Gorgo Go for Java Programmers — Answer Key

June 11, 2026

About This Answer Key

Work out the answers yourself before looking here. The explanations are meant to help you understand *why* the answer is what it is, not just tell you *what* the answer is.

Chapter 0: How to use this booklet — Answers

There are no exercises in Chapter 0.

Chapter 1: Hello, Go — Answers

Exercise 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?

What you gain:

- Simplicity — one rule covers everything: uppercase means visible outside the package, lowercase means not. There is no keyword to remember, no risk of accidentally writing `protected` when you meant `private`, and no three-word modifier like `public static final`.
- Consistency — the rule applies uniformly to functions, types, variables, struct fields, and methods. There is no special case for classes versus members.
- Readability — any identifier starting with a capital letter is part of the public API; the rest is implementation detail. You can tell at a glance what is exported without an IDE.

What you lose:

- Granularity — Java's four levels let you express "visible to subclasses in other packages" (`protected`) and "visible within the package only" (package-private) as distinct states. Go collapses both into "unexported."
- Per-symbol control within a package — every unexported name in a package is equally visible to every other file in that package. You cannot hide a helper from one file while exposing it to another in the same package.

A Java pattern with no direct Go equivalent:

`protected` members — visible to subclasses anywhere but hidden from unrelated code in other packages. Go has no inheritance, so the idea of "subclass visibility" does not exist. The closest substitutes are the `internal/` directory convention (covered in Chapter 13) and interfaces, but neither maps exactly onto `protected`.

Exercise 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)
}
```

Output:

```
Ozzy Osbourne has 2100000000 plays
type of plays: int
quoted: "Ozzy Osbourne"
```

`%s` formats the string without quotes. `%d` formats the integer in base 10; the `_` digit separators in the source literal `2_100_000_000` are purely cosmetic, so the value is `2100000000`. `%T` prints the Go type name, which for an untyped integer literal assigned with `:=` is `int`. `%q` wraps the string in double quotes and escapes any characters that need it.

Exercise 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?

`os.Args` contains every token on the command line including the program name at index 0. The full slice is `["<binary>", "Sandstorm", "Remix", "Darude"]`, so `len(os.Args)` is `4`. `os.Args[2]` is `"Remix"` (the second user-supplied argument, at index 2).

Output:

```
4
Remix
```

Exercise 4 (Where is the bug?):

```go
package main
```

```go
import (
    "fmt"
    "math"
)

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

`"math"` is imported but never used. The program will not compile. The compiler produces a message like:

```
./main.go: "math" imported and not used
```

The fix is to remove the `"math"` import. If you were writing this in an editor with `goimports` configured to run on save, it would have stripped the unused import automatically before you even tried to build.

Exercise 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`.

```go
package main

import (
    "fmt"
    "os"
)

func main() {
    if len(os.Args) < 3 {
        fmt.Fprintln(os.Stderr, "usage: plays <title> <count>")
        os.Exit(1)
    }
    title := os.Args[1]
    count := os.Args[2]
    fmt.Printf("%q has %s plays\n", title, count)
}
```

Sample run:

```
$ go run main.go "Sandstorm" 2100000000
"Sandstorm" has 2100000000 plays
```

Notes:

- The check is `len(os.Args) < 3` because index 0 is the binary name, 1 is the title, and 2 is the count.
- The play count is kept as a string here; Chapter 3 covers `strconv.Atoi` for converting it to an integer.
- `fmt.Fprintln(os.Stderr, ...)` sends the usage message to standard error, which is the convention for diagnostic output.
- `os.Exit(1)` terminates the program immediately with a non-zero exit code, signalling failure to the shell.

Chapter 2: Types and Variables — Answers

Exercise 1 (Think about it): In Java, using an uninitialized local variable is a compile error. In Go, every variable has a zero value. What are the practical benefits of Go's approach, and when might zero values mask a bug instead of preventing one?

The primary benefit is safety and predictability: there are no uninitialized reads and no undefined behavior from reading a garbage value off the stack. You can declare a `sync.Mutex` or a `bytes.Buffer` and use it immediately without calling a constructor, because the zero value is deliberately designed to be a valid, ready-to-use state. This is a Go idiom worth internalizing: if you design a type so that its zero value is useful, callers pay no initialization tax.

The risk is silent logic errors. In Java, forgetting to initialize a variable is a compile-time catch. In Go, `var count int` silently starts at `0`, `var name string` at `""`, and `var ptr *SomeType` at `nil`. If you forget to set `ptr` before dereferencing it, you get a runtime panic, not a compile error. Similarly, a `bool` zero value is `false`, so a field like `IsAdmin` starts as `false` — which is the safe default — but a field like `IsEnabled` also starts as `false`, which might not be what you want. Zero values encourage a different discipline: design your data so that the zero state is meaningful and correct.

Exercise 2 (What does this print?):

```go
package main

import "fmt"

type StreamingTier int

const (
    Free     StreamingTier = iota
    Standard
    Premium
    Lossless
)

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

Output:

```
0 1 2 3
tier type: main.StreamingTier, value: 2
```

`iota` starts at `0` for the first constant in a `const` block and increments by `1` for each subsequent constant, so `Free/Standard/Premium/Lossless` are `0/1/2/3`. `tier := Premium` infers the type of `tier` as `StreamingTier` (not plain `int`), so `%T` prints `main.StreamingTier` — the package-qualified named type — while `%d` prints its underlying integer value `2`.

Exercise 3 (Calculation): Given the declarations, which assignments compile and which produce errors?

```go
type Bpm float64

var tempo Bpm    = 120.0
```

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

Line	Result	Reason
A	error	`tempo` is type `Bpm`; Go does not implicitly convert a named type to its underlying `float64`.
B	compiles	`float64(tempo)` is an explicit conversion from `Bpm` to `float64`.
C	error	`cvt` is `float64`; assigning it to a `Bpm` variable needs an explicit `Bpm(cvt)`.

`type Bpm float64` creates a new, distinct type, not an alias. Even though `Bpm` and `float64` share the same underlying representation, the compiler treats them as separate types and requires an explicit conversion in each direction. Lines A and C fail with messages like `cannot use tempo (variable of float64 type Bpm) as float64 value` and `cannot use cvt (variable of type float64) as Bpm value`. The fix for both is an explicit conversion: `float64(tempo)` and `Bpm(cvt)`.

Exercise 4 (Where is the bug?):

```go
package main

import "fmt"

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

`:=` requires at least one *new* variable on the left side. Both `x` and `y` already exist in the same scope, so the third line uses `:=` with no new variables and fails to compile:

```
./main.go:8:10: no new variables on left side of :=
```

The intent is to reassign both at once, so use plain `=`:

```go
x, y = x + y, x
```

That compiles and prints `30 10` — Go evaluates all right-hand expressions before assigning, so `y` receives the *old* `x` (10) even though `x` is updated to `30` in the same statement. (If you genuinely needed a new variable here, introducing one — e.g. `x, z := x + y, x` — would also satisfy the "at least one new variable" rule.)

Exercise 5 (Write a program): Declare a `const` block using `iota + 1` for five chart positions (`Debut`, `Rising`, `Peak`, `Declining`, `Legacy`, numbered 1–5), print each name and value, then declare a variable of that type set to `Peak` and print its Go type with `%T`.

```go
package main

import "fmt"

type ChartPosition int
```

```go
const (
    Debut     ChartPosition = iota + 1 // 1
    Rising                             // 2
    Peak                               // 3
    Declining                          // 4
    Legacy                             // 5
)

func main() {
    fmt.Printf("Debut=%d Rising=%d Peak=%d Declining=%d Legacy=%d\n",
        Debut, Rising, Peak, Declining, Legacy)

    pos := Peak
    fmt.Printf("pos type: %T, value: %d\n", pos, pos)
}
```

Output:

```
Debut=1 Rising=2 Peak=3 Declining=4 Legacy=5
pos type: main.ChartPosition, value: 3
```

`iota` is `0` on the first line of the block, and `iota + 1` shifts the whole sequence to start at `1`. Because the expression is omitted on the following lines, Go repeats the first expression `iota + 1` for each, and `iota` keeps incrementing — giving 2, 3, 4, 5. Starting an enum at 1 is a common idiom: it reserves `0` (the zero value) to mean "unset," so a `var p ChartPosition` that nobody assigned is distinguishable from a real position. `%T` reports the named type `main.ChartPosition`, confirming `pos` is not a bare `int`.

Exercise 6 (What does this print?):

```go
package main

import "fmt"

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

func main() {
    a := 5
    b := &a
    double(b)
    fmt.Println(a)
    fmt.Println(*b)
}
```

Output:

```
10
10
```

`b := &a` makes `b` a pointer to `a`, so `b` and `&a` refer to the same memory. `double(b)` passes that pointer by value, but the pointer still points at `a`, and `*n *= 2` writes through it, doubling `a` from `5` to `10`. After the call, `a` is `10`, and `*b` dereferences the same address, so it also prints `10`. This is how Go functions modify a caller's variable: take a pointer parameter and write through it.

Exercise 7 (Where is the bug?):

```go
package main

import "fmt"

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

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

This does not compile. Inside `addExcitement`, `s` has type `*string` (a pointer), so `s += "!"` tries to add a `string` to a pointer:

```
./main.go:6:2: invalid operation: s += "!" (mismatched types *string and untyped string)
```

You meant to modify the string the pointer points at, not the pointer itself. Dereference first:

```go
func addExcitement(s *string) {
    *s += "!" // append through the pointer
}
```

With the fix, `*s` is the underlying `string`, `*s += "!"` builds a new string and stores it back through the pointer, and the program prints `Out Of The Blue!`. This mirrors Exercise 6: to mutate the caller's value through a pointer parameter, operate on `*p`, never on `p`.

Chapter 3: Strings, Bytes, and Runes — Answers

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

Immutability is not the source of the difference — the unit of iteration and the encoding being walked are. A Java `String` is backed by UTF-16, and when you iterate its `char` values (a `String` is not directly `Iterable`, so you go through `charAt`/index or a stream) the unit is a UTF-16 **code unit**. Go's `for range` decodes the string as **UTF-8** and yields one `rune` (a full Unicode code point) per iteration, with the loop index being the byte offset of that rune's first byte. So Java walks UTF-16 code units while Go walks decoded UTF-8 code points.

The two loops give the same sequence only when the text is restricted to characters that are both a single UTF-16 code unit in Java and a single byte in UTF-8 in Go — i.e. pure ASCII. For any non-ASCII character the counts and per-iteration values diverge: `é` is one Java `char` but two bytes (one rune) in Go, and a non-BMP emoji is two Java `char` values (a surrogate pair) but one Go `rune`.

Exercise 2 (What does this print?):

```go
package main

import "fmt"
```

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

Output:

```
195
```

`"Alizée"` is the characters `A l i z é e`. The four preceding letters (`A`, `l`, `i`, `z`) are one byte each, so the accented `é` (U+00E9) begins at byte index 4 and occupies bytes 4 and 5 (`0xC3 0xA9`). Indexing a string with `s[4]` returns a **byte**, not a character, so you get `195` (`0xC3`), the first byte of the UTF-8 encoding of `é` — not the character `é`, and not the trailing plain `e` (which lives at byte index 6).

Exercise 3 (Calculation): `len("Alizée")` returns how many bytes? And `utf8.RuneCountInString("Alizée")` returns how many runes?

`len("Alizée")` returns `7`. The letters `A`, `l`, `i`, `z`, `e` are each one byte (5 bytes), and `é` (U+00E9) encodes to two bytes in UTF-8 (`0xC3 0xA9`), for 7 bytes total.

`utf8.RuneCountInString("Alizée")` returns `6`: six code points — `A`, `l`, `i`, `z`, `é`, `e`. `len` counts bytes; `RuneCountInString` counts code points. They agree only on pure ASCII.

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

```go
func swapCase(s string) string {
    b := []byte(s)
    for i := range b {
        switch {
        case b[i] >= 'A' && b[i] <= 'Z':
            b[i] += 32
        case b[i] >= 'a' && b[i] <= 'z':
            b[i] -= 32
        }
    }
    return string(b)
}

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

The bug: the function treats every byte as an independent ASCII letter. For the ASCII letters this works as intended. Tracing `"Héroe"` byte by byte: `H` is 72, in `A`–`Z`, so it becomes 104 (`h`). The `é` is the two bytes `0xC3` (195) and `0xA9` (169); neither falls in the `A`–`Z` or `a`–`z` ranges, so the `switch` leaves both bytes untouched. Then `r` (114), `o` (111), and `e` (101) are each in `a`–`z`, so they lose 32 and become `R` (82), `O` (79), `E` (69).

So the program prints:

```
héROE
```

The `é` survives intact by design, not by luck: UTF-8 guarantees that every byte of a multibyte sequence is at least `0x80`, so it can never collide with the ASCII letter ranges and the function can never corrupt valid UTF-8. That guarantee is also exactly what exposes the fundamental flaw — the function silently fails to change the case of any non-ASCII letter (`é` should have become `É`), because byte-wise case logic only handles

ASCII. The correct approach is to work on runes (`for range` or `[]rune`), or simply use `strings.ToUpper` / `strings.ToLower`, which are Unicode-aware.

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

```
package main

import "fmt"

func reverseString(s string) string {
    runes := []rune(s)
    for i, j := 0, len(runes)-1; i < j; i, j = i+1, j-1 {
        runes[i], runes[j] = runes[j], runes[i]
    }
    return string(runes)
}

func main() {
    words := []string{"café", "Beyoncé", "hello"}
    for _, w := range words {
        fmt.Printf("%q -> %q\n", w, reverseString(w))
    }
}
```

Output:

```
"café" -> "éfac"
"Beyoncé" -> "écnoyeB"
"hello" -> "olleh"
```

The key step is converting the string to `[]rune` first. This decodes the UTF-8 and gives you one `rune` per Unicode code point regardless of how many bytes each one occupies. You then swap elements in the rune slice in place and convert back to a `string` at the end. Reversing the raw `[]byte` instead would shuffle the individual bytes of multibyte characters and produce invalid UTF-8 — `café` would come out garbled rather than as `éfac`.

Chapter 4: Control Flow — Answers

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

Look for stray `break` statements and, more importantly, for cases that were *meant* to share logic via fall-through. A Java programmer used to writing

```
switch (level) {
    case ERROR:
    case WARN:
        log(msg);   // intended: ERROR also logs
        break;
}
```

might translate it to Go as two separate cases, each with its own body, and only put the logging in one of them — assuming `case ERROR:` would "fall into" `case WARN:`. In Go it does not, so an `ERROR` value silently does nothing. The fix is either to list both values in one case (`case ERROR, WARN:`) or to add an explicit `fallthrough`. The bug is silent because the code compiles and runs — it just quietly skips the shared work.

Exercise 2 (What does this print?):

```go
package main

import "fmt"

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

Output:

```
done
2
1
0
```

Two things are happening here. First, `defer` runs after the surrounding function returns, so all three deferred calls happen after `fmt.Println("done")`. Second, `defer` arguments are evaluated immediately at the point of the `defer` statement, not when the deferred call executes. When `i` is `0`, `defer fmt.Println(0)` is registered with the value `0` baked in. When `i` is `1`, `defer fmt.Println(1)` is registered with `1`. When `i` is `2`, `defer fmt.Println(2)` is registered with `2`. Deferred calls execute in LIFO (last-in, first-out) order, so the last one registered runs first: `2`, then `1`, then `0`.

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

```go
package main

import "fmt"

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

func main() {
    classify(-3)
```

```
    classify(0)
    classify(4)
    classify(7)
}
```

Output:

```
negative
zero
positive even
positive odd
```

An expression-less `switch` is equivalent to `switch true` — each case is a boolean expression, and the first one that evaluates to `true` wins. Cases are evaluated top to bottom; once a match is found, the remaining cases are skipped. `classify(4)` skips `n < 0` and `n == 0`, then matches `n%2 == 0` and prints "positive even". `classify(7)` matches none of the cases, so the `default` clause prints "positive odd".

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

```
package main

import "fmt"

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

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

All three calls print `150`, not `50`, `100`, `150`.

The bug is that `factor` is declared outside the loop. All three closures capture the same `factor` variable by reference. By the time the closures run, the loop has finished and `factor` is `30` (the last value assigned). Every closure multiplies by `30`, so `f(5)` returns `150` for all three.

The fix is to declare `factor` inside the loop so each iteration gets its own copy:

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

Now each closure captures a distinct `factor` variable, and the output is `50`, `100`, `150`. Note that Go's per-iteration loop variable semantics (Go 1.22 and later) fix the classic `i`-capture bug automatically, but they do

not help here because the captured variable is `factor`, not the loop variable.

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

```go
package main

import (
    "fmt"
    "io"
    "os"
)

func processFile(path string) error {
    f, err := os.Open(path)
    if err != nil {
        return err
    }
    defer f.Close() // runs on every return path below

    buf := make([]byte, 64)
    n, err := f.Read(buf)
    if err != nil && err != io.EOF {
        return err
    }
    fmt.Printf("%s: %q\n", path, string(buf[:n]))
    return nil
}

func main() {
    if err := processFile("cancion.txt"); err != nil {
        fmt.Println("error:", err)
    }
    if err := processFile("missing.txt"); err != nil {
        fmt.Println("error:", err)
    }
}
```

The `defer f.Close()` is registered immediately after a successful `os.Open`, so the file is closed no matter which `return` the function takes — the successful path, the read-error path, or a panic. For the missing path, `os.Open` fails before the `defer` is registered (there is nothing to close), and `processFile` returns the wrapped `*PathError`, which `main` prints as something like `error: open missing.txt: no such file or directory`. Note that a single `Read` is not guaranteed to fill the buffer; using `io.ReadFull` would be the robust choice when you truly need all 64 bytes.

Exercise 6 (Calculation): Consider this loop.

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

The loop body executes **6** times, and the value of `i` when the loop condition is evaluated for the last time (and fails) is **128**.

`i` takes the values 2, 4, 8, 16, 32, 64 while the condition `i < 100` holds — that is six iterations, so `count` ends at 6. After the body runs with `i == 64`, the post statement doubles `i` to 128; the condition `128 < 100` is false, so the loop exits with `i == 128`.

Chapter 5: Functions — Answers

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

Java's checked exceptions provide a compiler-enforced contract: if a method declares `throws IOException`, the caller must either catch it or declare that it also throws it. There is no way to silently ignore a checked exception in Java without at least writing a catch block (even an empty one is conspicuous).

Go does not provide that same guarantee. You can write `result, _ := divide(a, b)` to throw the error away, and the compiler is perfectly happy. The compiler will reject an assignment that captures the wrong *number* of return values, so you cannot accidentally drop the error by writing `result := divide(a, b)` when `divide` returns two values — but it will gladly accept _ for any value you choose to discard.

What Go gains:

- Error handling is explicit code, not a separate control-flow mechanism. Errors flow through the same call stack as regular values, which makes them easier to wrap, annotate, and inspect.
- There is no distinction between checked and unchecked exceptions to manage. In Java, many APIs force you to wrap checked exceptions in `RuntimeException` just to use them in lambdas or streams.
- Errors are plain values you can store, compare, pass around, and test — no reflection required.

What Java gains:

- The compiler can prove at build time that every failure path is addressed. Go relies on code review and tools like `errcheck` to catch ignored errors.
- Stack traces are attached to exceptions automatically; in Go you must explicitly wrap errors with `fmt.Errorf("...: %w", err)` to build a chain.

In practice, Go's approach leads to more explicit error-handling code and fewer "surprise" failures from unchecked exceptions — but it also produces more repetitive `if err != nil` checks that disciplined engineers must not skip.

Exercise 2 (What does this print?):

```go
package main

import "fmt"

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

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

Output:

```
6
7
6
5
```

`makeAdder(5)` creates a single variable `n` initialized to `5`. Both `inc` and `dec` are closures that capture that *same* `n`, so they share one piece of state — mutating `n` through one closure is visible to the other.

- `inc()`: `n = 5 + 1 = 6`; prints `6`.
- `inc()`: `n = 6 + 1 = 7`; prints `7`.
- `dec()`: `n = 7 - 1 = 6`; prints `6`.
- `dec()`: `n = 6 - 1 = 5`; prints `5`.

The key point: returning two closures that close over the same variable gives you a pair of functions that operate on shared, hidden state — the closure equivalent of two methods on one struct field.

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

```
package main

import "fmt"

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

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

Output:

```
110
130
125
```

`running(100)` creates a closure that captures `total`, initialised to `100`. The same `total` variable is shared across all calls through `acc` because the closure captures it — it is the same memory location every time.

- `acc(10)`: `total = 100 + 10 = 110`; returns and prints `110`.

- `acc(20)`: `total = 110 + 20 = 130`; returns and prints `130`.
- `acc(-5)`: `total = 130 + (-5) = 125`; returns and prints `125`.

This is a running total (accumulator) implemented with a closure. Each call modifies and returns the accumulated value.

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

```go
package main

import "fmt"

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

The bug: Under Go 1.21 semantics, the `for range` loop reuses the same `i` and `name` variables for every iteration. All three closures capture the same `name` variable — not a copy of its value, but a reference to the single loop variable. By the time any of the closures run, the loop has finished and `name` holds the last value: `"priya"`.

Actual output under Go 1.21:

```
hola, priya
hola, priya
hola, priya
```

Expected output:

```
hola, benson
hola, amara
hola, priya
```

Fix 1 (works in all Go versions): Create a new local variable inside the loop body to shadow the loop variable:

```go
for i, name := range names {
    name := name // create a new variable for this iteration
    greets[i] = func() { fmt.Println("hola,", name) }
}
```

Fix 2 (works in Go 1.22+): Update the `go` directive in `go.mod` to `go 1.22` or later. The language change makes each loop iteration create its own variable automatically, so all existing closure-in-loop code behaves correctly without any source changes.

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

```go
package main

import "fmt"

// pipeline returns a function that applies each fn in fns left to right.
func pipeline(fns ...func(int) int) func(int) int {
    return func(x int) int {
        result := x
        for _, fn := range fns {
            result = fn(result) // apply each transform in sequence
        }
        return result
    }
}

func main() {
    double := func(n int) int { return n * 2 }  // multiply by 2
    addTen := func(n int) int { return n + 10 } // add 10
    square := func(n int) int { return n * n }  // square the value

    // double then addTen: (3*2)+10 = 16
    p1 := pipeline(double, addTen)
    fmt.Println(p1(3)) // 16

    // addTen then double: (3+10)*2 = 26
    p2 := pipeline(addTen, double)
    fmt.Println(p2(3)) // 26

    // double then square: (3*2)^2 = 36
    p3 := pipeline(double, square)
    fmt.Println(p3(3)) // 36

    // empty pipeline --- identity
    p4 := pipeline()
    fmt.Println(p4(7)) // 7
}
```

Key points in this solution:

- `pipeline` is variadic: it accepts any number of `func(int) int` values.
- Inside the returned closure, `fns` is captured — the same `[]func(int) int` slice the outer call built.
- The order of application matters: `pipeline(double, addTen)` and `pipeline(addTen, double)` produce different results for the same input.
- An empty `pipeline()` call returns an identity function because the loop body never executes.

Chapter 6: Objects using Methods and Embedding — Answers

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

Advantages of Go's separated approach:

1. **You can attach methods to any named type, not just classes.** In Java, methods live inside class bodies and you cannot add methods to types defined in other packages. In Go, you can define methods on any named type declared in the same package, including types built on standard-library types (e.g., `type Seconds float64`).
2. **Code reuse without coupling.** Java inheritance forces an is-a relationship: `FeaturedTrack extends Track` means every `FeaturedTrack` is substitutable for a `Track`. Go embedding is a has-a relationship with no automatic substitutability. You get promoted fields and methods without locking yourself into a hierarchy that may become wrong later.
3. **Interfaces decouple behavior from data completely.** A type satisfies a Go interface without naming or knowing the interface exists. This lets you define interfaces in the consumer package, not the producer package, so dependencies flow the right way.
4. **No fragile base-class problem.** Java's virtual dispatch means a change to a superclass method can silently alter the behavior of all subclasses. Go's promoted methods are not virtual: calling a promoted method on an outer struct always calls the embedded type's method, unless the outer struct defines its own method with the same name.

Where Java's approach is simpler:

- When you genuinely want polymorphism through a type hierarchy (e.g., a UI widget tree), Java's `extends` gives you substitutability, virtual dispatch, and `instanceof` checks in one declaration. In Go you need an interface plus embedding, and you must ensure both the outer and inner types satisfy the interface.
- A `toString()` override in Java is automatic through the `Object` base class. In Go, `fmt.Stringer` requires you to implement `String() string` on each type that wants custom formatting; there is no default.

Exercise 2 (What does this print?):

```go
package main

import "fmt"

type Base struct {
    ID int
}

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

type Widget struct {
    Base
    Color string
}

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

```
    fmt.Println(w.Describe())
    fmt.Println(w.Base.Describe())
}
```

Output:

```
42
blue
Base ID=42
Base ID=42
```

`w.ID` is promoted from `w.Base.ID` — accessing `w.ID` and `w.Base.ID` reach the same field. `w.Color` is a direct field of `Widget`. `w.Describe()` calls the promoted `Base.Describe()` method, because `Widget` has no `Describe` method of its own. `w.Base.Describe()` calls the same method through the explicit path. Both calls produce identical output.

Exercise 3 (Calculation): Given the types below, count the methods in each method set.

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

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

type FeaturedTrack struct {
    Track
    Feature string
}

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

Answer:

- a. `Track` (the value type): **1** — only `IsLong`, the sole value-receiver method. `String` and `ScaleBPM` have pointer receivers, so they are not in the value type's method set.
- b. `*Track`: **3** — `String`, `ScaleBPM`, and `IsLong`. A pointer type's method set includes both pointer-receiver and value-receiver methods.
- c. `FeaturedTrack` (the value type), counting promoted methods: **1** — only the promoted `IsLong`. Embedding a value `Track` promotes `Track`'s value-receiver method (`IsLong`) into `FeaturedTrack`'s value method set; the pointer-receiver methods (`Track.String`, `Track.ScaleBPM`) are not promoted into a value method set. `FeaturedTrack`'s own `String` has a pointer receiver, so it is not in the value method set either.
- d. `*FeaturedTrack`, counting promoted methods: **3** — its own `String` (which shadows the promoted `Track.String`), plus the promoted `ScaleBPM` and `IsLong`. Embedding a value `Track` promotes the full `*Track` method set onto `*FeaturedTrack`; the outer `String` shadows the inner one, so the count stays **3**, not 4.

The key insight: pointer-receiver methods join only the pointer type's method set, and an outer method with the same name shadows the promoted method rather than adding to the count. You can confirm these counts with `reflect.TypeOf(x).NumMethod()`, which reports 1, 3, 1, and 3 respectively.

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

```go
package main

import "fmt"

type Artist struct {
    Name string
}

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

type Song struct {
    *Artist
    Title string
}

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

The bug: `Song` is initialized without setting the embedded `*Artist` pointer, so `s.Artist` is `nil`. The first `fmt.Println(s.Title)` prints `Out Of The Blue` successfully. Line A (`fmt.Println(s.Label())`) is where the program panics. `Label` is promoted from the embedded `*Artist`, and to call it Go must dereference the embedded pointer. Dereferencing a `nil` pointer causes a runtime panic:

```
panic: runtime error: invalid memory address or nil pointer dereference
```

Fix: initialize the embedded pointer before use:

```go
s := Song{
    Artist: &Artist{Name: "System F"},
    Title:  "Out Of The Blue",
}
fmt.Println(s.Title)   // Out Of The Blue
fmt.Println(s.Label()) // Artist: System F
```

Alternatively, construct `Song` through a constructor that enforces initialization:

```go
func NewSong(title, artistName string) Song {
    return Song{Artist: &Artist{Name: artistName}, Title: title}
}
```

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

```go
package main
```

```go
import "fmt"

type Counter struct {
    Value int
}

func NewCounter(start int) *Counter { // constructor: returns *Counter for pointer receivers
    return &Counter{Value: start}
}

func (c *Counter) Increment()     { c.Value++ }       // pointer receiver: mutates Value
func (c *Counter) Reset()         { c.Value = 0 }     // pointer receiver: mutates Value
func (c *Counter) String() string {                   // pointer receiver for consistency
    return fmt.Sprintf("count: %d", c.Value)
}

func main() {
    defer fmt.Println("done") // runs last, after main's body returns

    c := NewCounter(10)
    c.Increment()
    c.Increment()
    c.Increment()
    fmt.Println(c) // count: 13
    c.Reset()
    fmt.Println(c) // count: 0
}
```

Output:

```
count: 13
count: 0
done
```

Notes:

- `NewCounter` returns `*Counter` so every method call works without taking an address at the call site.
- All three methods use pointer receivers for consistency — since `Increment` and `Reset` must mutate `c.Value`, keeping `String` on the pointer receiver too keeps the method set uniform.
- `fmt.Println(c)` calls `c.String()` automatically because `*Counter` satisfies `fmt.Stringer` (which requires `String() string`).
- `defer fmt.Println("done")` is registered first but runs last, when `main` returns, so `done` is the final line of output — the same LIFO cleanup mechanism used for releasing resources.

Exercise 6 (Where is the bug?): The following program does not compile. Explain the exact reason the compiler rejects it, and describe the smallest change that makes it work.

```go
package main

import "fmt"

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

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

The program does not compile because you cannot attach a method to a non-local type. `int` is a predeclared type defined by the language, not in this package, so `func (n int) IsFast() bool` is rejected with *cannot define new methods on non-local type int*. The same rule blocks attaching methods to types from other packages, such as `time.Duration`.

The smallest fix is to define your own named type whose underlying type is `int`, and hang the method on that:

```go
type BPM int

func (b BPM) IsFast() bool {       // method on a local named type --- allowed
    return b >= 125
}

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

`BPM` keeps all the arithmetic and comparison behavior of `int`, but because it is declared in this package you are free to give it methods.

Chapter 7: Maps and Slices — Answers

Exercise 1 (Think about it): In Java, `HashMap<K,V>` requires keys to implement `hashCode()` and `equals()`, and `ArrayList<E>` stores references to boxed objects on the heap. Go's `map[K]V` requires `K` to be comparable at the language level, and a `[]E` slice stores values directly in the backing array. What are the trade-offs of Go's approach for each collection type? Give one example of a Java key type you cannot use directly as a Go map key, and explain one scenario where storing values directly in a slice (rather than as heap references) matters for performance.

For maps, Go's comparable-at-the-language-level requirement is simpler and faster than Java's `hashCode()/equals()` contract, but it is also stricter. You cannot supply custom equality, and some types simply cannot be keys at all. A Java key type you cannot use directly as a Go map key is anything backed by a slice: for example, a Java `List<String>` (or a `byte[]`) used as a `HashMap` key has no Go equivalent, because slices are not comparable in Go. You must convert to a comparable form first — turn a `[]byte` into a `string`, or build a struct or string key out of the pieces. The trade-off is less flexibility (no custom hashing) in exchange for no boxing, no per-key method dispatch, and a guarantee that the compiler rejects un-hashable keys at compile time instead of letting you discover the problem at runtime.

For slices, storing values directly rather than as heap references gives much better memory layout, but it only helps for value types (`int`, `float64`, structs).

A Java `ArrayList<Integer>` stores a pointer (reference) to each boxed `Integer` object, and each `Integer` object lives somewhere on the heap. Iterating over the list means following a pointer for every element, and those `Integer` objects may be scattered around memory — poor cache locality. Each `Integer` also carries object-header overhead (typically 16 bytes) even though the actual integer value is just 4 bytes.

A Go `[]int` stores the integer values **directly** and **contiguously** in the backing array. Iterating is a sequential scan through a flat region of memory: the CPU prefetcher handles this extremely well. There is no per-element allocation overhead and no pointer chasing.

You feel the difference most in:

- **Tight loops** that process large slices: the sequential memory-access pattern is cache-friendly, and modern CPUs can vectorize flat integer arrays.
- **Memory usage**: a Go `[]int` of a million elements is roughly 8 MB (64-bit ints). A Java `ArrayList<Integer>` of a million elements is the list's pointer array (8 MB of references) plus a million `Integer` objects on the heap (at least 16 MB more), for a minimum of 24 MB — and GC pressure from all those small objects.
- **GC pauses**: the Go garbage collector has no pointers to trace inside a `[]int`, so it scans the array in constant time. A Java `ArrayList<Integer>` forces the GC to follow a million references.

The trade-off is that Go's flat layout only pays off for slices of value types. For slices of interfaces or pointers, Go has the same indirection that Java does.

Exercise 2 (What does this print?):

```go
package main

import "fmt"

func main() {
    catalog := map[string]int{
        "Saltwater":       1_200_000_000,
        "Out Of The Blue": 980_000_000,
    }
    hits := []string{"Out Of The Blue", "Watermelon Sugar", "Saltwater"}
    for _, title := range hits {
        if plays, ok := catalog[title]; ok {
            fmt.Printf("%s: %d\n", title, plays)
        } else {
            fmt.Printf("%s: not found\n", title)
        }
    }
}
```

Output:

```
Out Of The Blue: 980000000
Watermelon Sugar: not found
Saltwater: 1200000000
```

The loop iterates the `hits` slice in order. `"Out Of The Blue"` is in the catalog, so the comma-ok idiom sets `ok = true` and its play count is printed. `"Watermelon Sugar"` is not in the catalog, so `ok = false` and the `else` branch runs. `"Saltwater"` is in the catalog and is printed last. Map lookup order is randomized, but slice `range` iteration is always in index order, so the output here is deterministic.

Exercise 3 (Calculation): Given the following code, trace the value of `len(s)` and `cap(s)` after each line. Does any line cause a new backing array to be allocated?

```go
s := make([]int, 2, 5)
s = append(s, 10)
s = append(s, 20)
s = append(s, 30)
s = append(s, 40)
```

After line	`len(s)`	`cap(s)`	New array?
`make([]int, 2, 5)`	2	5	yes (initial)
`append(s, 10)`	3	5	no
`append(s, 20)`	4	5	no
`append(s, 30)`	5	5	no
`append(s, 40)`	6	10	**yes**

`make([]int, 2, 5)` allocates a backing array with capacity 5 and length 2 (two zero-valued ints already present). The first three `append` calls fit within the existing capacity, so the length grows 2 -> 3 -> 4 -> 5 with no new allocation. The fourth `append` would push the length to 6, which exceeds capacity 5, so the runtime allocates a new backing array and copies the existing elements. The exact new capacity is implementation-defined, but it must be at least 6; in go1.26.3 it is 10 (the runtime roughly doubles capacity for small slices).

Exercise 4 (Where is the bug?): The following program tries to build a word-frequency map and then print only the words that appear more than once. It compiles but panics at runtime, even though `"Gamemaster"` appears twice. What is wrong, and how do you fix it?

```go
package main

import "fmt"

func main() {
    words := []string{"Gouryella", "Gamemaster", "Flaming June", "Gamemaster", "Sandstorm"}
    var freq map[string]int
    for _, w := range words {
        freq[w]++
    }
    for word, count := range freq {
        if count > 1 {
            fmt.Println(word, count)
        }
    }
}
```

The bug: `var freq map[string]int` declares a nil map. Reading from a nil map returns the zero value (`0` for `int`), which is harmless, but **writing to a nil map panics** at runtime. The statement `freq[w]++` is a write (it reads the current value, adds one, and stores the result), so the program panics on the very first iteration:

```
panic: assignment to entry in nil map
```

Fix: Initialize the map with `make` (or a literal) before the loop:

```go
freq := make(map[string]int)
for _, w := range words {
    freq[w]++
}
```

With the fix, the program prints:

```
Gamemaster 2
```

`"Gamemaster"` is the only word that appears more than once.

Exercise 5 (Write a program): Write a program that reads a slice of song titles and builds a map from the first letter (as a `string`) to a slice of titles starting with that letter. Use the input `[]string{"Sandstorm", "Bad Apple!!", "Gouryella", "Better Off Alone", "Flaming June", "Sandstorm"}`. Print each letter and its titles in sorted order (sort both the letters and the titles within each group). Collect the map's keys into a slice with a `for range` loop, then use `slices.Sort` for both the keys and the titles within each group.

```go
package main

import (
    "fmt"
    "slices"
)

func main() {
    titles := []string{
        "Sandstorm", "Bad Apple!!", "Gouryella",
        "Better Off Alone", "Flaming June", "Sandstorm",
    }

    byLetter := make(map[string][]string)
    for _, t := range titles {
        letter := string(t[0]) // first byte; safe because all titles start with ASCII
        byLetter[letter] = append(byLetter[letter], t)
    }

    for _, ts := range byLetter {
        slices.Sort(ts) // sort titles within each group
    }

    keys := make([]string, 0, len(byLetter))
    for k := range byLetter {
        keys = append(keys, k)
    }
    slices.Sort(keys) // sort the letter keys

    for _, k := range keys {
        fmt.Printf("%s: %v\n", k, byLetter[k])
    }
}
```

Output:

```
B: [Bad Apple!! Better Off Alone]
F: [Flaming June]
G: [Gouryella]
S: [Sandstorm Sandstorm]
```

Key points: always initialize a map with `make` before writing to it. Collect the map's keys into a slice with a `for range` loop, then call `slices.Sort` on both the keys and the titles within each group, because map iteration order is randomized and you need an explicit sort for deterministic output.

Chapter 8: Interfaces — Answers

Exercise 1 (Think about it): Go's structural typing means any package can retroactively make its types satisfy an interface defined in any other package. In Java, if you want your `Song` class to satisfy a new interface `Playable` defined in a library you do not control, you must modify `Song`'s source. Explain how Go's approach changes the relationship between library authors and library users. What does this mean for extending types from packages you cannot modify?

Go's structural typing decouples the author of a type from the author of an interface. If library A defines `type Track struct { ... }` with a `Play()` method, and later library B defines `interface Playable { Play() }`, then `Track` satisfies `Playable` automatically — neither author needs to know about the other, and no source change is required on either side.

In Java the relationship between a class and an interface is declared at write-time with `implements` and baked into the source. A class can only satisfy interfaces that already existed (or that you can edit the class to add). In Go the relationship is checked at compile-time by the compiler from the methods the type actually has — it emerges from what the type does, not from what it says it is.

For types you cannot modify, this means: if the type already has the methods your interface requires, you can pass it directly wherever that interface is expected, with zero changes to the third-party code. If it does not, you wrap it in your own struct that adds the missing methods (the adapter pattern), or define a new named type whose underlying type is the original and add methods to that.

The practical payoff is that you define narrow interfaces in your own package, on demand, and third-party concrete types satisfy them for free. Interfaces tend to be defined by the consumer, not the producer, which keeps dependencies pointing the right way and makes testing easy: any test double with the right methods slots in.

Exercise 2 (What does this print?):

```go
package main

import "fmt"

type Celsius float64
type Fahrenheit float64

func (c Celsius) String() string {
    return fmt.Sprintf("%.1f°C", float64(c))
}

func printTemp(v fmt.Stringer) {
    fmt.Println(v.String())
}

func main() {
    c := Celsius(37.5)
    f := Fahrenheit(99.5)
    printTemp(c)
    fmt.Println(f)
}
```

Output:

```
37.5°C
99.5
```

`Celsius` has a `String() string` method, so it satisfies `fmt.Stringer` implicitly. `printTemp(c)` calls `v.String()`, which returns `37.5°C`, and `fmt.Println` prints that line.

`Fahrenheit` does **not** have a `String() string` method, so it does not satisfy `fmt.Stringer`. (Methods are not inherited across named types: `Celsius` and `Fahrenheit` are distinct types even though both have underlying type `float64`.) `fmt.Println(f)` therefore formats `f` with the default verb for its underlying type, `float64`, which prints the shortest decimal that rounds back to the value: `99.5`. No degree symbol, no unit, just the number.

Exercise 3 (Calculation): An interface value in Go stores two fields: a pointer to type information and a pointer to (or copy of) the data. Given a variable declared as `var r io.Reader = &bytes.Buffer{}`, how many distinct type/value components does `r` hold? If `r` is then assigned `nil`, describe the type and value components of the resulting interface value.

An interface value always has exactly two components, so `r` holds **two**: a type component and a value (data) component.

After `var r io.Reader = &bytes.Buffer{}`:

- The type component points to the type descriptor (itab) for the dynamic type `*bytes.Buffer` and its `io.Reader` method set.
- The value component holds the `*bytes.Buffer` pointer to the freshly allocated buffer.

Because both components are non-nil, `r == nil` is `false`.

After `r = nil`:

- The type component is `nil`.
- The value component is `nil`.

A nil interface is precisely the state where both components are nil, so now `r == nil` is `true`. This is the key thing to remember: an interface is nil only when both halves are nil — an interface that carries a non-nil type but a nil pointer value is not equal to `nil` (see Exercise 4).

Exercise 4 (Where is the bug?):

```go
package main

import "fmt"

type DBError struct{ code int }

func (e *DBError) Error() string { return fmt.Sprintf("db error %d", e.code) }

func connect(bad bool) error {
    var err *DBError
    if bad {
        err = &DBError{code: 500}
    }
    return err
}

func main() {
    e := connect(false)
    if e == nil {
        fmt.Println("connected OK")
```

```
    } else {
        fmt.Println("failed:", e)
    }
}
```

This is the classic nil-pointer-in-a-non-nil-interface trap. The program prints:

```
failed: <nil>
```

You probably expected `connected OK`. When `bad` is `false`, `err` stays a nil `*DBError` pointer. But `connect` returns `error` (an interface), so returning `err` wraps that typed nil pointer into an interface value whose **type** component is `*DBError` and whose **value** component is `nil`.

An interface is `nil` only when both components are nil. Here the type component is non-nil (`*DBError`), so `e == nil` is `false`, and the `else` branch runs. The printed value is `<nil>` because `fmt` calls the `Error()` method, which panics dereferencing the nil receiver; `fmt` recovers from that panic and, seeing that the argument is a nil pointer, prints `<nil>` instead.

The fix: do not declare the return holder as the concrete pointer type and then return it. Return an explicit `nil` (the untyped interface nil) on the success path:

```
func connect(bad bool) error {
    if bad {
        return &DBError{code: 500}
    }
    return nil
}
```

Now the success path returns a genuinely nil interface, the type component is nil, and `e == nil` is `true`.

Exercise 5 (Write a program): Define an interface `Shape` with two methods: `Area() float64` and `Perimeter() float64`. Implement `Shape` for two concrete types: `Rectangle` (with fields `Width` and `Height float64`) and `Circle` (with field `Radius float64`; use `math.Pi`). Write a function `printShapeInfo(s Shape)` that prints the area and perimeter. In `main`, create one `Rectangle` and one `Circle` and call `printShapeInfo` on each.

```
package main

import (
    "fmt"
    "math"
)

type Shape interface {
    Area() float64
    Perimeter() float64
}

type Rectangle struct {
    Width, Height float64
}

func (r Rectangle) Area() float64      { return r.Width * r.Height }
func (r Rectangle) Perimeter() float64 { return 2 * (r.Width + r.Height) }

type Circle struct {
    Radius float64
}
```

```go
func (c Circle) Area() float64      { return math.Pi * c.Radius * c.Radius }
func (c Circle) Perimeter() float64 { return 2 * math.Pi * c.Radius }

func printShapeInfo(s Shape) {
    fmt.Printf("area=%.2f perimeter=%.2f\n", s.Area(), s.Perimeter())
}

func main() {
    printShapeInfo(Rectangle{Width: 3, Height: 4})
    printShapeInfo(Circle{Radius: 5})
}
```

Output:

```
area=12.00 perimeter=14.00
area=78.54 perimeter=31.42
```

Both `Rectangle` and `Circle` define `Area` and `Perimeter` with value receivers, so values of either type satisfy `Shape` implicitly — no `implements` keyword needed. `printShapeInfo` accepts any `Shape`, and Go picks the right method via the interface's dynamic type at the call site. For the circle, `math.Pi * 5 * 5 = 78.54...` and `2 * math.Pi * 5 = 31.42...`, which match the `%.2f`-formatted output above.

Chapter 9: Error Handling — Answers

Exercise 1 (Think about it): Java uses checked exceptions to force callers to handle failures. Go returns `error` values that the compiler does not require you to inspect. What are the trade-offs of each approach? In what situations does Go's approach lead to more reliable code, and in what situations might it lead to less reliable code compared to Java's checked exceptions?

Java's checked exceptions make the compiler your partner: if a method declares `throws IOException`, every caller must either catch it or re-declare the throws clause. This guarantees that failure modes are documented in method signatures and that callers cannot silently ignore them — the code will not compile otherwise.

Go takes the opposite position: `error` is a return value like any other. The compiler does not prevent you from discarding it with `_` or simply not capturing it at all. The discipline must come from the programmer and tooling (`errcheck`, `staticcheck`) rather than the language itself.

Where Go tends to win:

- Error handling becomes regular control flow, not exception propagation through a separate, parallel mechanism. Errors can be stored in slices, combined with `errors.Join`, and processed with the same code that handles any other value.
- There is no checked-exception pollution: Java's `throws` clauses ripple upward through call chains, forcing every intermediate method to declare or re-wrap exceptions even when it has nothing meaningful to add. Go functions that merely forward an error just `return err` — no signature change required.
- Errors do not skip stack frames invisibly. The flow of control through a Go program is always traceable by reading the `if err != nil` checks; no hidden stack unwinding occurs.

Where Java's checked exceptions tend to win:

- The compiler catches ignored errors at the call site. A Go developer who writes `n, _ := parseTrackNumber(s)` has silently discarded the error, and the compiler says nothing.
- Checked exceptions create a discoverable, machine-readable contract: the method signature lists every failure mode. Go's error convention requires reading documentation or source code to learn what errors a function may return.

- Refactoring is easier in Java when you add a new failure mode: the compiler identifies every call site that needs updating. In Go, adding a new error condition to a function is invisible to callers.

The bottom line: Go trades compile-time enforcement for simplicity and composability. The approach works well in teams that run linters and review code carefully, and it shines in functions that produce or transform errors as data. It can lead to less reliable code in projects where error checking is informal or tooling is not enforced.

Exercise 2 (What does this print?):

```go
package main

import (
    "errors"
    "fmt"
)

var ErrNotFound = errors.New("not found")

type CatalogError struct {
    Track string
    Err   error
}

func (e *CatalogError) Error() string {
    return fmt.Sprintf("catalog: %s: %s", e.Track, e.Err)
}

func (e *CatalogError) Unwrap() error {
    return e.Err
}

func lookup(track string) error {
    return &CatalogError{Track: track, Err: ErrNotFound}
}

func main() {
    err := lookup("Insomnia")
    fmt.Println(err)
    fmt.Println(errors.Is(err, ErrNotFound))

    var ce *CatalogError
    if errors.As(err, &ce) {
        fmt.Println(ce.Track)
    }
}
```

Output:

```
catalog: Insomnia: not found
true
Insomnia
```

`lookup("Insomnia")` returns a `*CatalogError` with `Track = "Insomnia"` and `Err = ErrNotFound`.

`fmt.Println(err)` calls `err.Error()`, which returns `"catalog: Insomnia: not found"`. `fmt.Println` ap-

pends a newline, so the first line of output is `catalog: Insomnia: not found`.

`errors.Is(err, ErrNotFound)` starts at `err` (a `*CatalogError`) and compares it with `==` against `ErrNotFound` — no match. It then calls `err.Unwrap()`, which returns `ErrNotFound` itself. `ErrNotFound == ErrNotFound` is `true`. So `errors.Is` returns `true`, and `fmt.Println(true)` prints `true`.

`errors.As(err, &ce)` checks whether any error in the chain is assignable to `*CatalogError`. The outer error is exactly a `*CatalogError`, so `ce` is set to that value and `errors.As` returns `true`. The `if` body prints `ce.Track`, which is `"Insomnia"`.

Exercise 3 (Calculation): Consider the following code. For the input `Song{Title: "", Artist: "DJ Analyzer", Year: 2021, BPM: -1}`, how many sub-errors does the joined error returned by `validateSong` contain? What is the output of `fmt.Println(err)` for that input?

```go
package main

import (
    "errors"
    "fmt"
)

type Song struct {
    Title  string
    Artist string
    Year   int
    BPM    int
}

func validateSong(s Song) error {
    var errs []error
    if s.Title == "" {
        errs = append(errs, errors.New("title required"))
    }
    if s.Year < 2000 || s.Year > 2030 {
        errs = append(errs, fmt.Errorf("year %d out of range", s.Year))
    }
    if s.BPM <= 0 {
        errs = append(errs, errors.New("BPM must be positive"))
    }
    return errors.Join(errs...)
}

func main() {
    s := Song{Title: "", Artist: "DJ Analyzer", Year: 2021, BPM: -1}
    err := validateSong(s)
    fmt.Println(err)
}
```

Answer: The joined error contains **2** sub-errors.

Trace through the conditions for `Song{Title: "", Artist: "DJ Analyzer", Year: 2021, BPM: -1}`:

- `s.Title == ""` is `true` — `errors.New("title required")` is appended. (1 error)
- `s.Year < 2000 || s.Year > 2030`: `2021 < 2000` is `false` and `2021 > 2030` is `false`, so the condition is `false` — no error appended.
- `s.BPM <= 0`: `-1 <= 0` is `true` — `errors.New("BPM must be positive")` is appended. (2 errors)

`errors.Join` receives a slice of 2 non-nil errors. Its `Error()` method joins their messages with a newline between them.

Output:

```
title required
BPM must be positive
```

Note that `Artist` has no validation rule, so `"DJ Analyzer"` (a valid, non-empty value) does not contribute any error. `Year = 2021` falls within the range `[2000, 2030]`, so no year error is produced either.

Exercise 4 (Where is the bug?):

```go
package main

import (
    "fmt"
    "io"
    "strings"
)

func readAll(r io.Reader) ([]byte, error) {
    buf := make([]byte, 4)
    var result []byte
    for {
        n, err := r.Read(buf)
        result = append(result, buf[:n]...)
        if err == io.EOF {
            break
        }
        if err != nil {
            return nil, fmt.Errorf("readAll: %w", err)
        }
    }
    return result, nil
}

func main() {
    r := strings.NewReader("Children")
    data, err := readAll(r)
    if err != nil {
        fmt.Println("error:", err)
        return
    }
    fmt.Println(string(data))
}
```

The bug: The program compiles and runs as written (it prints `Children`), but it has one idiomatic-correctness bug.

The bug — comparing `err == io.EOF` directly instead of using `errors.Is`: The sentinel check `if err == io.EOF` happens to work for the bare `io.EOF` returned by `strings.Reader`, but it silently misses `io.EOF` if any reader in the future wraps it (e.g., `fmt.Errorf("read: %w", io.EOF)`). The chapter recommends never comparing errors with `==` when they might be wrapped. To be fair, the `io` package documents that `Read` implementations return the bare, unwrapped `io.EOF`, so on a direct `Read` call `==` is exactly what the standard

library itself does — `errors.Is` here is a robustness and consistency improvement rather than a correctness fix. The idiomatic fix is to use `errors.Is`, which walks the entire chain:

```
if errors.Is(err, io.EOF) {
    break
}
```

That fix also requires adding `"errors"` to the import list (the current program does not import it). With `errors.Is` in place the read loop is consistent with the rest of the chapter and robust to wrapped EOF.

Corrected `readAll`:

```
func readAll(r io.Reader) ([]byte, error) {
    buf := make([]byte, 4)
    var result []byte
    for {
        n, err := r.Read(buf)
        result = append(result, buf[:n]...)
        if errors.Is(err, io.EOF) {
            break
        }
        if err != nil {
            return nil, fmt.Errorf("readAll: %w", err)
        }
    }
    return result, nil
}
```

With the fix applied, the program prints:

```
Children
```

Exercise 5 (Write a program): Write a function `parseTimecode(s string) (int, int, error)` that parses a string in the format `"MM:SS"` (e.g., `"03:45"`) and returns the minutes, seconds, and `nil`. Return a descriptive error using `fmt.Errorf` if the string is not in the expected format, if either part is not a valid integer, or if minutes or seconds are out of range (minutes >= 0, seconds 0–59). Define a sentinel `var ErrInvalidTimecode = errors.New("invalid timecode")` and wrap it with `%w` in your error returns so that callers can use `errors.Is(err, ErrInvalidTimecode)`. In `main`, call `parseTimecode` with at least three inputs: one valid (`"03:45"`), one with a bad format (`"345"`), and one with an out-of-range second (`"01:61"`). Print the result or error for each.

```
package main

import (
    "errors"
    "fmt"
    "strconv"
    "strings"
)

var ErrInvalidTimecode = errors.New("invalid timecode")

func parseTimecode(s string) (int, int, error) {
    parts := strings.Split(s, ":")
    if len(parts) != 2 {
        return 0, 0, fmt.Errorf("parseTimecode: expected MM:SS, got %q: %w",
```

```
            s, ErrInvalidTimecode)
    }

    minutes, err := strconv.Atoi(parts[0])
    if err != nil {
        return 0, 0, fmt.Errorf("parseTimecode: invalid minutes %q: %w",
            parts[0], ErrInvalidTimecode)
    }

    seconds, err := strconv.Atoi(parts[1])
    if err != nil {
        return 0, 0, fmt.Errorf("parseTimecode: invalid seconds %q: %w",
            parts[1], ErrInvalidTimecode)
    }

    if minutes < 0 {
        return 0, 0, fmt.Errorf("parseTimecode: minutes %d is negative: %w",
            minutes, ErrInvalidTimecode)
    }
    if seconds < 0 || seconds > 59 {
        return 0, 0, fmt.Errorf("parseTimecode: seconds %d out of range [0,59]: %w",
            seconds, ErrInvalidTimecode)
    }

    return minutes, seconds, nil
}

func main() {
    inputs := []string{"03:45", "345", "01:61"}

    for _, tc := range inputs {
        m, s, err := parseTimecode(tc)
        if err != nil {
            fmt.Printf("%-10s => error: %s\n", tc, err)
            fmt.Printf("           is ErrInvalidTimecode: %v\n",
                errors.Is(err, ErrInvalidTimecode))
        } else {
            fmt.Printf("%-10s => %dm %ds\n", tc, m, s)
        }
    }
}
```

Output:

```
03:45      => 3m 45s
345        => error: parseTimecode: expected MM:SS, got "345": invalid timecode
           is ErrInvalidTimecode: true
01:61      => error: parseTimecode: seconds 61 out of range [0,59]: invalid timecode
           is ErrInvalidTimecode: true
```

Key design decisions explained:

- `ErrInvalidTimecode` is a package-level sentinel declared with `errors.New`. Exporting it (capital `E`) lets callers in other packages use `errors.Is` to distinguish timecode errors from other error kinds.
- Every error path uses `fmt.Errorf("...: %w", ErrInvalidTimecode)` to wrap the sentinel. This means

the returned error has a human-readable message that includes the context (the bad input, the specific reason) **and** a chain that `errors.Is` can walk to find `ErrInvalidTimecode`.

- The function returns three values: `(int, int, error)`, matching the requested signature. Both `int` values are zero on error, consistent with the Go convention of returning zero values alongside a non-nil error.
- `strings.Split(s, ":")` with a check on `len(parts) != 2` is the idiomatic way to parse a two-part format. Using `fmt.Sscanf` or a regex are also valid; `strings.Split` is the most readable for this simple case.

Chapter 10: Goroutines and Channels — Answers

Exercise 1 (Think about it): Java's `Thread` and `Runnable` model requires you to think about thread pool sizing. Go's goroutine model mostly frees you from this. Explain the runtime mechanism that makes goroutines cheap enough to use one per task. What cost, if any, do goroutines impose that Java threads do not, and when might you still want to limit the number of running goroutines?

The key mechanism is **M:N scheduling**: the Go runtime multiplexes M goroutines onto N OS threads, where N defaults to the number of CPU cores (`GOMAXPROCS`). The scheduler lives in user space, so switching between goroutines does not require a kernel mode transition — it is many times faster than a Java thread context switch.

Goroutines start with a small stack (around 2 KB) that grows dynamically as needed, up to a configurable maximum. Java threads allocate their full stack (typically 512 KB to 1 MB) up front from virtual memory. This means creating a million goroutines consumes roughly 2 GB of initial stack memory, while a million Java threads would require 500 GB to 1 TB. In practice the OS would refuse long before that.

Costs goroutines impose: Each goroutine is a heap-tracked object that the scheduler manages. At very high goroutine counts (hundreds of thousands) the scheduler itself becomes a bottleneck, and GC pressure rises because goroutine stacks live on the heap. There is also per-goroutine overhead in the runtime's internal bookkeeping.

When to still limit goroutines: Any time the goroutines drive downstream resource pressure — for example, goroutines that each open a database connection, a file descriptor, or an outbound HTTP request. Even when the goroutines themselves are cheap, the external resources they consume are not. The common Go idiom for bounding concurrency is a buffered channel used as a semaphore, or the worker-pool pattern.

Exercise 2 (What does this print?):

```go
package main

import "fmt"

func main() {
    ch := make(chan int, 3)

    ch <- 7
    ch <- 13
    ch <- 21
    close(ch)

    for v := range ch {
```

```
        fmt.Println(v)
    }

    v, ok := <-ch
    fmt.Println(v, ok)
}
```

Output:

```
7
13
21
0 false
```

Why:

The channel has capacity 3, so all three sends succeed without blocking — no receiving goroutine is needed. `close(ch)` marks the channel closed, but the three buffered values are still available to receive.

`range ch` drains the channel in FIFO order, printing `7`, `13`, and `21`. When the buffer is empty and the channel is closed, `range` terminates.

After the loop, `<-ch` receives from a channel that is both closed and empty. The comma-ok idiom returns the **zero value** of the element type (`0` for `int`) and `false` for `ok`, because the channel is exhausted. So `fmt.Println(v, ok)` prints `0 false`.

This demonstrates two rules: buffered values survive a `close`, and receiving from an empty closed channel always returns `(zero, false)` rather than blocking or panicking.

Exercise 3 (Calculation): Consider the following program. Trace its execution and determine the exact output. How many goroutines are alive (other than `main`) when the final `fmt.Println` in `main` runs?

```
package main

import "fmt"

func double(in <-chan int, out chan<- int) {
    for v := range in {
        out <- v * 2
    }
    close(out)
}

func main() {
    src := make(chan int, 3)
    dst := make(chan int, 3)

    src <- 3
    src <- 5
    src <- 8
    close(src)

    go double(src, dst)

    for result := range dst {
        fmt.Println(result)
    }
```

```
    fmt.Println("done")
}
```

Output:

```
6
10
16
done
```

Trace:

`main` fills `src` with `3`, `5`, `8` (the buffer of capacity 3 holds all three without blocking) and closes it. `go double(src, dst)` launches one goroutine that ranges over `src`, doubling each value and sending `6`, `10`, `16` to `dst`. Because `dst` also has capacity 3, all three doubled values fit in the buffer, so the `double` goroutine never blocks on send. After `src` is drained and closed, `double`'s `range` loop ends and it calls `close(out)` (which is `dst`).

`main` ranges over `dst`, receiving `6`, `10`, `16` in FIFO order, then the loop terminates once `dst` is empty and closed. Finally `main` prints `done`.

Goroutine count: By the time `range dst` has fully drained the channel, the `double` goroutine has already executed `close(out)` and returned. A returned goroutine is no longer alive. So when the final `fmt.Println("done")` runs, **zero** goroutines are alive other than `main`.

(Strictly, the scheduler may not have reclaimed the goroutine's bookkeeping the instant it returns, but logically it has finished — the only goroutine the program ever created has exited.)

Exercise 4 (Where is the bug?):

```go
package main

import (
    "fmt"
    "sync"
)

func main() {
    var wg sync.WaitGroup
    results := make(chan string, 3)
    tracks := []string{"Turn Me On", "Legend", "Escape"}

    for _, t := range tracks {
        go func() {
            wg.Add(1)
            defer wg.Done()
            results <- "Playing: " + t
        }()
    }

    wg.Wait()
    close(results)

    for r := range results {
        fmt.Println(r)
    }
}
```

The bug is that `wg.Add(1)` is called **inside** the goroutine instead of before launching it.

`wg.Wait()` runs in `main` concurrently with the three goroutines. There is no guarantee any goroutine has reached its `wg.Add(1)` before `main` calls `wg.Wait()`. If `Wait` observes the counter still at `0`, it returns immediately, `main` closes `results`, and the goroutines then panic with `panic: send on closed channel` when they try to send.

Running the program under `-race` reports a data race between `wg.Add` / the channel send and `close(results)`, and the program often panics or prints nothing.

The rule is: **call `wg.Add(n)` before starting the goroutines**, on the goroutine that owns the `WaitGroup`, never inside the goroutine being waited on.

Fixed version:

```
for _, t := range tracks {
    wg.Add(1)
    go func() {
        defer wg.Done()
        results <- "Playing: " + t
    }()
}
```

With `Add` hoisted out, the counter reaches 3 before `Wait` can return, so all three sends complete before `close(results)`, and the `range` loop prints all three lines (in nondeterministic order).

Note: the original `func()` literal capturing `t` is fine under Go 1.22+, where each loop iteration gets a fresh `t`. Under older Go you would also need to pass `t` as a parameter to avoid all goroutines seeing the last value.

Exercise 5 (Write a program): Write a program that launches three goroutines. Each goroutine sleeps for a different duration (10ms, 20ms, 30ms) and then sends one of the given strings on its own channel. In `main`, use a `select` loop to receive from all three channels and print each message as it arrives. Also add a `time.After(100 * time.Millisecond)` case that prints `"timeout"` and exits the loop if no message arrives within 100 ms of the last received one. Print the messages in the order they actually arrive.

```
package main

import (
    "fmt"
    "time"
)

func main() {
    c1 := make(chan string)
    c2 := make(chan string)
    c3 := make(chan string)

    go func() {
        time.Sleep(10 * time.Millisecond)
        c1 <- "Jaroslav Beck & Crispin: Legend"
    }()
    go func() {
        time.Sleep(20 * time.Millisecond)
        c2 <- "Jaroslav Beck: $100 Bills"
    }()
    go func() {
        time.Sleep(30 * time.Millisecond)
```

```
        c3 <- "Jaroslav Beck: Turn Me On"
    }()

    for {
        select {
        case msg := <-c1:
            fmt.Println(msg)
        case msg := <-c2:
            fmt.Println(msg)
        case msg := <-c3:
            fmt.Println(msg)
        case <-time.After(100 * time.Millisecond):
            fmt.Println("timeout")
            return
        }
    }
}
```

Output:

```
Jaroslav Beck & Crispin: Legend
Jaroslav Beck: $100 Bills
Jaroslav Beck: Turn Me On
timeout
```

How it works:

Each goroutine sleeps its own duration and then sends on its own unbuffered channel. The `select` in `main` blocks until one of the channels is ready, then runs that case and loops again. Because the sleeps are 10ms, 20ms, and 30ms apart, the messages arrive in that order.

The `time.After(100 * time.Millisecond)` case is re-created on **every** iteration of the loop because `select` re-evaluates its cases each pass. So the 100ms timer is reset after each received message and measures the gap since the last receive, exactly as required. After the third message, no further sends happen, so 100ms later the `time.After` case fires, prints `"timeout"`, and `return` exits `main`.

The three messages and `timeout` are deterministic here given the chosen sleep values; only the exact wall-clock timing varies. Verified clean under `go run -race`.

Chapter 11: Synchronization — Answers

Exercise 1 (Think about it): Java's `synchronized` keyword locks an object's monitor, which is built into every Java object. Go has no per-object monitor; instead you declare explicit `sync.Mutex` fields. What are the practical advantages and disadvantages of each approach? Consider: what happens when you need to protect two independent fields in the same struct, and how would you do it with each language's mechanism?

Java's per-object monitor is convenient for simple cases: every object already has a lock, so you can write `synchronized (this)` with no extra declarations. The downside is that the monitor is coarse-grained — there is only one per object. If a class has two independent fields that can be updated concurrently without affecting each other, locking the whole object monitor for either field creates unnecessary contention. Java programmers work around this with separate `java.util.concurrent.locks.Lock` objects or with dedicated inner lock objects:

```
private final Object tracksLock = new Object();
private final Object playsLock  = new Object();
```

```
synchronized (tracksLock) { tracks.add(track); }
synchronized (playsLock)  { plays.increment(); }
```

Go's approach makes the fine-grained case natural: you simply declare two independent mutex fields.

```
type Catalog struct {
    tracksMu sync.Mutex
    tracks   []string

    playsMu sync.Mutex
    plays   map[string]int
}
```

Each mutex protects only the field it is paired with, and neither blocks the other. This is less magic but more explicit.

The practical advantages of Go's approach:

- **Granularity:** you can have as many independent mutexes as you need at zero structural cost.
- **Clarity:** the pairing between a mutex and the data it protects is visible in the struct definition (often with a `// protects tracks` comment).
- **No accidental sharing:** in Java, every synchronized method on the same object uses the same monitor, even if they protect unrelated state. In Go, each mutex is independent by default.

The practical disadvantages:

- **Verbosity:** you must declare, name, and document each mutex. Java's implicit monitor requires no declaration.
- **Copy hazard:** Go structs are value types. Copying a struct that contains a `sync.Mutex` after first use is a bug; the struct must always be passed and stored by pointer. Java objects are always references, so this hazard does not exist (and `go vet` catches the Go version, as Exercise 4 shows).

Exercise 2 (What does this print?):

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    var once sync.Once
    var wg sync.WaitGroup
    results := make([]string, 3)

    for i := 0; i < 3; i++ {
        wg.Add(1)
        go func(n int) {
            defer wg.Done()
            once.Do(func() {
                results[n] = "loaded"
            })
            if results[n] == "" {
                results[n] = "skipped"
            }
        }(i)
```

```
    }

    wg.Wait()
    loaded := 0
    skipped := 0
    for _, r := range results {
        if r == "loaded" {
            loaded++
        } else if r == "skipped" {
            skipped++
        }
    }
    fmt.Printf("loaded=%d skipped=%d\n", loaded, skipped)
}
```

The output is always:

```
loaded=1 skipped=2
```

Here is why.

`sync.Once` guarantees that the function passed to `Do` runs **exactly once** across all goroutines. One of the three goroutines (whichever wins the race — the scheduler decides) executes `results[n] = "loaded"`, setting exactly one slot of the `results` slice.

The other two goroutines call `once.Do` as well, but their function bodies are silently dropped because the once is already done. They proceed past `once.Do` and check `results[n] == ""` for their own slot `n`. Because the winning goroutine wrote to a **different** index than these two, their slots are still empty, so they set `results[n] = "skipped"`.

The final tally is always exactly one `"loaded"` and two `"skipped"`, regardless of which goroutine wins the `once.Do` race.

Note: even though goroutines access different indices of `results` concurrently, this specific program is **not** a data race. Each goroutine writes only to its own `results[n]` (where `n` is passed by value), no two goroutines touch the same index, and `once.Do` establishes a happens-before edge for the one shared write. Running with `go run -race` reports no race.

Exercise 3 (Calculation): Consider the following program fragment:

```
var counter atomic.Int64
var wg sync.WaitGroup

for i := 0; i < 4; i++ {
    wg.Add(1)
    go func() {
        defer wg.Done()
        counter.Add(10)
    }()
}
wg.Wait()
fmt.Println(counter.Load())
```

(a) What value does `counter.Load()` always print, regardless of goroutine scheduling order?
(b) If `counter.Add(10)` were replaced by `counter.Add(int64(i))` (capturing `i` from the loop), what value would always be printed? Would your answer have differed in Go 1.21 or earlier, and if so, why?

(a) `counter.Load()` always prints `40`.

`atomic.Int64.Add` is an atomic read-modify-write operation. No matter what order the four goroutines execute, each `Add(10)` is applied to the current value atomically, and all four additions complete before `wg.Wait()` returns. The final value is always 4 x 10 = 40. This would **not** be guaranteed with a plain `int` counter and no synchronization — that would be a data race with unpredictable results.

(**b**) Replacing `counter.Add(10)` with `counter.Add(int64(i))` (capturing `i` directly from the loop) prints `6` under Go 1.22 and later.

Under Go 1.22+, each iteration of the `for` loop gets its **own** copy of the loop variable `i`. So the four goroutines capture distinct values 0, 1, 2, and 3, and the total is always 0 + 1 + 2 + 3 = 6, deterministically. Verified under go1.26.3: the program prints `6` on every run.

Would your answer have differed in Go 1.21 or earlier? Yes. Before Go 1.22, the loop variable `i` was a **single** variable shared across all iterations of the loop, including in a C-style three-clause `for` (the per-iteration change applies to both `for i := ...; ...; ... {}` loops and `for ... range` loops — not just range). With the old shared-variable semantics, the goroutines capture the **variable**, not its value at launch time. By the time a goroutine runs, the loop may already have advanced `i`. In the worst case all four goroutines observe `i == 4` (the value after the loop ends) and add 4 each, for a total of 16; in the best case they each capture a distinct value and total 6; intermediate totals are also possible. The result under Go 1.21 was non-deterministic, somewhere between 6 and 16.

The version-independent fix that works correctly on every Go version is to pass `i` as a parameter:

```go
go func(n int) {
    defer wg.Done()
    counter.Add(int64(n))
}(i) // pass i by value here
```

With this fix the result is always 0 + 1 + 2 + 3 = 6, on Go 1.21 and Go 1.22+ alike.

Note that the Go 1.22 per-iteration loop-variable change is gated on the `go` directive in your `go.mod`. You can confirm which loops the compiler rewrites with `go build -gcflags=-d=loopvar=2`, which prints `loop variable i now per-iteration, stack-allocated` for both three-clause and range loops.

Exercise 4 (Where is the bug?):

```go
package main

import (
    "fmt"
    "sync"
)

type SafeMap struct {
    mu sync.Mutex
    m  map[string]int
}

func NewSafeMap() SafeMap {
    return SafeMap{m: make(map[string]int)}
}

func (s SafeMap) Inc(key string) {
    s.mu.Lock()
    defer s.mu.Unlock()
    s.m[key]++
}
```

```
func (s SafeMap) Get(key string) int {
    s.mu.Lock()
    defer s.mu.Unlock()
    return s.m[key]
}

func main() {
    sm := NewSafeMap()
    var wg sync.WaitGroup
    for i := 0; i < 100; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            sm.Inc("Escape")
        }()
    }
    wg.Wait()
    fmt.Println(sm.Get("Escape"))
}
```

The bug is that `Inc` and `Get` have **value receivers** (`s SafeMap`), not pointer receivers (`s *SafeMap`).

When a method has a value receiver, Go passes a **copy** of the struct. Each call to `Inc` locks the mutex in its own private copy — a different mutex instance than the one in `sm` in `main`. The lock is acquired and released on a throwaway copy, providing no mutual exclusion on the real `sm`. One hundred goroutines therefore mutate the shared map concurrently without any synchronization.

The map header (`sm.m`) is a reference type, so the map operations do land on the same underlying map — but they are completely unprotected, and concurrent writes to a Go map are not allowed. Verified under go1.26.3: the program crashes with `fatal error: concurrent map writes`. `go vet` also flags the root cause statically: `Inc passes lock by value: command-line-arguments.SafeMap contains sync.Mutex` (and the same for `Get`).

The fix is to use pointer receivers throughout:

```
func (s *SafeMap) Inc(key string) {
    s.mu.Lock()
    defer s.mu.Unlock()
    s.m[key]++
}

func (s *SafeMap) Get(key string) int {
    s.mu.Lock()
    defer s.mu.Unlock()
    return s.m[key]
}
```

And have the constructor return a pointer so no copy is ever made:

```
func NewSafeMap() *SafeMap {
    return &SafeMap{m: make(map[string]int)}
}

func main() {
    sm := NewSafeMap() // *SafeMap; no copy
    // ...
```

```
    sm.Inc("Escape")
    // ...
    fmt.Println(sm.Get("Escape")) // 100
}
```

With pointer receivers and a pointer variable, every call to `Inc` and `Get` locks the **same** mutex, and the output is reliably `100`.

Exercise 5 (Write a program): Implement a concurrent-safe `RateLimiter` struct that uses a `sync.Mutex` to protect a counter and a `time.Time` field tracking when the window resets. The struct should have a method `Allow(n int) bool` that returns `true` if `n` tokens are available in the current one-second window, deducting them if so, and `false` otherwise (without deducting). Write a `main` function that launches 10 goroutines, each calling `Allow(1)` in a loop 5 times, and prints how many calls were allowed versus denied across all goroutines combined. Use `sync.WaitGroup` to wait for all goroutines to finish.

```
package main

import (
    "fmt"
    "sync"
    "sync/atomic"
    "time"
)

// RateLimiter allows at most limit tokens per one-second window.
type RateLimiter struct {
    mu      sync.Mutex
    limit   int
    used    int
    resetAt time.Time
}

func NewRateLimiter(limit int) *RateLimiter {
    return &RateLimiter{
        limit:   limit,
        resetAt: time.Now().Add(time.Second),
    }
}

// Allow returns true and deducts n tokens if they are available.
// It returns false without deducting if the window is exhausted.
func (r *RateLimiter) Allow(n int) bool {
    r.mu.Lock()
    defer r.mu.Unlock()

    now := time.Now()
    if now.After(r.resetAt) {
        r.used = 0
        r.resetAt = now.Add(time.Second)
    }

    if r.used+n > r.limit {
        return false
    }
```

```
    r.used += n
    return true
}

func main() {
    limiter := NewRateLimiter(25) // allow 25 calls per second
    var wg sync.WaitGroup
    var allowed, denied atomic.Int64

    for i := 0; i < 10; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            for j := 0; j < 5; j++ {
                if limiter.Allow(1) {
                    allowed.Add(1)
                } else {
                    denied.Add(1)
                }
            }
        }()
    }

    wg.Wait()
    fmt.Printf("allowed=%d denied=%d total=%d\n",
        allowed.Load(), denied.Load(), allowed.Load()+denied.Load())
}
```

Sample output (limit 25, 50 total calls; the 50 calls all land in the same one-second window):

```
allowed=25 denied=25 total=50
```

Verified under go1.26.3, including with `go run -race`: no data race is reported and the totals are stable across runs.

Key points of the implementation:

- The `sync.Mutex` in `RateLimiter` protects both `used` and `resetAt` together as a single invariant. Neither field is read or written outside the lock.
- `Allow` checks the current time inside the lock so that the window reset and the token deduction are one atomic decision. If the check and the deduction were in separate lock acquisitions, another goroutine could sneak in between them.
- The `allowed` and `denied` counters in `main` use `atomic.Int64` rather than a mutex because they are independent single-variable updates — a textbook atomic use case.
- `wg.Add(1)` is called in the outer loop, **before** the goroutine is launched, following the `WaitGroup` rule from the chapter (and the bug in Exercise 6). You could equally use `wg.Go(func() { ... })` (Go 1.25+) and drop the explicit `Add/Done`.

Exercise 6 (Where is the bug?):

```
package main

import (
    "fmt"
    "sync"
)
```

```go
func main() {
    var wg sync.WaitGroup
    results := make([]int, 5)
    for i := 0; i < 5; i++ {
        go func(n int) {
            wg.Add(1)
            defer wg.Done()
            results[n] = n * n
        }(i)
    }
    wg.Wait()
    fmt.Println(results)
}
```

The author expects `[0 1 4 9 16]` but usually sees something like `[0 0 0 0 0]`, and `go run -race` reports a data race.

The bug: `wg.Add(1)` is called *inside* the goroutine instead of before launching it.

`wg.Wait()` blocks only while the counter is greater than zero. But the counter is incremented inside each goroutine, and the main goroutine reaches `wg.Wait()` immediately after the `for` loop dispatches the five goroutines — before any of them have necessarily run. If `Wait` observes the counter at zero (which it can, because no goroutine has executed `wg.Add(1)` yet), it returns instantly, and `main` prints `results` while the goroutines are still writing to it.

This produces two distinct problems:

- **Premature `Wait`:** the program usually prints a partial or all-zero slice such as `[0 0 0 0 0]` instead of `[0 1 4 9 16]`. The output is non-deterministic and changes run to run.
- **Data race:** `main` reads `results` (via `fmt.Println`) at the same time the surviving goroutines write to `results[n]`. Verified under go1.26.3: `go run -race` reports `WARNING: DATA RACE`, and `go vet` flags it statically with the exact diagnostic `WaitGroup.Add called from inside new goroutine`.

The fix: call `wg.Add(1)` in the main goroutine *before* the `go` statement, so the counter is guaranteed non-zero before `Wait` can observe it:

```go
for i := 0; i < 5; i++ {
    wg.Add(1)
    go func(n int) {
        defer wg.Done()
        results[n] = n * n
    }(i)
}
wg.Wait()
fmt.Println(results) // [0 1 4 9 16]
```

With `Add` moved out, `Wait` cannot return until all five `Done` calls have happened, which establishes a happens-before relationship so the subsequent read of `results` is race-free.

Even cleaner, Go 1.25 added the `WaitGroup.Go` helper, which runs a function in a new goroutine and handles the matching `Add`/`Done` automatically, sidestepping this whole class of mistake:

```go
for i := 0; i < 5; i++ {
    wg.Go(func() {
        results[i] = i * i // i is per-iteration under Go 1.22+
    })
}
```

```
wg.Wait()
fmt.Println(results) // [0 1 4 9 16]
```

Verified under go1.26.3: this `WaitGroup.Go` version compiles and prints `[0 1 4 9 16]`.

Chapter 12: Context and Concurrency Patterns — Answers

Exercise 1 (Think about it): In Java, cancelling an in-flight operation typically means calling `Future.cancel(true)` or interrupting a thread via `Thread.interrupt()`. Describe how Go's `context.Context` model differs from Java's thread-interrupt approach. What are the advantages of passing a context explicitly rather than relying on a thread-level interrupt mechanism? Consider what happens when a Java thread is blocked in a third-party library that does not handle `InterruptedException`, compared to how a Go function using a context-aware library would behave.

Java's thread-interrupt model is **implicit and cooperative at the thread level**. When you call `Thread.interrupt()`, a flag is set on the thread, and blocking calls like `Object.wait()`, `Thread.sleep()`, and some `java.io.InputStream.read()` implementations throw `InterruptedException` when they notice it. But not every blocking operation checks the flag: a thread blocked in a native call, a third-party lock, or a legacy `InputStream` implementation may never see the interrupt at all. The interrupt propagates up the call stack only as long as every layer catches and re-throws (or re-sets) the flag, which is notoriously easy to accidentally swallow:

```
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    // oops, swallowed it; the interrupt flag is now cleared
}
```

Go's `context.Context` is **explicit and uniform**. Every function that can be cancelled accepts a `context.Context` parameter, conventionally as its first argument. Cancellation is communicated by closing `ctx.Done()`, which is observable without any thread-local state. Any function that calls another context-aware function simply passes the same context through; the propagation is visible in every function signature.

The advantages over thread interrupts are:

1. **Explicit propagation.** You can see in the function signature that a function is cancellable. In Java, there is no signature-level signal that a method checks `Thread.interrupted()`.
2. **Deadlines and timeouts as first-class values.** `context.WithTimeout` and `context.WithDeadline` associate a deadline with the context object itself, not with a thread. Multiple goroutines can share the same context and respect the same deadline without any shared mutable state.
3. **No accidental swallowing.** Because `ctx.Done()` is a channel, you either select on it or you do not — there is no exception to catch and accidentally discard.
4. **Composability.** Derived contexts (`WithCancel`, `WithTimeout`) form a tree. Cancelling a parent automatically cancels all children. Java's thread-interrupt model is flat: each thread has exactly one interrupt flag.
5. **Request-scoped values.** `context.WithValue` lets you attach metadata (trace IDs, auth tokens) to a context and retrieve it anywhere in the call tree without global state.

If a Java thread is blocked in a third-party library that does not handle `InterruptedException` — for example, a legacy JDBC driver — calling `Thread.interrupt()` may have no effect. The thread stays blocked, and the only recourse is to close the underlying socket from another thread or wait for the operation to time out at the OS level. A Go function calling a database driver built on top of `database/sql` passes a context

to `db.QueryContext`; the driver layer itself monitors `ctx.Done()` and closes the connection if the context is cancelled. The library author opts in once; all callers benefit automatically.

Exercise 2 (What does this print?):

```go
package main

import (
    "context"
    "fmt"
    "time"
)

func work(ctx context.Context, label string) {
    select {
    case <-time.After(500 * time.Millisecond):
        fmt.Println(label, "done")
    case <-ctx.Done():
        fmt.Println(label, "cancelled:", ctx.Err())
    }
}

func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 200*time.Millisecond)
    defer cancel()

    go work(ctx, "Flaming June")
    go work(ctx, "Saltwater")
    time.Sleep(400 * time.Millisecond)
    fmt.Println("main done")
}
```

Output (order of the first two lines may vary):

```
Flaming June cancelled: context deadline exceeded
Saltwater cancelled: context deadline exceeded
main done
```

The context has a 200 ms timeout. Both `work` goroutines are launched immediately and block in their `select` statement waiting for either `time.After(500ms)` or `ctx.Done()`. After 200 ms the timeout fires, `ctx.Done()` is closed, and both goroutines unblock on the `ctx.Done()` case. Each prints its label with `"cancelled: context deadline exceeded"`. The goroutines finish well before `main`'s `time.Sleep(400ms)` elapses, so `"main done"` appears last.

The two cancelled lines may appear in either order because goroutine scheduling is not deterministic. `main done` always appears last because `time.Sleep(400ms)` is longer than the 200 ms timeout plus the goroutines' response time.

Exercise 3 (Calculation): You run a worker pool with `workers = 3` and feed it a slice of 7 tasks. Each task takes exactly 100 ms. Assuming no overhead and perfect parallelism, how many milliseconds does the pool take to complete all 7 tasks? Show your work: how many rounds of 3 concurrent workers are needed and what does each round contribute?

Answer: 300 ms.

With 3 workers processing tasks that each take 100 ms:

Round	Tasks processed	Wall-clock time elapsed
1	tasks 1, 2, 3	0 – 100 ms
2	tasks 4, 5, 6	100 – 200 ms
3	task 7 (+ 2 idle)	200 – 300 ms

Round 1 dispatches tasks 1–3 in parallel. All three finish at T=100 ms. Round 2 dispatches tasks 4–6 in parallel; all finish at T=200 ms. Round 3 dispatches task 7 alone (only one task remains); it finishes at T=300 ms.

Total elapsed time = ceil(7 / 3) x 100 ms = 3 x 100 ms = **300 ms**.

General formula: `ceil(N / workers) x task_duration`.

Note that the third round still takes a full 100 ms even though only one worker is busy: wall-clock time is governed by the slowest (in this case the only) task in the round, so 2 workers sitting idle does not shorten it.

Exercise 4 (Where is the bug?):

```go
package main

import (
    "context"
    "fmt"
    "time"
)

func fetchData(url string) <-chan string {
    ch := make(chan string)
    go func() {
        time.Sleep(2 * time.Second)
        ch <- "result for " + url
    }()
    return ch
}

func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 500*time.Millisecond)
    defer cancel()

    ch := fetchData("https://example.com/songs")
    select {
    case result := <-ch:
        fmt.Println(result)
    case <-ctx.Done():
        fmt.Println("timed out")
    }
}
```

The bug: goroutine leak in `fetchData`.

`fetchData` launches a goroutine that sleeps for 2 seconds and then sends on the unbuffered `ch`. When the context times out after 500 ms, `main` exits the `select` via `ctx.Done()` and prints `"timed out"`. At this point

`ch` is no longer being read by anyone. The goroutine inside `fetchData` is still sleeping; when it wakes up at T=2 s and tries to send `ch <- "result for ..."`, it blocks forever because nobody will ever receive from `ch`. The goroutine is leaked — it will never exit. A `go.uber.org/goleak` check confirms the goroutine is still parked on `time.Sleep` after `main` returns.

The fix: pass the context into `fetchData` so the goroutine can bail out early, and buffer the channel so the send never blocks.

```
func fetchData(ctx context.Context, url string) <-chan string {
    ch := make(chan string, 1) // buffered so the goroutine can send even if nobody reads
    go func() {
        select {
        case <-time.After(2 * time.Second):
            ch <- "result for " + url // send result if we finish in time
        case <-ctx.Done():
            // context was cancelled; exit cleanly without sending
        }
    }()
    return ch
}

func main() {
    ctx, cancel := context.WithTimeout(context.Background(), 500*time.Millisecond)
    defer cancel()

    ch := fetchData(ctx, "https://example.com/songs")
    select {
    case result := <-ch:
        fmt.Println(result)
    case <-ctx.Done():
        fmt.Println("timed out")
    }
}
```

Using a buffered channel of capacity 1 also guards against a secondary leak: if the result arrives at the same instant `main`'s `select` exits via the `ctx.Done()` branch (a narrow race), the goroutine can still send on `ch` without blocking, and then exit. With this fix, `goleak.VerifyNone` reports no leaked goroutines.

Exercise 5 (Write a program): Implement a function `fanOutFetch(ctx context.Context, songs []string) ([]string, error)` that uses `errgroup` to fetch all song titles concurrently. Simulate each fetch with a `time.Sleep` of a random duration between 50 and 150 ms (use `math/rand`). If any fetch takes longer than 300 ms total (enforced by a timeout on the context passed to `fanOutFetch`), the entire operation should be cancelled and an error returned. Print either all results in order or the cancellation error.

```
package main

import (
    "context"
    "fmt"
    "math/rand"
    "time"

    "golang.org/x/sync/errgroup"
)
```

```go
// fanOutFetch fetches all song titles concurrently using errgroup.
// Each fetch is simulated with a random sleep between 50 and 150 ms.
// The function returns the titles in the same order as songs, or an error
// if the context is cancelled before all fetches complete.
func fanOutFetch(ctx context.Context, songs []string) ([]string, error) {
    results := make([]string, len(songs))
    g, ctx := errgroup.WithContext(ctx)

    for i, song := range songs {
        i, song := i, song // capture for Go < 1.22; harmless on 1.22+
        g.Go(func() error {
            delay := time.Duration(50+rand.Intn(100)) * time.Millisecond
            select {
            case <-time.After(delay):
                results[i] = "fetched: " + song
                return nil
            case <-ctx.Done():
                return ctx.Err()
            }
        })
    }

    if err := g.Wait(); err != nil {
        return nil, err
    }
    return results, nil
}

func main() {
    songs := []string{
        "From The Start",
        "Bewitched",
        "Too Sweet",
        "Work Song",
    }

    ctx, cancel := context.WithTimeout(context.Background(), 300*time.Millisecond)
    defer cancel()

    results, err := fanOutFetch(ctx, songs)
    if err != nil {
        fmt.Println("error:", err)
        return
    }
    for _, r := range results {
        fmt.Println(r)
    }
}
```

Explanation:

`errgroup.WithContext` derives a new context from the one passed in. If any goroutine returns a non-nil error, `errgroup` cancels that derived context, causing all other goroutines that are still sleeping to unblock on `ctx.Done()` and return `ctx.Err()`. `g.Wait()` returns the first non-nil error. Each goroutine writes to a distinct index `results[i]`, so there is no data race — this passes `go run -race` cleanly. Because each goroutine writes

only its own slot, the results stay in the same order as `songs` without any locking.

Because the outer `context.WithTimeout` fires after 300 ms, any fetch whose random delay exceeds the remaining budget is cancelled. Fetches with delays in the 50–150 ms range all complete well within 300 ms under normal conditions; lower the timeout (for example to 20 ms) to reliably trigger a cancellation in testing.

Sample output when all fetches succeed:

```
fetched: From The Start
fetched: Bewitched
fetched: Too Sweet
fetched: Work Song
```

Sample output when the timeout fires:

```
error: context deadline exceeded
```

Note that `errgroup` lives in the `golang.org/x/sync/errgroup` package, not the standard library. Add it with `go get golang.org/x/sync/errgroup`.

Exercise 6 (What does this print?):

```go
package main

import (
    "context"
    "fmt"
)

type ctxKey string

func main() {
    const userKey ctxKey = "user"

    ctx := context.Background()
    ctx = context.WithValue(ctx, userKey, "ana")
    ctx = context.WithValue(ctx, ctxKey("user"), "beto")

    fmt.Println(ctx.Value(userKey))
    fmt.Println(ctx.Value("user"))
}
```

Output:

```
beto
<nil>
```

A context key is matched by **both its type and its value**, not by value alone. `userKey` is a `ctxKey` whose value is `"user"`, and `ctxKey("user")` is also a `ctxKey` with value `"user"` — they are the *same key*. So the second `WithValue` shadows the first: looking up that key returns the most recently attached value, `"beto"`. (`Value` walks from the innermost derived context outward, so the last value attached for a given key wins.)

The final line passes a plain `string` literal `"user"`, whose type is `string`, not `ctxKey`. Because the key types differ, no match is found anywhere in the tree, so `ctx.Value("user")` returns `nil`, which prints as `<nil>`. This is exactly why idiomatic Go uses an unexported key type: a `string` key from another package can never collide with your typed key.

Note: this program uses a named string-based key type purely to demonstrate the type-matching rule; `go vet` does not flag it because the key type is a defined (named) type rather than a built-in type. In production,

prefer an unexported empty-struct key type as shown in the chapter.

Chapter 13: Packages and Modules — Answers

Exercise 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`?

MVS works by computing, for each dependency, the maximum of the minimum versions requested across the whole module graph. If module A requires `library v1.2.0` and module B requires `library v1.3.0`, Go selects `v1.3.0` — the smallest version that satisfies both requirements. No module ever silently gets a version newer than the one its author actually tested against, unless someone explicitly asks for an upgrade.

Problems MVS avoids:

- **Silent upgrades.** In Maven's nearest-wins model, adding one new dependency can quietly pull in a newer (or older) version of a transitive library and break unrelated code. MVS never introduces a version you did not ask for.
- **Non-reproducible builds.** Because MVS is deterministic and the exact versions plus hashes are recorded in `go.mod` and `go.sum`, two developers checking out the same commit get bit-for-bit identical dependencies. Maven and Gradle resolution can drift depending on what is cached or what repositories are reachable.

What MVS makes harder:

- **Staying current.** MVS deliberately resists upgrading. If your graph pins a library at `v1.2.0`, you stay there until someone runs `go get library@v1.4.0`. In a large organization this means security patches can go unnoticed unless you actively check.
- **Downgrading.** To use an older version than the graph currently requires, you have to downgrade or remove every module that requires the newer one, because MVS always takes the maximum of the minimums.

Surprise from `go get pkg@latest`: The newly upgraded module may itself require newer versions of its own transitive dependencies. MVS will then bump those transitives up to whatever the new `pkg` requires, so a single `go get` can ripple through `go.mod` and change versions you never named. Running `go mod tidy` afterward and reviewing the diff in `go.mod` and `go.sum` is the habit that keeps these surprises visible.

Exercise 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`:

```go
package detail

import "fmt"

func Show() {
    fmt.Println("internal detail")
}
```

File `main.go`:

```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.

The build fails.

The rule for `internal` is: an `internal` package may only be imported by code rooted at the directory that is the *parent* of `internal/`. Here the package lives at `github.com/angoscia/demo/lyrics/internal/detail`, so the parent of `internal/` is `github.com/angoscia/demo/lyrics`. Only code under `github.com/angoscia/demo/lyrics/...` is allowed to import it.

`main.go` sits at the module root, `github.com/angoscia/demo`, which is *not* under `github.com/angoscia/demo/lyrics`, so the import is rejected. The import of the public `github.com/angoscia/demo/lyrics` package is fine; only the `internal/detail` import is the problem.

Running `go build ./...` produces:

```
package github.com/angoscia/demo
    main.go:5:5: use of internal package
        github.com/angoscia/demo/lyrics/internal/detail not allowed
```

To fix it without changing the import semantics, either move `detail` to `internal/detail` directly under the module root (so its parent is the module root, which `main.go` can reach), or move `main.go` into a directory under `lyrics/` (so it is rooted under `lyrics`).

Exercise 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?

MVS selects `golang.org/x/text v0.14.0`.

Collect every minimum requirement for `golang.org/x/text` across the graph:

- the root `app` module requires `v0.14.0` (the `// indirect` line),
- `audio v1.4.0` requires `v0.12.0`,
- `catalog v0.9.2` requires `v0.14.0`.

MVS takes the *maximum of those minimums*, which is `v0.14.0`. It is the smallest version that satisfies all three requirements at once: it is at least `v0.12.0` (so `audio` is happy) and at least `v0.14.0` (so `catalog` and the root are happy). Note that "minimum version selection" does not mean "pick the lowest version present"; it means "pick the lowest version that still meets every requirement," which is the highest of the listed minimums.

After adding a dependency that requires `golang.org/x/text v0.16.0`: the set of minimums becomes `{v0.14.0, v0.12.0, v0.14.0, v0.16.0}`, and the maximum is now `v0.16.0`. So MVS selects `golang.org/x/text v0.16.0`.

Exercise 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)
}
```

Output:

```
f called
init, a = 5
main, a = 5
```

Go does not initialise package-level variables top to bottom; it initialises them in *dependency order*. A variable is initialised only after every variable it references has been initialised, and ties are broken by declaration order.

- `a` depends on `b` and `c`, so it cannot go first.
- `b` depends on `f()`; `f()` has no variable dependencies, so `b` can be initialised, which calls `f()` and prints `f called`. `b` becomes `3`.
- `c` depends on nothing, so it becomes `2`.
- `a` now has both operands ready: `a = b + c = 3 + 2 = 5`.

After all package-level variables are initialised, Go runs `init()`, which prints `init, a = 5`. Finally `main` runs and prints `main, a = 5`. The key point is that even though `a` is declared first in the source, it is initialised last because of its dependencies, and `f` is called exactly once during variable initialisation, before `init` and `main`.

Exercise 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/`.

The build fails inside `player/`.

The `internal` rule is enforced by import path, and it does not care about module boundaries — if anything, crossing a module boundary makes it stricter. The parent of `internal/` here is

`github.com/djcobra/betteroffalone`, so only code rooted under `github.com/djcobra/betteroffalone/...` may import `github.com/djcobra/betteroffalone/internal/config`.

Inside the `betteroffalone` module, `main.go` is rooted at `github.com/djcobra/betteroffalone`, so its import is fine. The `player` module is a *different* module (`github.com/djcobra/player`); none of its code is under `github.com/djcobra/betteroffalone`, so importing that module's `internal/config` is forbidden.

`go build ./...` inside `player/` produces:

```
package github.com/djcobra/player
    main.go:5:5: use of internal package
        github.com/djcobra/betteroffalone/internal/config not allowed
```

The bug is that `player` is trying to reach into another module's `internal` tree, which the toolchain refuses by design — `internal` packages are private to the module (and subtree) that contains them.

How to fix it without moving `config` out of `internal/`: keep `config` exactly where it is and instead expose what `player` needs through a *public* package in `betteroffalone`. For example, add `github.com/djcobra/betteroffalone/region/region.go` (a non-`internal` package) that re-exports `DefaultRegion`:

```go
package region

import "github.com/djcobra/betteroffalone/internal/config"

const DefaultRegion = config.DefaultRegion
```

That re-export is legal because `region` is itself rooted under `github.com/djcobra/betteroffalone`, so it may import the `internal` package. Then `player/main.go` imports the public `github.com/djcobra/betteroffalone/region` instead of the internal path. `config` stays private, and `player` only ever touches the public surface you deliberately chose to expose.

Exercise 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.

Here is a complete implementation that builds and runs under go1.26.3.

`go.mod`:

```
module github.com/robertdreamhouse/children

go 1.26
```

`tracks/tracks.go`:

```go
package tracks

type Track struct {
    Title  string
    Artist string
}

var Catalog = []Track{
    {Title: "Childhood Dreams", Artist: "Robert Dreamhouse"},
    {Title: "Paper Houses", Artist: "Robert Dreamhouse"},
}
```

`internal/format/format.go`:

```go
package format

import (
    "fmt"

    "github.com/robertdreamhouse/children/tracks"
)

func Label(t tracks.Track) string {
    return fmt.Sprintf("%s by %s", t.Title, t.Artist)
}
```

`main.go`:

```go
package main

import (
    "fmt"

    "github.com/robertdreamhouse/children/internal/format"
    "github.com/robertdreamhouse/children/tracks"
)

func main() {
    for _, t := range tracks.Catalog {
        fmt.Println(format.Label(t))
    }
}
```

The `internal/format` import is allowed here because `main.go` sits at the module root. Its module path `github.com/robertdreamhouse/children` is the parent of `internal/`, which is exactly what the internal-package rule requires. Running `go run ./...` prints:

```
Childhood Dreams by Robert Dreamhouse
Paper Houses by Robert Dreamhouse
```

`Label` is exported (capital `L`) so other packages *inside* the module can call it, but because it lives under `internal/`, no code outside `github.com/robertdreamhouse/children` can import it — exactly the "exported within the module, private outside it" behaviour the exercise asked for.

Chapter 14: Essential Standard Library — Answers

Exercise 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?

The key difference is that Java's hierarchy distinguishes between byte-oriented I/O (`InputStream`/`OutputStream`) and character-oriented I/O (`Reader`/`Writer`), giving four root types as a result. Go does not make that split at the interface level: `io.Reader` and `io.Writer` always deal in `[]byte`. Character encoding is handled separately — either at the edges (for example `bufio.Scanner`, which returns `string` tokens) or by explicit conversion. This simplification is possible because Go treats `string` and `[]byte` as first-class, cheaply convertible types, so the language does not need a parallel hierarchy to make text feel natural.

The composition functions (`io.TeeReader`, `io.MultiWriter`, and friends) are ordinary functions that return an interface value. In Java the same decorators are abstract classes (`FilterInputStream`, `BufferedInputStream`) because the language needed a concrete supertype to share implementation; Go can express the same patterns with lightweight wrappers because interfaces are structural.

What is harder in Go's model:

- Seeking and positioning. Java's `RandomAccessFile` supports `seek` directly. Go separates this into `io.Seeker` (a third interface) and requires callers to do a type assertion or accept an `io.ReadSeeker` parameter.
- Buffered reads with unread/pushback. Java's `PushbackInputStream` is a first-class class. In Go you use `bufio.Reader.UnreadByte()` or `bufio.Reader.UnreadRune()`, which requires wrapping in `bufio` first.
- Encoding-aware text I/O. Java's `InputStreamReader` bridges bytes to characters with a named charset. In Go you must use third-party packages (for example `golang.org/x/text/encoding`) or write the conversion yourself.

Exercise 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),
    )
}
```

Output:

```
level=INFO msg="scan complete" lines=3 elapsed=0s
```

Step by step:

1. A `slog.TextHandler` is created writing to `os.Stdout`. The `ReplaceAttr` function strips the `time` attribute, so no timestamp appears.
2. `strings.NewReader` wraps the literal string as an `io.Reader`, and `bufio.NewScanner` wraps that reader.
3. The scanner splits on newlines (the default `ScanLines`). The input has three lines (`"Café Del Mar"`, `"Zombie"`, `"Crazy Train"`) each terminated by a newline. `Scan` returns `true` three times and then `false` at EOF, so `count` ends up as `3`.
4. `logger.Info` emits a text-format log line. The `time` key is suppressed by `ReplaceAttr`. `slog.Int("lines", 3)` formats as `lines=3`. `slog.Duration("elapsed", 0)` formats as `elapsed=0s` — a zero `time.Duration` formats as `"0s"`.

The key ordering in `log/slog` text format is `level`, `msg`, then the attributes in the order they were passed.

Exercise 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?

The file is 100,000 lines times 100 bytes = 10,000,000 bytes, or about 9.5 MiB.

(a) `os.ReadFile`

`os.ReadFile` reads the entire file into a single `[]byte`. Peak heap allocation: about 9.5 MiB (the whole file in one slice). If you then process the result into strings or split on newlines, you may double or triple the allocation. This is the simplest approach but the most memory-hungry for large files.

(b) `bufio.NewScanner`

`Scanner` uses an internal buffer (default starting size 4 KiB, default maximum token size 64 KiB). It reads the file in chunks, scanning for newline boundaries. At any instant only the current chunk plus the current token are in memory. Peak heap allocation: about 64 KiB (the scanner's internal buffer) plus the length of the longest individual line. For 100-byte lines this is well under 1 MiB.

(c) `io.Copy(io.Discard, f)`

`io.Copy` uses a single 32 KiB copy buffer when neither side implements a fast path. Peak heap allocation: about 32 KiB. (In fact `io.Discard` implements `io.ReaderFrom`, so `io.Copy` takes that fast path, skips the 32 KiB buffer entirely, and uses a small pooled buffer — even less memory.) However, this approach does not count lines — it just discards all bytes.

Best for counting lines without storing content: `bufio.NewScanner`.

`io.Copy(io.Discard, f)` uses the least memory but cannot count lines without inspecting the bytes. `bufio.NewScanner` counts lines with a constant-size buffer (under 1 MiB peak) and is the idiomatic Go

choice. `os.ReadFile` uses the most memory and should be avoided for large files when you only need a count.

Exercise 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]`))
}
```

The bug: `regexp.MustCompile(pattern)` is called inside the `for` loop, so the pattern is recompiled on every iteration. With four strings this is merely wasteful, but inside a hot path processing millions of records it becomes a serious performance problem — `regexp.MustCompile` parses the pattern, builds a finite automaton, and allocates memory each time.

The output is correct (it prints `4` — all four titles start with an uppercase ASCII letter), so this is a performance bug, not a logic bug.

The fix: compile the pattern once, before the loop. If the pattern is constant, hoist it to a package-level variable:

```
var startsUpperRE = regexp.MustCompile(`^[A-Z]`)

func countMatches(texts []string) int {
    total := 0
    for _, t := range texts {
        if startsUpperRE.MatchString(t) {
            total++
        }
    }
    return total
}
```

If the pattern is a runtime parameter, compile it once before the loop and return an error if the pattern is invalid:

```
func countMatches(texts []string, pattern string) (int, error) {
    re, err := regexp.Compile(pattern)
    if err != nil {
```

```
        return 0, fmt.Errorf("invalid pattern %q: %w", pattern, err)
    }
    total := 0
    for _, t := range texts {
        if re.MatchString(t) {
            total++
        }
    }
    return total, nil
}
```

Note the switch from `MustCompile` to `Compile` with a returned error — caller-supplied patterns should never use `MustCompile`, because a bad pattern would crash the program. `MustCompile` is reserved for compile-time-constant patterns where a bad pattern is a programmer error, not a user error.

Exercise 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`.

```
package main

import (
    "flag"
    "fmt"
    "io/fs"
    "log/slog"
    "os"
    "path/filepath"
    "strings"
)

func main() {
    dir     := flag.String("dir",     ".",   "directory to search")
    ext     := flag.String("ext",     ".go", "file extension to count")
    verbose := flag.Bool("verbose",   false, "log each matching file")
    flag.Parse()

    logger := slog.New(slog.NewTextHandler(os.Stderr, nil))

    count := 0
    err := filepath.WalkDir(*dir, func(path string, d fs.DirEntry, err error) error {
        if err != nil {
            return err
        }
        if !d.IsDir() && strings.HasSuffix(d.Name(), *ext) {
            count++
            if *verbose {
                logger.Info("match", slog.String("file", path))
            }
        }
        return nil
    })
```

```
    if err != nil {
        logger.Error("walk failed", slog.Any("error", err))
        os.Exit(1)
    }

    fmt.Printf("found %d %s file(s) in %s\n", count, *ext, *dir)
}
```

Sample runs (note that the `slog` line goes to stderr and the summary to stdout):

```
$ go run main.go -dir . -ext .go -verbose
time=2026-06-04T13:41:55.983-07:00 level=INFO msg=match file=main.go
found 1 .go file(s) in .

$ go run main.go -dir /usr/local/go/src -ext .go
found 8412 .go file(s) in /usr/local/go/src
```

Key points in the solution:

- `flag.Parse()` is called at the start of `main`, after all flag variables are defined, so all flags are parsed before use.
- `slog.New(slog.NewTextHandler(os.Stderr, nil))` writes structured logs to stderr, leaving stdout clean for program output.
- `filepath.WalkDir` is preferred over `filepath.Walk` because it passes an `fs.DirEntry`, which avoids an extra `os.Stat` call per entry.
- `!d.IsDir()` excludes directories, so a directory named `tools.go` is not counted; `d.Name()` is the base name of the entry, which is all that matters for an extension match.
- The error from `WalkDir` is checked and reported; a non-nil error returned from the callback halts the walk.

Chapter 15: JSON, HTTP, and the Web — Answers

Exercise 1 (Think about it): In Java with Spring MVC or JAX-RS, you annotate a class method with `@GetMapping("/songs/{id}")` or `@GET @Path("/songs/{id}")` and the framework discovers handlers via reflection and classpath scanning. In Go, you call `mux.HandleFunc("GET /songs/{id}/", getSong)` explicitly in `main`. What are the tradeoffs of each approach? Consider startup time, debuggability, IDE navigation, and what happens when two handlers are registered for the same pattern.

Annotation/reflection-based frameworks (Spring, JAX-RS):

- *Startup time:* The framework scans the classpath, processes annotations, and builds a routing table at startup. For large applications this can add seconds — sometimes tens of seconds. Spring Boot's startup time is a well-known pain point for serverless and container workloads.
- *IDE navigation:* IDEs understand Spring annotations deeply; `@GetMapping` provides clickable navigation to the handler. However, reconstructing the full request path often means tracing through a chain of `@RequestMapping` annotations on the class, the method, and any inherited base classes.
- *Debuggability:* Routing bugs can be subtle. The framework discovers handlers at runtime, so a typo in a path annotation compiles cleanly and only fails when a request is made. Error messages from annotation-driven frameworks can be verbose and hard to relate back to a specific source line.
- *Duplicate pattern:* Spring raises a startup-time exception (for example, an ambiguous-mapping error) when two methods map to the same path and method.

Explicit registration (Go `ServeMux`):

- *Startup time:* Registration happens in `main` — it is just function calls. There is no scanning, so startup overhead is negligible.

- *IDE navigation:* In `mux.HandleFunc("GET /songs/{id}/", getSong)`, `getSong` is a direct function reference. Your IDE can jump to it with a single click, with no framework-specific plugin needed.
- *Debuggability:* The routing table is built from ordinary Go code. If you register the wrong path, you can add a `fmt.Println` or set a breakpoint in `main` and see exactly what is registered.
- *Duplicate pattern:* The Go 1.22 `ServeMux` panics at registration time if two patterns conflict. This is a startup crash rather than a silent routing bug, which is the right trade-off — it catches the mistake before any request is served.

The Go approach is more explicit and has less magic, at the cost of writing each registration by hand. The annotation approach is more convenient in large teams where developers add handlers across many files and rely on the framework to assemble the routing table. Neither is universally better; the right choice depends on team size, application complexity, and how much framework overhead you are willing to accept.

Exercise 2 (What does this print?):

```go
package main

import (
    "encoding/json"
    "fmt"
)

type Artist struct {
    Name    string `json:"name"`
    Country string `json:"country,omitempty"`
    Secret  string `json:"-"`
}

func main() {
    a := Artist{Name: "Chicane", Country: "", Secret: "UK"}
    data, _ := json.Marshal(a)
    fmt.Println(string(data))

    var b Artist
    json.Unmarshal([]byte(`{"name":"Darude","secret":"Finland"}`), &b)
    fmt.Printf("Name: %s, Secret: %q\n", b.Name, b.Secret)
}
```

Output:

```
{"name":"Chicane"}
Name: Darude, Secret: ""
```

First `Println`: `a.Country` is `""`, the zero value for `string`. The tag `json:"country,omitempty"` tells `encoding/json` to omit the `country` field when its value is empty, so it disappears from the output. `a.Secret` is `"UK"`, but the tag `json:"-"` instructs the encoder to always skip this field regardless of its value. The result is `{"name":"Chicane"}` — only `name` survives.

Second `Printf`: The JSON input contains a `"secret"` key. However, the struct field is `Secret string \json:“-”“. Thejson:“-”tag meansencoding/jsonignores this field during both marshalling **and** unmarshalling. The“secret”key in the JSON is silently discarded, sob.Secretkeeps its zero value““.b.Nameis correctly set to”Darude”from the“name”“ key.`

`%q` formats a string with Go double-quote syntax, so an empty string prints as `""`.

Exercise 3 (Calculation): Consider the following `ServeMux` registration and the three incoming requests. For each request, state which handler function is called. If no handler runs, give the HTTP error status the mux returns, and say whether the mux failed to match the path at all (`404 Not Found`) or matched the path pattern but not the request method (`405 Method Not Allowed`).

```
mux := http.NewServeMux()
mux.HandleFunc("GET /tracks/",       listTracks)
mux.HandleFunc("GET /tracks/{id}/",  getTrack)
mux.HandleFunc("POST /tracks/",      createTrack)
```

a. `GET /tracks/` — **`listTracks`**

The method is `GET` and the path is exactly `/tracks/`. The pattern `GET /tracks/` ends in a slash, so it matches the subtree rooted at `/tracks/`, including the exact path `/tracks/`. The pattern `GET /tracks/{id}/` requires at least one non-empty segment for `{id}` between the slashes (for example `/tracks/42/`), so it does not match `/tracks/` on its own. `listTracks` is called.

b. `GET /tracks/42/` — **`getTrack`**

The method is `GET` and the path is `/tracks/42/`. The pattern `GET /tracks/{id}/` matches, with `{id}` capturing `42`. A more specific pattern wins over the less specific `GET /tracks/`, so `getTrack` is chosen. Inside the handler, `r.PathValue("id")` returns `"42"`.

c. `DELETE /tracks/7/` — **`405 Method Not Allowed`**

The path `/tracks/7/` matches the pattern `GET /tracks/{id}/`, but that pattern only accepts `GET`, and the request method is `DELETE`. Because some registered pattern matches the path while none matches the method, the Go 1.22 `ServeMux` returns `405 Method Not Allowed` rather than `404`. No registered handler runs.

The mux also sets an `Allow` header listing the methods that are valid for that path. Verified with `httptest`, the response is `405` with `Allow: GET, HEAD, POST`. `HEAD` appears automatically because a registered `GET` pattern also serves `HEAD`, and `POST` appears because `POST /tracks/` is registered for the same subtree.

Exercise 4 (Where is the bug?):

```go
package main

import (
    "fmt"
    "io"
    "net/http"
)

func fetchLyrics(url string) (string, error) {
    resp, err := http.Get(url)
    if err != nil {
        return "", err
    }
    body, err := io.ReadAll(resp.Body)
    if err != nil {
        return "", err
    }
    return string(body), nil
}

func main() {
    lyrics, err := fetchLyrics("https://api.example.com/lyrics/sandstorm")
```

```
    if err != nil {
        fmt.Println("error:", err)
        return
    }
    fmt.Println(lyrics)
}
```

The bug: `resp.Body` is never closed.

When `http.Get` succeeds, `resp.Body` is a live network stream wrapped as an `io.ReadCloser`. If the function reads the body but never calls `resp.Body.Close()`, the underlying TCP connection is not returned to the connection pool — it is leaked. Under load, a program making many requests will exhaust its file descriptors and connection pool, eventually causing new HTTP requests to fail. Note that the early-return error path after `io.ReadAll` (`return "", err`) leaks the body too.

The fix:

```
func fetchLyrics(url string) (string, error) {
    resp, err := http.Get(url)
    if err != nil {
        return "", err
    }
    defer resp.Body.Close() // close on every return, success or error

    body, err := io.ReadAll(resp.Body)
    if err != nil {
        return "", err
    }
    return string(body), nil
}
```

Placing `defer resp.Body.Close()` immediately after the `err` check ensures the body is closed on every path out of the function, including early returns. This is the canonical Go idiom for HTTP client response bodies.

A secondary point worth noting: this code never checks `resp.StatusCode`. A `404` or `500` response still returns a non-nil `resp` with `err == nil`, so the "lyrics" you get back may actually be an error page. In production you would also check `resp.StatusCode` before trusting the body, but the connection leak is the load-bearing bug here.

Exercise 5 (Write a program): Build a small in-memory HTTP server that manages a list of songs. Define a `Song` struct with fields `ID int`, `Title string`, and `Artist string`, all with appropriate `json` tags. Store songs in a package-level `map[int]Song`. Register two routes using a `http.NewServeMux()`:

- `GET /songs/` returns all songs as a JSON array.
- `GET /songs/{id}/` returns the single song with that ID, or HTTP 404 if not found.

Pre-populate the map with two songs (use Darude or Chicane tracks). Start the server on `:8080`.

```
package main

import (
    "encoding/json"
    "net/http"
    "strconv"
)

type Song struct {
```

```
    ID     int    `json:"id"`
    Title  string `json:"title"`
    Artist string `json:"artist"`
}

var catalog = map[int]Song{
    1: {ID: 1, Title: "Saltwater", Artist: "Chicane"},
    2: {ID: 2, Title: "Sandstorm", Artist: "Darude"},
}

func listSongs(w http.ResponseWriter, r *http.Request) {
    songs := make([]Song, 0, len(catalog))
    for _, s := range catalog {
        songs = append(songs, s)
    }
    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(songs)
}

func getSong(w http.ResponseWriter, r *http.Request) {
    id, err := strconv.Atoi(r.PathValue("id"))
    if err != nil {
        http.Error(w, "invalid id", http.StatusBadRequest)
        return
    }
    song, ok := catalog[id]
    if !ok {
        http.Error(w, "not found", http.StatusNotFound)
        return
    }
    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(song)
}

func main() {
    mux := http.NewServeMux()
    mux.HandleFunc("GET /songs/",      listSongs)
    mux.HandleFunc("GET /songs/{id}/", getSong)
    http.ListenAndServe(":8080", mux)
}
```

Testing the server (with `curl` in a second terminal):

```
$ curl http://localhost:8080/songs/
[{"id":2,"title":"Sandstorm","artist":"Darude"},{"id":1,"title":"Saltwater","artist":"Chicane"}]

$ curl http://localhost:8080/songs/1/
{"id":1,"title":"Saltwater","artist":"Chicane"}

$ curl http://localhost:8080/songs/99/
not found
```

Key points illustrated by this solution:

- `json.NewEncoder(w).Encode(songs)` streams the JSON directly to the `http.ResponseWriter` without allocating an intermediate `[]byte`.

- `r.PathValue("id")` retrieves the wildcard captured by `{id}` in the Go 1.22 pattern.
- `strconv.Atoi` converts the path segment to an integer; a malformed segment returns `400 Bad Request` instead of panicking.
- The `Content-Type` header is set before the body is written. Headers must be set before the first `Write` (or `Encode`) call — once the body starts, the headers are already sent and cannot be changed.
- Map iteration order is random (Chapter 7), so the array in `/songs/` may come back in either order. In a real service you would sort the result before encoding it to give clients a stable response.

Chapter 16: gRPC — Answers

Exercise 1 (Think about it): For a public API consumed by third-party web and mobile clients you do not control, versus high-volume internal traffic between your own services, which would you reach for — REST/JSON or gRPC — and why?

Public, third-party API: REST/JSON.

- *Browser support:* Browsers speak HTTP/1.1 and JSON natively. gRPC needs HTTP/2 trailers that browsers do not expose to JavaScript, so it requires gRPC-Web plus a translating proxy — friction you are pushing onto every consumer.
- *Debuggability:* Anyone can hit a JSON endpoint with `curl`, a browser, or Postman and read the response. A binary protobuf payload is opaque without the `.proto` and a decoder.
- *Adoption:* Third parties do not want to compile your `.proto` or pull a generated client just to call you. A documented JSON contract is the lowest barrier to entry.

Internal, high-volume service-to-service traffic: gRPC.

- *Payload size and speed:* Binary protobuf is smaller and faster to (de)serialize than text JSON, which matters at high request rates.
- *Schema evolution:* The `.proto` is a compile-time contract for every language; a renamed field is caught at generation time, not at 3 a.m. in production. Tag numbers let you add fields without breaking old peers.
- *Streaming and deadlines:* gRPC has first-class streaming and propagates `context` deadlines across the network for free — both awkward to bolt onto plain REST.

Many systems do both: gRPC between internal services, with a REST/JSON edge (often via `grpc-gateway`) for the public face. The right answer is "REST at the boundary, gRPC in the core," not one universally.

Exercise 2 (What does this do?): A teammate writes this proto and regenerates, then is surprised the change "broke" old clients:

```
// before
message Song {
  string id    = 1;
  string title = 2;
}
// after
message Song {
  string id     = 2;
  string title  = 1;
}
```

They only swapped the tag numbers, not the field names or types. What happens when a new server sends a `Song` to a client built from the *old* proto, and why?

Protobuf identifies fields on the wire by their **tag number**, never by their name. A new server built from the "after" proto encodes `id` under tag 2 and `title` under tag 1. An old client, built from the "before" proto, decodes tag 1 as `id` and tag 2 as `title`. So the old client reads the server's `title` into its `id` field and the server's `id` into its `title` field: the two values are silently swapped.

Crucially, there is no error. Both fields are `string`, so the bytes decode cleanly — the data is just wrong. (Had the swapped fields been different types, the client would instead see decode errors or garbage values.) The lesson: tag numbers *are* the contract. You may rename a field freely, but you must never reuse or renumber a tag.

Exercise 3 (Calculation): A unary handler does 80 ms of work; the client calls it with a 50 ms timeout, and the handler never checks `ctx.Done()`.

(a) The client observes **`codes.DeadlineExceeded`**. At 50 ms the client's context fires and the RPC is cancelled locally, regardless of what the server is still doing.

(b) **Yes** — the handler runs all 80 ms. Cancellation in Go is cooperative (Chapter 12): a context that is "done" does nothing on its own; code must check it. Since this handler ignores `ctx`, it finishes, and its response is then thrown away because the client already gave up and the stream is closed.

(c) Make the handler **honor the context** — check `ctx.Err()` (or `select` on `ctx.Done()`) at a cancellation point and return early, e.g.:

```
if err := ctx.Err(); err != nil {
    return nil, status.FromContextError(err).Err()  // DeadlineExceeded / Canceled
}
```

and pass `ctx` to any downstream calls (database, other RPCs) so they cancel too.

Exercise 4 (Where is the bug?):

```
func (s *jukeboxServer) ListSongs(
    req *musicpb.ListRequest,
    stream grpc.ServerStreamingServer[musicpb.Song],
) error {
    for _, song := range s.catalog {
        stream.Send(song)
    }
    return errors.New("done sending")
}
```

Bug 1 — the `Send` error is ignored. If the client disconnects or its deadline fires partway through, `stream.Send` starts returning an error, but the loop keeps calling `Send` into a dead stream and never notices. It should be `if err := stream.Send(song); err != nil { return err }`. *Client effect:* wasted work, and a real send failure is swallowed instead of surfaced.

Bug 2 — it returns a non-`nil` error to end the stream. A server-streaming handler ends the stream cleanly by returning `nil`; returning any error terminates the RPC with a failure status. Worse, `errors.New("done sending")` carries no gRPC status, so gRPC labels it `codes.Unknown`. *Client effect:* after receiving every song, the client's `Recv` returns that `Unknown`-coded error instead of the expected `io.EOF`, so a perfectly good response looks like a failure. The fix is to `return nil`.

Exercise 5 (Write a program): Define a `.proto` with a `Library` service exposing `AddSongs(stream Song) returns (Summary)` where `Summary` has an `int32 count` and an `int32 total_bpm`. Implement the server so it accumulates the count and the sum of all `bpm` fields across the streamed songs, then returns the summary

with `SendAndClose`. Write a client that streams three songs and prints the returned count and average BPM. Use `status.Errorf(codes.InvalidArgument, ...)` if any streamed song has an empty `id`.

The proto:

```
syntax = "proto3";
package library.v1;
option go_package = "example/librarypb";

message Song {
  string id     = 1;
  string title  = 2;
  string artist = 3;
  int32  bpm    = 4;
}

message Summary {
  int32 count     = 1;
  int32 total_bpm = 2;
}

service Library {
  rpc AddSongs(stream Song) returns (Summary);
}
```

The server accumulates the count and BPM sum, rejecting any song with an empty `id`:

```
func (s *libraryServer) AddSongs(
    stream grpc.ClientStreamingServer[librarypb.Song, librarypb.Summary],
) error {
    var count, total int32
    for {
        song, err := stream.Recv()
        if err == io.EOF {
            return stream.SendAndClose(&librarypb.Summary{Count: count, TotalBpm: total})
        }
        if err != nil {
            return err
        }
        if song.GetId() == "" {
            return status.Errorf(codes.InvalidArgument, "song with empty id")
        }
        count++
        total += song.GetBpm()
    }
}
```

The client streams three songs and prints the count and average:

```
stream, err := client.AddSongs(ctx)
if err != nil {
    log.Fatal(err)
}
songs := []*librarypb.Song{
    {Id: "1", Title: "Monaco", Artist: "Bad Bunny", Bpm: 130},
    {Id: "2", Title: "Despecha", Artist: "Rosalia", Bpm: 130},
    {Id: "3", Title: "Todo De Ti", Artist: "Rauw Alejandro", Bpm: 92},
```

```
}
for _, song := range songs {
    if err := stream.Send(song); err != nil {
        log.Fatal(err)
    }
}
sum, err := stream.CloseAndRecv()
if err != nil {
    log.Fatal(err)
}
avg := float64(sum.GetTotalBpm()) / float64(sum.GetCount())
fmt.Printf("%d songs, avg %.1f BPM\n", sum.GetCount(), avg)
// 3 songs, avg 117.3 BPM
```

The key moves: the client `Send`s in a loop and finishes with `CloseAndRecv`; the server `Recv`s until `io.EOF` and replies once with `SendAndClose`.

Chapter 17: Database Access — Answers

Exercise 1 (Think about it): JDBC requires explicit transaction management and connection pooling through a `DataSource`, usually provided by an application server or a library like HikariCP. Go's `database/sql` builds connection pooling directly into `sql.DB`. What are the tradeoffs of each approach? In what situations might you still want an external connection pool in a Go application?

Go's approach is simpler for the common case: you call `sql.Open`, tune a few settings (`SetMaxOpenConns`, `SetMaxIdleConns`, `SetConnMaxLifetime`), and the pool manages itself. There is no additional dependency, no configuration file, and no separate object to wire up. This is consistent with Go's philosophy of including batteries for common needs.

JDBC's reliance on an external pool (HikariCP, DBCP, c3p0, or an application server pool) adds setup complexity but provides more configurability. HikariCP, for example, offers connection validation queries, connection test-on-borrow, metric integration with Micrometer, and health check endpoints.

In a Go application you might still want an external or proxy pool in a few situations:

- **PgBouncer / ProxySQL:** These are database-side proxy pools that multiplex many application connections onto fewer server connections. They are useful when you have many application instances and the database itself limits total connections. `sql.DB`'s pool operates within one process; PgBouncer aggregates across many processes.
- **Serverless / short-lived processes:** If your Go binary starts and exits quickly (a CLI, a Lambda function), the in-process pool provides little benefit. A proxy pool keeps connections warm across many cold starts.
- **Observability:** Some proxy pools offer detailed query-level metrics and slow-query logging that are difficult to achieve from application code alone.

In most long-running Go services, the built-in pool is sufficient and external pooling adds unnecessary complexity.

Exercise 2 (What does this print?):

```
package main

import (
    "database/sql"
```

```
    "fmt"
)

func main() {
    a := sql.Null[string]{V: "J'ai pas vingt ans !", Valid: true}
    b := sql.Null[string]{V: "Gouryella", Valid: false}
    c := sql.Null[int64]{V: 0, Valid: false}

    fmt.Println(a.Valid, a.V)
    fmt.Println(b.Valid, b.V)
    fmt.Println(c.Valid, c.V)
}
```

Output:

```
true J'ai pas vingt ans !
false Gouryella
false 0
```

`sql.Null[T]` is a plain struct with two exported fields: `V` (the value) and `Valid` (a bool). It has no logic in its fields; they are whatever you set them to.

- `a` has `Valid: true` and `V: "J'ai pas vingt ans !"`, so `fmt.Println` prints `true J'ai pas vingt ans !`.
- `b` has `Valid: false` but `V` is still `"Gouryella"` — setting `Valid` to `false` does not zero out `V`. This might be surprising: the struct remembers the value even though it would represent `NULL` in the database. `fmt.Println` prints `false Gouryella`.
- `c` has `Valid: false` and `V: 0` (the zero value for `int64`). `fmt.Println` prints `false 0`.

The key takeaway: `Valid` controls whether the value is considered non-NULL; it does not affect what is stored in `V`. When scanning from a database, `Scan` sets `V` to the zero value and `Valid` to `false` for a `NULL` column.

Exercise 3 (Calculation): Trace the following transaction sequence and state whether the database ends up with the row inserted or not, and why.

```
tx, _ := db.BeginTx(ctx, nil)
defer tx.Rollback()

_, err := tx.ExecContext(ctx, "INSERT INTO songs (title, artist) VALUES (?, ?)",
    "Sounds of Slashdot", "San Mehat")
if err != nil {
    return err
}

return tx.Commit()
```

Case A: `ExecContext` succeeds and `Commit` succeeds.

The deferred `tx.Rollback()` fires after `tx.Commit()` returns. `Rollback` on a committed transaction is a no-op — it returns `sql.ErrTxDone`, which is silently discarded because the return value of a deferred call is not used here. The row is **inserted and committed** permanently.

Case B: `ExecContext` succeeds but `Commit` returns an error.

`tx.Commit()` fails, so the function returns an error. The deferred `tx.Rollback()` then fires. However, when a commit fails at the database level, the transaction is typically already rolled back by the database. `Rollback` here is a safety net that confirms the abort. The row is **not inserted** — the transaction was not committed.

Case C: `ExecContext` returns an error.

The `if err != nil { return err }` branch fires, returning the error. The deferred `tx.Rollback()` fires before the function returns to the caller. The `INSERT` is undone (or was never applied, depending on the database). The row is **not inserted**.

In all three cases the deferred rollback provides a guarantee: the transaction is always cleaned up, regardless of which path the function takes. This is the entire point of the deferred rollback pattern.

Exercise 4 (Where is the bug?):

```go
func getArtistSongs(ctx context.Context, db *sql.DB, artist string) ([]string, error) {
    rows, err := db.QueryContext(ctx,
        "SELECT title FROM songs WHERE artist = ?", artist)
    if err != nil {
        return nil, err
    }

    var titles []string
    for rows.Next() {
        var title string
        if err := rows.Scan(&title); err != nil {
            return nil, err
        }
        titles = append(titles, title)
    }
    return titles, nil
}
```

Two bugs:

Bug 1: `rows` is never closed. If `rows.Next()` completes the loop normally, `rows` is closed automatically. But if `rows.Scan` returns an error and the function returns early via `return nil, err`, `rows` is never closed. The connection borrowed from the pool is never returned, leaking it. Under sustained load this exhausts the pool.

Bug 2: `rows.Err()` is never checked. After the loop, `rows.Err()` may hold an error that caused iteration to stop early (e.g., a network failure mid-result-set). Ignoring it means the function silently returns a partial result as if it were complete.

Fixed version:

```go
func getArtistSongs(ctx context.Context, db *sql.DB, artist string) ([]string, error) {
    rows, err := db.QueryContext(ctx,
        "SELECT title FROM songs WHERE artist = ?", artist)
    if err != nil {
        return nil, err
    }
    defer rows.Close() // always close, even on early return

    var titles []string
    for rows.Next() {
        var title string
        if err := rows.Scan(&title); err != nil {
            return nil, err
        }
        titles = append(titles, title)
    }
```

```
    if err := rows.Err(); err != nil { // check for iteration errors
        return nil, err
    }
    return titles, nil
}
```

`defer rows.Close()` immediately after checking the error from `QueryContext` is the standard pattern. Calling `rows.Close()` on already-closed rows is a no-op, so it is always safe.

Exercise 5 (Write a program): Using `database/sql` and the `github.com/mattn/go-sqlite3` driver, open an in-memory SQLite database, create a `playlists` table, insert at least three rows inside a single transaction using the deferred rollback pattern, query and print all rows with `QueryContext` and `Scan`, and use `sql.Null[string]` for a nullable `description` column.

```
package main

import (
    "context"
    "database/sql"
    "fmt"
    "log"

    _ "github.com/mattn/go-sqlite3"
)

type Playlist struct {
    ID          int
    Name        string
    Owner       string
    Description sql.Null[string]
}

func main() {
    db, err := sql.Open("sqlite3", ":memory:")
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()

    ctx := context.Background()

    if err := db.PingContext(ctx); err != nil {
        log.Fatal(err)
    }

    _, err = db.ExecContext(ctx, `
        CREATE TABLE playlists (
            id          INTEGER PRIMARY KEY AUTOINCREMENT,
            name        TEXT    NOT NULL,
            owner       TEXT    NOT NULL,
            description TEXT
        )`)
    if err != nil {
        log.Fatal(err)
```

```
    }

    // insert three rows inside a single transaction
    if err := insertPlaylists(ctx, db); err != nil {
        log.Fatal(err)
    }

    // query and print all rows
    rows, err := db.QueryContext(ctx,
        "SELECT id, name, owner, description FROM playlists ORDER BY id")
    if err != nil {
        log.Fatal(err)
    }
    defer rows.Close()

    for rows.Next() {
        var p Playlist
        if err := rows.Scan(&p.ID, &p.Name, &p.Owner, &p.Description); err != nil {
            log.Fatal(err)
        }
        desc := "(no description)"
        if p.Description.Valid {
            desc = p.Description.V
        }
        fmt.Printf("%d: %s by %s --- %s\n", p.ID, p.Name, p.Owner, desc)
    }
    if err := rows.Err(); err != nil {
        log.Fatal(err)
    }
}

func insertPlaylists(ctx context.Context, db *sql.DB) error {
    tx, err := db.BeginTx(ctx, nil)
    if err != nil {
        return err
    }
    defer tx.Rollback() // no-op if Commit succeeds

    stmt, err := tx.PrepareContext(ctx,
        "INSERT INTO playlists (name, owner, description) VALUES (?, ?, ?)")
    if err != nil {
        return err
    }
    defer stmt.Close()

    playlists := []Playlist{
        {
            Name:        "Trance Essentials",
            Owner:       "gamemaster",
            Description: sql.Null[string]{V: "Gouryella and friends", Valid: true},
        },
        {
            Name:        "Slashdot Sounds",
            Owner:       "gamemaster",
```

```
            Description: sql.Null[string]{V: "San Mehat sets", Valid: true},
        },
        {
            Name:        "Late Night Mix",
            Owner:       "alizee_fan",
            Description: sql.Null[string]{}, // NULL description
        },
    }

    for _, p := range playlists {
        if _, err := stmt.ExecContext(ctx, p.Name, p.Owner, p.Description); err != nil {
            return err
        }
    }

    return tx.Commit()
}
```

Output:

```
1: Trance Essentials by gamemaster --- Gouryella and friends
2: Slashdot Sounds by gamemaster --- San Mehat sets
3: Late Night Mix by alizee_fan --- (no description)
```

Key points demonstrated:

- `sql.Open` + `PingContext` verifies connectivity before doing any work.
- The transaction uses the **deferred rollback pattern**: `defer tx.Rollback()` is unconditional; `Commit` at the end makes it a no-op on success.
- A prepared statement is created once inside the transaction and reused for each insert.
- `sql.Null[string]` handles the nullable `description` column; `Valid: false` results in a `NULL` stored in the database and scanned back correctly.
- `rows.Err()` is checked after the loop to catch any mid-stream errors.

Note that the SQLite driver requires `CGO_ENABLED=1` (the default on most systems with a C compiler installed) because `github.com/mattn/go-sqlite3` links against the C SQLite library.

Chapter 18: Generics — Answers

Exercise 1 (Think about it): Java generics use **type erasure**: at runtime, `List<String>` and `List<Integer>` are both just `List`. Generic type information is only available at compile time. Go generics use **monomorphization** (or a shared pointer-shaped representation): the compiler may generate distinct code for each instantiation. Describe one concrete advantage and one concrete disadvantage of each approach. How does type erasure affect what you can do with a Java generic type at runtime (e.g., `instanceof List<String>`)? Does Go have the same limitation?

Type erasure (Java):

Advantage: A single compiled class file handles all instantiations. `ArrayList<String>` and `ArrayList<Integer>` share bytecode, which keeps the compiled output compact and means that libraries compiled against an older JDK are forward-compatible with new generic code without recompilation.

Disadvantage: The generic type argument is gone at runtime. You cannot write `obj instanceof List<String>` — the JVM sees only `List`. Creating a generic array (`new T[]`) is illegal because the

runtime cannot know the element type. Working around these limitations requires unchecked casts and `@SuppressWarnings("unchecked")`, which reintroduces the `ClassCastException` risk that generics were designed to prevent.

Monomorphization (Go):

Advantage: The compiler generates type-specific code, so operations on value types like `int` or `float64` are never boxed. A `Stack[int]` stores plain `int` values directly in the backing slice — no `Integer` wrapper objects, no extra allocations, no GC pressure. There is no unchecked cast at runtime; type safety is total.

Disadvantage: The compiler may produce multiple instantiations of the same function, increasing binary size. For large programs with many instantiation combinations, compile times can grow. (In practice, Go mitigates this by using a GC-shape-based approach that shares code for pointer-shaped types, but the tradeoff is still present for value types.)

Runtime reflection:

In Java, because of erasure, `List<String>.class` does not exist; you can only get `List.class`. `instanceof List<String>` is a compile-time warning and a runtime check against `List`, not `List<String>`. Accessing the actual type argument at runtime requires passing a `Class<T>` token explicitly.

In Go, you can use `reflect.TypeOf` to inspect the concrete type of a value, and since Go does not erase type information the way Java does, the concrete type is always available. However, Go reflection operates on concrete values, not on type parameters themselves — you cannot query "what was `T`?" from inside a generic function without passing the type explicitly. Both languages have some limitations here, but the flavor of the limitation differs.

Exercise 2 (What does this print?):

```go
package main

import "fmt"

func Filter[T any](s []T, keep func(T) bool) []T {
    var out []T
    for _, v := range s {
        if keep(v) {
            out = append(out, v)
        }
    }
    return out
}

type BPM int

func main() {
    beats := []BPM{72, 128, 96, 140, 80}
    fast := Filter(beats, func(b BPM) bool { return b >= 120 })
    fmt.Println(fast)

    words := []string{"Escape", "J'ai pas vingt ans !", "J'en ai marre !", "$100 Bills"}
    long := Filter(words, func(s string) bool { return len(s) > 7 })
    fmt.Println(long)
}
```

Output:

```
[128 140]
[J'ai pas vingt ans ! J'en ai marre ! $100 Bills]
```

For the first call, `T` is inferred as `BPM`. The predicate keeps elements greater than or equal to 120. `72`, `96`, and `80` are below 120 and are excluded. `128` and `140` pass and are appended to `out`.

For the second call, `T` is inferred as `string`. The predicate keeps strings whose length in bytes is greater than 7. `"Escape"` has 6 bytes (excluded), `"J'ai pas vingt ans !"` has 20 bytes (included), `"J'en ai marre !"` has 15 bytes (included), and `"$100 Bills"` has 10 bytes (included). Note that `len` on a string counts bytes, not runes, but these titles are all ASCII so byte count and character count happen to agree.

Exercise 3 (Calculation): A function with the signature `func Reduce[T, U any](s []T, init U, f func(U, T) U) U` folds a slice into a single value. Trace the execution of `Reduce([]int{1, 2, 3, 4}, 0, func(acc, v int) int { return acc + v })`. What is the concrete type bound to `T`? What is the concrete type bound to `U`? What value does the function return, and what are the intermediate values of `acc` after each call to `f`?

`T` is bound to `int` (the element type of `[]int{1, 2, 3, 4}`). `U` is also bound to `int` (the type of the initial accumulator `0` and the return type of `f`).

A reasonable implementation of `Reduce` is:

```go
func Reduce[T, U any](s []T, init U, f func(U, T) U) U {
    acc := init
    for _, v := range s {
        acc = f(acc, v)
    }
    return acc
}
```

Tracing each call to `f`:

Iteration	`acc` before	`v`	`acc` after (`acc + v`)
1	0	1	1
2	1	2	3
3	3	3	6
4	6	4	10

The function returns **10**.

Exercise 4 (Where is the bug?):

```go
package main

import "fmt"

type Playlist []string

func Dedupe[T any](s []T) []T {
    seen := make(map[T]bool)
    var out []T
    for _, v := range s {
        if !seen[v] {
            seen[v] = true
```

```
            out = append(out, v)
        }
    }
    return out
}

func main() {
    p := Playlist{
        "Escape", "J'ai pas vingt ans !", "Escape",
        "J'en ai marre !", "J'ai pas vingt ans !",
    }
    fmt.Println(Dedupe(p))
}
```

The bug: `T` is constrained to `any`, but `map[T]bool` requires `T` to be `comparable`. The compiler rejects this with:

```
./prog.go:8:22: invalid map key type T (missing comparable constraint)
```

Using a type as a map key requires that it support `==` and `!=`. `any` does not guarantee this.

The fix: Change the constraint from `any` to `comparable`:

```
func Dedupe[T comparable](s []T) []T {
    seen := make(map[T]bool)
    var out []T
    for _, v := range s {
        if !seen[v] {
            seen[v] = true
            out = append(out, v)
        }
    }
    return out
}
```

That is the only change required. The call `Dedupe(p)` then compiles and runs.

It is tempting to worry about a second bug here: `Playlist` has underlying type `[]string`, and slices are never comparable in Go, so surely passing a `Playlist` cannot satisfy `comparable`? But that worry is misplaced. `Dedupe(p)` passes a value of type `Playlist`, which is itself a `[]string`. The parameter is `s []T`, so type inference matches `[]T` against `Playlist` (underlying `[]string`) and infers `T = string`, the **element** type — not `T = Playlist`. The map key is therefore `string`, which is perfectly comparable.

You can confirm the inference with reflection: inside `Dedupe`, `reflect.TypeOf` of a zero `T` reports `string`, and the returned slice has dynamic type `[]string`, not `Playlist`.

With the constraint changed to `comparable`, the program compiles and prints:

```
[Escape J'ai pas vingt ans ! J'en ai marre !]
```

The duplicate `"Escape"` and the duplicate `"J'ai pas vingt ans !"` are dropped, preserving first-seen order.

In short, there is exactly one bug: the constraint must be `comparable`, not `any`. No change to the call in `main` is needed, because `T` is inferred as `string` (the element type), not as the slice type `Playlist`.

Exercise 5 (Write a program): Implement a generic `Set[T comparable]` type backed by a `map[T]struct{}`. It should support three methods: `Add(v T)` (add an element), `Contains(v T) bool` (membership test), and `Values() []T` (return all elements as a slice in any order). In `main`, create a `Set[string]`, add the four song titles "Escape", "$100 Bills", "J'ai pas vingt ans !", and "J'en ai marre !", add "J'ai pas vingt ans !" a second time, and print the length of the set and whether it contains "Escape" and "Legend".

```go
package main

import "fmt"

// Set is a generic unordered collection of unique comparable values.
type Set[T comparable] struct {
    m map[T]struct{}
}

// Add inserts v into the set.
func (s *Set[T]) Add(v T) {
    if s.m == nil {
        s.m = make(map[T]struct{}) // lazy initialization
    }
    s.m[v] = struct{}{}
}

// Contains reports whether v is in the set.
func (s *Set[T]) Contains(v T) bool {
    _, ok := s.m[v]
    return ok
}

// Values returns all elements of the set as a slice in unspecified order.
func (s *Set[T]) Values() []T {
    out := make([]T, 0, len(s.m))
    for v := range s.m {
        out = append(out, v) // iteration order is random
    }
    return out
}

func main() {
    var songs Set[string]
    songs.Add("Escape")
    songs.Add("$100 Bills")
    songs.Add("J'ai pas vingt ans !")
    songs.Add("J'en ai marre !")
    songs.Add("J'ai pas vingt ans !") // duplicate --- should be ignored

    fmt.Println("length:", len(songs.Values()))
    fmt.Println("contains Escape:", songs.Contains("Escape"))
    fmt.Println("contains Legend:", songs.Contains("Legend"))
}
```

Output:

```
length: 4
contains Escape: true
contains Legend: false
```

`map[T]struct{}` is the standard Go idiom for a set. An empty struct (`struct{}`) occupies zero bytes, so only the keys consume memory. The second `Add("J'ai pas vingt ans !")` call is a no-op because the map key already exists — map assignment is idempotent. `Values()` returns four strings because the duplicate was silently dropped, but their order will vary between runs since Go map iteration is randomized. The `Set[T`

`comparable]` constraint is required because the map key type `T` must be comparable.

Chapter 19: Testing — Answers

Exercise 1 (Think about it): JUnit 5's `@ParameterizedTest` with `@CsvSource` and Go's table-driven tests with `t.Run` both let you run the same logic against many inputs. Describe two concrete advantages that Go's table-driven approach gives you over `@CsvSource`. Then explain the key behavioral difference between `t.Fatal` and `t.Error` inside a subtest, and describe a scenario where you would deliberately choose `t.Error` over `t.Fatal`.

Table-driven tests vs `@CsvSource` — two concrete advantages:

1. **Structured, type-safe test cases.** With `@CsvSource`, each row is a comma-separated string; numeric values must be parsed at runtime and type errors surface only when the test runs. A Go table is a slice of structs — the compiler checks every field at compile time. If you rename a field or change its type, the build breaks immediately. There is no equivalent compile-time safety in `@CsvSource`.
2. **Arbitrary Go values in each case.** `@CsvSource` can only express types that JUnit knows how to convert from strings: primitives, strings, enums. A Go table can hold any value — a function, an `error`, a struct, a slice — as a field in the test case struct. This lets you express cases like "given this pre-built request object, expect this error" without any serialization or custom converter.

`t.Fatal` vs `t.Error` inside a subtest:

Inside a `t.Run` subtest, `t.Fatal` stops only that subtest's goroutine — it calls `runtime.Goexit()` on the subtest's goroutine. The outer test loop continues and the next subtest runs normally. `t.Error` also affects only the subtest: it marks it as failed but the subtest continues executing.

A scenario where you would choose `t.Error` over `t.Fatal`: when you are validating multiple independent fields of a response struct and you want to see all failures at once. For example, if you call an HTTP handler and want to check both the status code and the response body, use `t.Error` for each. If you used `t.Fatal` on the status code check, a wrong status code would hide a potentially wrong body — you would have to fix and re-run to see the body error. With `t.Error`, one failing run shows you everything that is wrong.

Exercise 2 (What does this print?): Trace the output when this test is run with `go test -v`.

```go
package music_test

import "testing"

func checkPositive(t *testing.T, n int) {
    if n <= 0 {
        t.Errorf("expected positive, got %d", n)
    }
}

func TestSoundOfSilence(t *testing.T) {
    checkPositive(t, 1)
    t.Log("checked 1")
    checkPositive(t, -1)
    t.Log("checked -1")
    checkPositive(t, 2)
    t.Log("checked 2")
}
```

Output (with `go test -v`):

```
=== RUN   TestSoundOfSilence
    music_test.go:13: checked 1
    music_test.go:7: expected positive, got -1
    music_test.go:15: checked -1
    music_test.go:17: checked 2
--- FAIL: TestSoundOfSilence (0.00s)
FAIL
```

Key points to trace:

- `checkPositive(t, 1)`: `1 > 0`, so no error is recorded.
- `t.Log("checked 1")`: message is queued (line 13).
- `checkPositive(t, -1)`: `-1 <= 0`, so `t.Errorf` fires (line 7 inside the helper). `t.Errorf` is `t.Error` with formatting — it records a failure message and marks the test failed, but **does not stop execution**. The test keeps running.
- `t.Log("checked -1")`: message is queued (line 15).
- `checkPositive(t, 2)`: `2 > 0`, no error.
- `t.Log("checked 2")`: message is queued (line 17).
- With `-v`, all `t.Log` and error output is printed, and it appears in the order the calls executed (not errors first).

The buffered output prints in chronological call order, so "checked 1" (line 13) appears before the error (line 7), followed by "checked -1" (line 15) and "checked 2" (line 17). The test finishes with `--- FAIL`. Execution continues past the failing check because `t.Errorf` is used, not `t.Fatalf`.

Note on `t.Helper()` absence: `checkPositive` does not call `t.Helper()`. As a result, the failure line reported is inside `checkPositive` (the `t.Errorf` call on line 7), not in `TestSoundOfSilence` where `checkPositive(t, -1)` was called. Adding `t.Helper()` as the first line of `checkPositive` would make the reported line point to the `checkPositive(t, -1)` call inside `TestSoundOfSilence` instead.

Exercise 3 (Calculation): A benchmark function has the following structure:

```go
func BenchmarkCrazyTrain(b *testing.B) {
    for range b.N {
        _ = processTrack("Crazy Train")
    }
}
```

On the first probe the framework sets `b.N = 1` and measures elapsed time. It then sets `b.N = 100`, then `b.N = 10_000`, then `b.N = 1_000_000`. The framework stops when the total elapsed time exceeds one second. If `processTrack` takes exactly 2 µs per call, at which value of `b.N` does the total elapsed time first exceed one second? What is the reported ns/op value?

Answer:

Total elapsed time = `b.N * 2 µs` = `b.N * 2000 ns`.

b.N	Total time
1	2 µs
100	200 µs
10,000	20,000 µs = 20 ms
1,000,000	2,000,000 µs = 2 s

The first three probes are all well under one second. At `b.N = 1,000,000` the total is 2 s, which is the first

probe whose total elapsed time reaches (and exceeds) the 1 s default `-benchtime`. So the framework stops at **b.N = 1,000,000**.

Reported ns/op: The framework reports `total_time / b.N = 2,000,000,000 ns / 1,000,000 = 2000 ns/op`.

This matches `processTrack`'s actual per-call cost of 2 µs — the benchmark is accurate.

Exercise 4 (Where is the bug?): The following test helper is supposed to make failure output point to the call site in `TestBadApple`, but it does not. Identify the bug and show the fix.

```
package music

import "testing"

func assertNormalized(t *testing.T, input, want string) {
    got := normalize(input)
    if got != want {
        t.Fatalf("normalize(%q): got %q, want %q", input, got, want)
    }
}

func TestBadApple(t *testing.T) {
    assertNormalized(t, "bad apple!!", "Bad Apple!!")
    assertNormalized(t, "better off alone", "Better Off Alone")
}
```

The bug: `assertNormalized` does not call `t.Helper()`.

When `t.Fatalf` fires inside `assertNormalized`, Go's test framework records the file and line number of the `t.Fatalf` call inside the helper. The reported failure location is something like:

```
music_test.go:8: normalize("bad apple!!"): got "bad apple!!", want "Bad Apple!!"
```

That points inside `assertNormalized`, not to the line in `TestBadApple` that triggered the failure. You have to manually trace back to find which call site is responsible.

The fix: add `t.Helper()` as the first statement in `assertNormalized`:

```
func assertNormalized(t *testing.T, input, want string) {
    t.Helper()  // attribute failures to the caller, not this function
    got := normalize(input)
    if got != want {
        t.Fatalf("normalize(%q): got %q, want %q", input, got, want)
    }
}
```

With `t.Helper()` present, the reported failure location moves up to the line inside `TestBadApple` that called the helper:

```
music_test.go:13: normalize("bad apple!!"): got "bad apple!!", want "Bad Apple!!"
```

That line number points directly to `assertNormalized(t, "bad apple!!", "Bad Apple!!")` in `TestBadApple`, which is exactly where the problematic call lives.

Secondary note: `t.Fatalf` inside a helper is a reasonable choice here — once `normalize` returns a wrong value there is little point continuing. But whether to use `Fatal` or `Error` is a judgment call; the `t.Helper()` omission is the clear bug.

Exercise 5 (Write a program): Write a table-driven test for `TitleCase`. Your test must include at least five cases covering normal input, empty string, all-caps input, and a multi-word title. Use `t.Run` for each case and `t.Helper` in any helper you write.

```
// TitleCase converts a string to title case.
// Each word's first letter is uppercased; the rest are lowercased.
// Words are separated by spaces.
func TitleCase(s string) string
```

A complete solution, including a reference implementation of `TitleCase` so the test compiles and passes:

```
package music_test

import (
    "strings"
    "testing"
    "unicode"
)

// TitleCase converts a string to title case.
// Each word's first letter is uppercased; the rest are lowercased.
// Words are separated by spaces.
func TitleCase(s string) string {
    words := strings.Fields(s)
    for i, w := range words {
        if len(w) == 0 {
            continue
        }
        runes := []rune(w)
        runes[0] = unicode.ToUpper(runes[0])
        for j := 1; j < len(runes); j++ {
            runes[j] = unicode.ToLower(runes[j])
        }
        words[i] = string(runes)
    }
    return strings.Join(words, " ")
}

// assertEqual reports mismatches at the caller's site.
func assertEqual(t *testing.T, got, want, label string) {
    t.Helper()
    if got != want {
        t.Errorf("%s: got %q, want %q", label, got, want)
    }
}

func TestTitleCase(t *testing.T) {
    cases := []struct {
        name  string
        input string
        want  string
    }{
        {name: "empty",         input: "",               want: ""},
        {name: "single word",   input: "golden",         want: "Golden"},
        {name: "multi-word",    input: "good as hell",    want: "Good As Hell"},
```

```
        {name: "all caps",      input: "ABOUT DAMN TIME",  want: "About Damn Time"},
        {name: "mixed case",    input: "wOmAn",            want: "Woman"},
        {name: "already title", input: "Good As Hell",     want: "Good As Hell"},
    }

    for _, tc := range cases {
        t.Run(tc.name, func(t *testing.T) {
            got := TitleCase(tc.input)
            assertEqual(t, got, tc.want, "TitleCase("+tc.input+")")
        })
    }
}
```

Running `go test -v` produces:

```
=== RUN   TestTitleCase
=== RUN   TestTitleCase/empty
=== RUN   TestTitleCase/single_word
=== RUN   TestTitleCase/multi-word
=== RUN   TestTitleCase/all_caps
=== RUN   TestTitleCase/mixed_case
=== RUN   TestTitleCase/already_title
--- PASS: TestTitleCase (0.00s)
    --- PASS: TestTitleCase/empty (0.00s)
    --- PASS: TestTitleCase/single_word (0.00s)
    --- PASS: TestTitleCase/multi-word (0.00s)
    --- PASS: TestTitleCase/all_caps (0.00s)
    --- PASS: TestTitleCase/mixed_case (0.00s)
    --- PASS: TestTitleCase/already_title (0.00s)
PASS
```

Notes on the solution:

- The `name` field in each case struct is passed to `t.Run`, producing descriptive subtest names. A failing case shows up as `--- FAIL: TestTitleCase/all_caps` rather than just `--- FAIL: TestTitleCase`. Note that `t.Run` replaces spaces in subtest names with underscores, which is why `single word` appears as `single_word` in the output.
- `assertEqual` calls `t.Helper()` so that any failure message points to the line inside the `t.Run` body that called `assertEqual`, not to the `t.Errorf` line inside `assertEqual` itself.
- `t.Errorf` (not `t.Fatalf`) is used in the helper because each subtest has only one assertion; there is no reason to stop early. If the helper checked multiple things, `t.Fatalf` could be appropriate for a critical precondition.
- The six cases satisfy the problem requirements: empty string, single word (normal), multi-word, all-caps, and mixed case. The "already title" case is a bonus regression check.