Gorgo Go for Java Programmers

June 11, 2026

Contents

Chapter 5

Functions

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

Function Syntax

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

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

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

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

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

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

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

Multiple Return Values

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

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

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

The caller receives both values and must handle them:

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

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

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

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

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

strconv Revisited

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

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

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

Named Return Values

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

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

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

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

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

defer Modifying Named Returns

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

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

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

Variadic Functions

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

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

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

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

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

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

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

`fmt.Println` is itself variadic:

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

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

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

First-Class Functions

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

Function Types and Variables

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

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

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

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

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

Dispatch Tables

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

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

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

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

Closures

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

A Counter Example

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

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

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

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

Loop Variable Capture

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

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

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

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

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

init()

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

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

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

Rules

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

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

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

When to Use init()

Use `init` for:

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

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

Function Types as Parameters

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

Callbacks

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

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

Middleware Pattern

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

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

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

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

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

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

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

Pointer vs Value Semantics

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

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

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

Value copy behavior:

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

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

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

The Swap That Doesn't Work

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

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

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

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

The Swap That Works

Pass pointers and write through them:

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

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

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

When Mutation Requires a Pointer

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

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

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

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

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

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

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

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

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

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

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

Reference-Like Types

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

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

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

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

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

```
}

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

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

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

Contrast with Java

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

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

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

Escape Analysis

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

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

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

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

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

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

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

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

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

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

Inspecting Escape Decisions

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

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

The output contains lines like:

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

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

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

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

Try It

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

```
package main

import (
    "fmt"
    "strings"
)

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

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

```
    return sum
}

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

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

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

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

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

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

Try these modifications:

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

Key Points

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

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

Exercises

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

2. **What does this print?**

```go
package main

import "fmt"

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

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

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

```go
package main

import "fmt"

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

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

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

```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()
    }
}
```

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.