Gorgo Go for Java Programmers

June 11, 2026

Contents

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`.

Pike, Rob. 2009. "Re: Tabs or spaces?" golang-nuts mailing list. https://groups.google.com/g/golang-nuts/c/iHGLTFalb54/m/zqMoq9JRBAAJ.