmd/dlv@latest
```

Key Commands

Start a debug session for a main package:

```
dlv debug ./cmd/myapp              # compile with debug info and attach
dlv debug ./cmd/myapp -- --port 8080  # pass flags to the program after --
```

Debug tests in a package:

```
dlv test ./pkg/mypackage           # debug test binary
dlv test ./pkg/mypackage -- -test.run TestFoo  # run only TestFoo under the debugger
```

Attach to an already-running process:

```
dlv attach <pid>                   # attach to a running process by PID
```

Inside a Delve session the most useful commands are:

```
break main.main       # set a breakpoint at a function
break myfile.go:42    # set a breakpoint at a file and line
continue              # run until the next breakpoint
next                  # step over the current line
step                  # step into the current call
stepout               # step out of the current function
print varname         # print a variable
locals                # print all local variables
goroutines            # list all goroutines
goroutine 3           # switch to goroutine 3
stack                 # print the current call stack
```

Tip: In VS Code with the Go extension, the **Run and Debug** panel uses Delve under the hood. You get full GUI debugging — breakpoints, watch expressions, call stack, goroutine list — without leaving the editor. IntelliJ IDEA's Go plugin also integrates Delve. For day-to-day work you rarely need the `dlv` CLI directly.

Wut: A plain `go build` keeps DWARF debug info but compiles with optimizations and inlining, which makes stepping jumpy and variables invisible. `dlv debug` rebuilds with `-gcflags="all=-N -l"` to disable them, but if you attach to a production binary built with `-ldflags="-s -w"` you will have no symbols at all. Keep an unstripped build around when you need to debug a production issue.

Goroutine Inspection

One of Delve's most useful capabilities over `jdb` is first-class goroutine support. The `goroutines` command lists every live goroutine with its current state and top-of-stack location. The `goroutine <id>` command switches your debugging context to that goroutine, and `stack` then shows its full call stack. This makes diagnosing deadlocks and goroutine leaks far easier than thread dumps in Java.

go build -gcflags=-m

Every Go value is allocated either on the stack or on the heap. Stack allocation is free; heap allocation requires the garbage collector to eventually reclaim it. The compiler performs *escape analysis* to decide where each value lives: if a value's address outlives the function that created it, the value *escapes to the heap*.

You can see these decisions:

```
go build -gcflags='-m' ./...              # print escape analysis decisions
go build -gcflags='-m=2' ./...            # more verbose; show the reason for each decision
go build -gcflags='-m -l' ./...           # disable inlining, then show escape analysis
```

Sample output for a small function:

```
./main.go:12:6: can inline greet
./main.go:18:12: "hola " + name escapes to heap
./main.go:22:13: moved to heap: result
```

Java programmers do not usually think about stack vs. heap allocation explicitly because the JVM decides everything. In Go the distinction matters for performance-sensitive code: a value that escapes forces a heap allocation and adds GC pressure.

Tip: Do not prematurely optimize by trying to prevent all escapes. Run `-gcflags=-m` only when a profiler shows that allocations are a bottleneck, then look at the output to understand why specific values escape and whether anything can be restructured to avoid it.

Wut: Passing a pointer to a function does not automatically cause the pointee to escape. The compiler performs interprocedural analysis. If it can prove the pointer does not outlive the call, the value stays on the stack. This is why small structs passed by pointer in tight loops often don't show up as heap allocations.

Summary

The table below maps each Go tool to its closest Java equivalent.

Go Tool	Purpose	Java Equivalent
`gofmt`	Canonical code formatter	Google Java Format, Spotless
`goimports`	Format + manage imports	IntelliJ optimize imports, `google-java-format`
`go vet`	Built-in static analysis	SpotBugs, ErrorProne
`golangci-lint`	Linter aggregator	Checkstyle + SpotBugs + PMD combined
`go doc`	Terminal doc viewer	`javadoc` CLI output
`go doc -http`	Local documentation server (Go 1.25+)	`javadoc` HTML output, pkg.go.dev
`gopls`	LSP language server	Eclipse JDT LS, IntelliJ built-in engine
`dlv` (Delve)	Debugger with goroutine support	`jdb`, IntelliJ / Eclipse debugger
`go build -gcflags=-m`	Escape analysis output	JVM `-XX:+PrintEscapeAnalysis` (JIT only)

Appendix C

Go Code Review Rules

The rules in this appendix are drawn from the official **Go Code Review Comments** wiki maintained by the Go team (The Go Authors 2024). That document describes the kinds of issues that arise during code review of Go programs — the things that automated tools like `gofmt` and `go vet` do not catch. Each rule is numbered **CR-N** for ordering within this appendix and is given a **short-descriptive-name**. The main text cites a rule by its short name in bold italics, written as [***short-rule-name***] (the convention introduced in Chapter 0), so you can find the matching entry here by name.

Formatting

CR-1. gofmt Run `gofmt` (or `go fmt`) on all code to automatically fix mechanical style issues before review.

CR-2. goimports Prefer `goimports` over `gofmt`; it is a superset that also adds missing and removes unused imports.

Comments

CR-3. sentence-for-comments Comments that document declarations must be complete sentences ending with a period.

CR-4. name-starts-comment A doc comment should begin with the name of the thing it describes: `// Request represents a request to run a command.`

CR-5. all-top-comments All exported top-level names must have doc comments; non-trivial unexported declarations should too.

Context

CR-6. ctx-for-context Functions that use `context.Context` should accept it as the first parameter, named `ctx`.

CR-7. no-ctx-for-struct Never store a `Context` in a struct; pass it as a method parameter instead (exception: signatures forced by third-party interfaces).

CR-8. pass-ctx-by-default Prefer passing `context.Context` even when you think you don't need it; only use `context.Background()` with a clear reason.

CR-9. data-in-params-not-ctx Keep application data in function parameters, receivers, or globals — not in `Context` values — unless it genuinely belongs to the request lifecycle.

CR-10. ctx-safe-for-concurrent A `Context` is safe to pass to concurrent calls that share its deadline and cancellation signal.

Copying

CR-11. no-copy-pointer-type Do not copy a value of type `T` if its methods are on `*T`; copying may cause unexpected aliasing of internal slice or pointer fields.

Cryptographic Randomness

CR-12. crypto-rand-for-keys Never use `math/rand` or `math/rand/v2` to generate keys, tokens, or other security-sensitive values; use `crypto/rand.Reader`.

CR-13. crypto-rand-for-text For random text output use `crypto/rand.Text()` or encode `crypto/rand.Reader` bytes with `encoding/hex` or `encoding/base64`.

Declaring Empty Slices

CR-14. nil-slice-preferred Prefer `var t []string` (nil slice) over `t := []string{}` (non-nil empty slice); they are functionally equivalent for `len` and `cap`, and nil is the idiomatic zero value — but note a nil slice encodes to JSON `null` while a non-nil empty slice encodes to `[]`, so prefer the non-nil form when encoding JSON.

CR-15. no-nil-vs-empty-api Avoid API designs that distinguish between nil and empty slices; the distinction is subtle and causes bugs.

Error Strings

CR-16. lowercase-error-strings Error strings must not be capitalized (unless they begin with a proper noun or acronym) and must not end with punctuation, because they are typically embedded in larger messages: `fmt.Errorf("something bad")` not `fmt.Errorf("Something bad.")`.

Don't Panic

CR-17. errors-not-panic Normal error handling must use error return values and multiple return values rather than `panic`.

CR-18. panic-for-exceptional Reserve `panic` for truly exceptional situations that indicate programmer error or unrecoverable state.

Examples

CR-19. include-examples When adding a new package, include a runnable `Example*` test function that demonstrates the intended usage.

Goroutine Lifetimes

CR-20. goroutine-must-exit When you spawn a goroutine, make it clear when or whether it exits; goroutines that cannot exit are leaks.

CR-21. leaked-goroutine-grows-memory Goroutines block garbage collection of the values they close over; goroutines that leak repeatedly (e.g., one per request) cause unbounded memory growth.

CR-22. obvious-goroutine-lifetimes Keep concurrent code simple enough that goroutine lifetimes are obvious; document lifetime guarantees when simplicity is not achievable.

Handle Errors

CR-23. no-discard-error Never discard an error with `_`; if a function returns an error, check it.

CR-24. handle-return-or-panic When you receive an error, either handle it, return it to the caller, or (in truly exceptional cases) panic — never silently ignore it.

Imports

CR-25. no-rename-imports Avoid renaming imports; a well-chosen package name should not require renaming at the call site.

CR-26. rename-local-on-collision When renaming is unavoidable (name collision), rename the most local or project-specific import, not the standard library one.

CR-27. group-imports Organize imports in groups separated by blank lines: standard library first, then third-party, then internal packages.

Import Blank

CR-28. blank-import-main-only Packages imported only for side effects (`import _ "pkg"`) belong only in the `main` package of a program or in tests that require them.

Import Dot

CR-29. no-dot-import Avoid `import .`; it makes code harder to read because it is unclear which identifiers come from the imported package. Use it only in tests that, due to circular dependencies, cannot be made part of the package being tested.

In-Band Errors

CR-30. no-in-band-errors Do not use in-band error signals (returning `-1`, `""`, or `nil` to indicate failure) when those values are also valid results; use a second return value of type `error` or `bool` instead.

CR-31. in-band-ok-if-unambiguous An in-band sentinel value is acceptable only when the sentinel is unambiguously not a valid result (e.g., `strings.Index` returning `-1` for "not found").

Indent Error Flow

CR-32. error-first-return-early Keep the success (normal) code path at the minimum indentation level; handle errors first and return, so readers can scan the happy path without reading error branches.

CR-33. no-else-after-error Avoid the `if err != nil { ... } else { // success }` pattern; invert it so the error branch returns and the success code is unindented.

Initialisms

CR-34. consistent-initialism-case Acronyms and initialisms must have consistent case throughout: `URL` not `Url`, `HTTP` not `Http`, `ID` not `Id`, `ServeHTTP` not `ServeHttp`.

CR-35. lowercase-leading-initialism When an initialism begins an unexported name, lowercase the whole initialism: `xmlHTTPRequest` or `urlPony`.

Interfaces

CR-36. interface-in-consumer Define interfaces in the package that *uses* them, not in the package that implements them; Go's implicit satisfaction makes this possible and keeps dependencies pointing the right way.

CR-37. return-concrete-types Implementing packages should return concrete types (structs or pointers to structs), not interface types; this allows new methods to be added without breaking callers.

CR-38. no-interface-for-mocking Do not define an interface solely to support mocking in tests; design your API so it can be tested through its real public surface, or use a consumer-side fake.

CR-39. no-premature-interface Do not define an interface before you have a realistic use case; premature interfaces lead to awkward, over-abstract designs.

Line Length

CR-40. no-hard-line-limit There is no hard line-length limit; break lines for semantic clarity (a natural pause in the logic), not to satisfy an arbitrary character count.

CR-41. name-before-wrap If a line feels too long, first consider whether a better name or a local variable would eliminate the length problem before adding a line break.

Mixed Caps

CR-42. mixed-caps-always Use `MixedCaps` (or `mixedCaps`) for multi-word names in all contexts, including constants: `maxLength` not `MAX_LENGTH`.

Named Result Parameters

CR-43. name-results-for-clarity Name result parameters when doing so genuinely clarifies the meaning of multiple same-typed return values: `func Location() (lat, long float64, err error)`.

CR-44. no-name-for-naked-return Do not name result parameters solely to enable naked returns in non-trivial functions; the clarity cost of naked returns in longer functions outweighs the brevity gain.

CR-45. name-for-deferred-modify Naming a result parameter is appropriate when a deferred closure needs to modify it (e.g., to capture a close error).

Package Comments

CR-46. comment-adjacent-to-package Package comments must appear immediately above the `package` clause with no blank line between them.

CR-47. package-comment-sentence Package comments must begin with a capital letter and be complete sentences: `// Package math provides basic constants and mathematical functions.`

CR-48. main-package-comment-forms For `main` packages, acceptable forms include: "Binary seedgen ...", "Command seedgen ...", or "The seedgen command ...".

Package Names

CR-49. no-package-name-in-export Remove the package name from exported identifiers: in package `chubby`, use `File` not `ChubbyFile` (callers write `chubby.File`).

CR-50. no-generic-package-names Avoid generic package names such as `util`, `common`, `misc`, `api`, `types`, and `interfaces`; they communicate nothing about purpose.

Pass Values

CR-51. no-pointer-to-save-bytes Do not pass a pointer just to save a few bytes; if the function only dereferences `*x` throughout, the argument should not be a pointer.

CR-52. no-pointer-to-string-or-iface Do not pass pointers to strings (`*string`) or interface values (`*io.Reader`) just to save bytes; both are small fixed-size values that can be passed directly.

CR-53. pointer-for-large-structs Exception: large structs or structs expected to grow should be passed by pointer for efficiency.

Receiver Names

CR-54. receiver-name-abbreviation The receiver name should be a short abbreviation of the type name (one or two letters), not `this`, `self`, or `me`.

CR-55. receiver-name-consistent The receiver name must be consistent across all methods of a type: if one method uses `c`, all must use `c`.

Receiver Type

CR-56. pointer-receiver-for-mutation Use a pointer receiver when the method needs to mutate the receiver.

CR-57. pointer-receiver-for-mutex Use a pointer receiver when the struct contains a `sync.Mutex` or similar synchronization field, to avoid copying the lock.

CR-58. pointer-receiver-for-large Use a pointer receiver for large structs or arrays where copying on each call would be expensive.

CR-59. value-receiver-for-immutable Use a value receiver for small, immutable structs or basic types (integers, strings) that hold no pointers and do not need mutation.

CR-60. no-mixed-receivers Do not mix value and pointer receivers on the same type; if any method needs a pointer receiver, use pointer receivers for all methods so the method set is consistent regardless of how the value is stored.

CR-61. default-pointer-receiver When in doubt, use a pointer receiver.

Synchronous Functions

CR-62. prefer-synchronous Prefer synchronous functions — those that return results directly or finish callbacks/channel operations before returning — over asynchronous ones.

CR-63. caller-adds-concurrency If callers need concurrency, they can call a synchronous function from a goroutine; removing unnecessary concurrency from an API is much harder after the fact.

Useful Test Failures

CR-64. test-failure-describes-wrong Test failure messages must describe what was wrong: state the inputs, the actual output, and the expected output.

CR-65. actual-before-expected Write failure messages in the order `actual != expected`: `t.Errorf("Foo(%q) = %d; want %d", in, got, want)`.

CR-66. table-driven-tests Use table-driven tests to reduce repetition and to ensure every case produces an identifiable failure message.

CR-67. t-helper-for-helpers Ensure test helpers produce useful failure messages that identify which case failed; use `t.Helper()` to attribute failures to the call site.

Variable Names

CR-68. short-local-names Prefer short variable names, especially for local variables with limited scope: `c` over `lineCount`, `i` over `sliceIndex`.

CR-69. scope-determines-length The further a variable's use is from its declaration, the more descriptive its name must be; single-letter names are appropriate only for very short scopes.

CR-70. descriptive-global-names Global variables and variables representing unusual or domain-specific concepts require descriptive names.

Index

Google. 2024. “Go Best Practices: Package Size.” Google’s Go Style Guide. https://google.github.io/styleguide/go/best-practices.html#package-size.
Kernighan, Brian W., and Dennis M. Ritchie. 1988. *The C Programming Language*. 2nd ed. Prentice Hall.
Pike, Rob. 2009. “Re: Tabs or spaces?” golang-nuts mailing list. https://groups.google.com/g/golang-nuts/c/iHGLTFalb54/m/zqMoq9JRBAAJ.
Pike, Rob. 2015. “Errors are values.” The Go Blog. https://go.dev/blog/errors-are-values.
The Go Authors. 2024. “Go Code Review Comments.” Go Wiki. https://go.dev/wiki/CodeReviewComments.
The Go Authors. 2025. “The Go Programming Language Specification.” https://go.dev/ref/spec. https://go.dev/ref/spec.
The gRPC Authors. 2025. “Basics Tutorial: gRPC in Go.” gRPC Documentation. https://grpc.io/docs/languages/go/basics/.
Wikipedia contributors. 2024. “Programming idiom.” Wikipedia. https://en.wikipedia.org/wiki/Programming_idiom.