Gorgo Go for Java Programmers

June 11, 2026

Contents

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:

```go
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.