Gorgo Go for Java Programmers

June 11, 2026

Contents

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

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`.