Gorgo Go for Java Programmers

June 11, 2026

Contents

Chapter 4

Control Flow

Java programmers are immediately comfortable with `if`, `for`, and `switch` in Go — the syntax is close enough that you can write working code on day one. But Go has a few surprises: there is only one loop keyword, `switch` does not fall through by default, and `defer` is a concept with no Java equivalent. This chapter covers all of it.

if / else

`if` and `else` work the same as in Java. The main syntactic difference is that the condition does not require parentheses (though they are allowed).

```
score := 95
if score >= 90 {
    fmt.Println("A")
} else if score >= 80 {
    fmt.Println("B")
} else {
    fmt.Println("C")
}
// A
```

The Init Statement

Go's `if` statement supports an optional **init statement** separated from the condition by a semicolon. Variables declared in the init statement are scoped to the entire `if/else if/else` chain — they disappear after the closing brace.

```
if n := len("Sandstorm"); n > 6 {
    fmt.Println("long:", n)
} else {
    fmt.Println("short:", n)
}
// long: 9
// n is not accessible here
```

The most common use is capturing the result of a function call and checking the error in one line:

```
if err := doSomething(); err != nil {
    fmt.Println("error:", err)
```

```
    return
}
```

This pattern is ubiquitous in Go. You will see it constantly when you read idiomatic Go code. Chapter 9 covers error handling in full, but start recognizing this shape now.

for — the Only Loop

Go has exactly one loop keyword: `for`. There is no `while`, no `do...while`, and no `foreach` (that role is played by `for range`). Three forms cover every use case.

C-Style Loop

The familiar three-clause form works exactly as in Java:

```
for i := 0; i < 5; i++ {
    fmt.Println(i)
}
// 0 1 2 3 4 (each on its own line)
```

While-Style Loop

Omit the init and post clauses and you get a while loop:

```
n := 1
for n < 100 {
    n *= 2
}
fmt.Println(n) // 128
```

Infinite Loop

Omit the condition entirely for an infinite loop. Use `break` to exit.

```
for {
    line := readLine()
    if line == "" {
        break
    }
    process(line)
}
```

Tip: The infinite `for` loop with an explicit `break` is the idiomatic Go replacement for `do...while`.

range

`for range` is Go's replacement for Java's `for-each` loop, extended to work on slices, arrays, maps, strings, channels, integers, and iterator functions. It yields up to two values, depending on what you range over: slices, arrays, maps, and strings yield an index (or key) and a value, while integer ranges (`for i := range n`), channels, and `iter.Seq[V]` iterators yield a single value. You can always bind just the first value and let Go implicitly drop the second — `for i := range fruits` gives you the index alone, no blank identifier needed.

The table below summarizes what the first and second values are for each rangeable type:

Range over	First value	Second value
Slice / array	index	element
Map	key	value
String	byte index of rune	rune
Integer (`range n`)	`0` to `n-1`	— (none)
Channel	received value	— (none)
`iter.Seq[V]`	value	— (none)
`iter.Seq2[K,V]`	key	value

Slices and arrays yield the index and the element, replacing Java's `for (int i = 0; i < list.size(); i++)` and `for (T v : list)` in one construct:

```
fruits := []string{"manzana", "naranja", "uva"}
for i, v := range fruits {
    fmt.Println(i, v)
}
// 0 manzana
// 1 naranja
// 2 uva
```

Maps yield each key–value pair in unspecified order:

```
m := map[string]int{"one": 1, "two": 2, "three": 3}
for k, v := range m {
    fmt.Println(k, v)
}
// output order is random --- never rely on it
```

Wut: Map iteration order in Go is deliberately randomized on every run. Java's `HashMap` also makes no order guarantee, but in practice the order is stable within a run. Go actively randomizes it to prevent accidental reliance on order.

Strings decode Unicode code points rather than bytes, which is important for non-ASCII text. As covered in Chapter 3, the index is the **byte** position of the rune, not its character index:

```
for i, r := range "café" {
    fmt.Printf("%d: %c\n", i, r)
}
// 0: c
// 1: a
// 2: f
// 3: é
```

Channels receive values one at a time, blocking until the next value arrives or the channel is closed — a clean way to drain a producer without a separate `ok` check on each receive. Channels are covered in Chapter 10.

Integers (Go 1.22+) give you a concise way to loop n times without a separate counter variable — a common pattern when you need a fixed number of iterations but do not need the index for anything else:

```
for i := range 5 {
    fmt.Print(i, " ")
}
// 0 1 2 3 4
```

Iterator functions (Go 1.23+) let library authors expose lazy, on-demand sequences without materializing the whole collection into a slice first — useful for large or infinite sequences. Any function that matches the `iter.Seq[V]` or `iter.Seq2[K,V]` signature from the `iter` package can be ranged over directly:

```
for title := range playlist.Titles() { // playlist.Titles() returns iter.Seq[string]
    fmt.Println(title)
}
```

The signatures, yield mechanics, and how to write your own iterators are covered in Chapter 18 (Generics).

In any `for range` form, use the blank identifier `_` to discard the index or the value when you do not need it:

```
for _, v := range fruits {
    fmt.Println(v) // index discarded
}

for i := range fruits {
    fmt.Println(i) // value implicitly discarded (single-variable range)
}
```

switch

Go's `switch` looks familiar but has two important differences from Java:

1. Cases do **not** fall through by default.
2. The `switch` expression is optional.

Basic switch

```
day := "lunes"
switch day {
case "lunes", "martes", "miércoles", "jueves", "viernes":
    fmt.Println("weekday")
case "sábado", "domingo":
    fmt.Println("weekend")
default:
    fmt.Println("unknown")
}
// weekday
```

Notice that multiple values can appear in one case, separated by commas. No `break` is needed — each case exits automatically.

Unlike Java, Go's `switch` is not limited to integers, strings, or enums. You can switch on any comparable type — structs, arrays, or any user-defined type that supports `==`:

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

p := Point{1, 2}
switch p {
case Point{0, 0}:
    fmt.Println("origin")
case Point{1, 2}:
    fmt.Println("one, two") // matches
default:
    fmt.Println("somewhere else")
}
```

Wut: In Java, forgetting `break` causes execution to fall into the next case. In Go, the opposite is true: execution stops at the end of each case by default. This eliminates an entire class of Java bugs.

fallthrough

When you genuinely need Java-style fall-through, use the `fallthrough` keyword explicitly. It transfers control to the **first statement** of the next case body without re-evaluating the case condition.

```
n := 1
switch n {
case 1:
    fmt.Println("one")
    fallthrough
case 2:
    fmt.Println("one or two")
case 3:
    fmt.Println("three")
}
// one
// one or two
```

`fallthrough` is unconditional — it always falls through regardless of whether the next case condition would match. It is rarely needed in practice.

Expression-Less switch

Omit the switch expression and each case becomes an independent boolean condition. This is a cleaner alternative to a long `if/else if` chain:

```
temp := 38.5
switch {
case temp < 0:
    fmt.Println("freezing")
case temp < 20:
    fmt.Println("cold")
case temp < 37:
    fmt.Println("warm")
default:
    fmt.Println("fiebre!")
}
// fiebre!
```

Type Switch (Preview)

A type switch selects a case based on the dynamic type of an interface value:

```
switch v := i.(type) {
case int:
    fmt.Println("int:", v)
case string:
    fmt.Println("string:", v)
default:
    fmt.Printf("other: %T\n", v)
}
```

Type switches are covered fully in Chapter 8 alongside interfaces.

Labeled break and continue

Java supports labeled statements to break out of nested loops. Go supports the same with labeled `break` and `continue`.

```
outer:
for i := 0; i < 3; i++ {
    for j := 0; j < 3; j++ {
        if i == 1 && j == 1 {
            break outer // exits both loops
        }
        fmt.Println(i, j)
    }
}
// 0 0
// 0 1
// 0 2
// 1 0
```

`continue outer` would skip the rest of the inner loop body and continue with the next iteration of the outer loop.

goto

Go has `goto`, which jumps to a labeled statement within the same function. It cannot jump over variable declarations. It is legal but almost never the right tool — `goto` is mentioned here so you know it exists, not as an invitation to use it.

defer

`defer` is one of Go's most distinctive features. A `defer` statement pushes a function call onto a per-function stack. All deferred calls run when the enclosing function returns, in **last-in, first-out** (LIFO) order.

```
func greet() {
    defer fmt.Println("goodbye")
    defer fmt.Println("see you later")
    fmt.Println("hello")
}

// greet() prints:
// hello
// see you later
// goodbye
```

You might be thinking "that's nifty" — and then "why would I ever use that?!?" Go does not have Java's `try-finally`, but `defer` is used in a similar way. Careful though: the mechanism is quite different — the `finally` clause runs at the end of a block in Java, but `defer` runs when a function returns.

Arguments Are Evaluated Immediately

The arguments to a deferred function call are evaluated **at the `defer` statement**, not when the deferred call actually runs.

```go
func demo() {
    x := 10
    defer fmt.Println(x) // x is captured as 10 right now
    x = 99
}
// prints: 10
```

This catches many Go beginners off guard. The value of `x` at defer time (10) is baked in; the change to `x = 99` does not affect it.

Closures Capture Variables by Reference

If the deferred function is a **closure** (a function that references a variable from an enclosing scope rather than receiving it as a parameter), it captures the variable itself as closures normally do, so it reads whatever value that variable holds at the time the deferred call actually runs.

```go
func demo() {
    x := 10
    defer fmt.Print(" direct ", x)              // x is captured as 10 right now
    defer func() { fmt.Print("closure ", x) }() // closure, not a direct call
    x = 99
}
// prints: closure 99 direct 10
```

This distinction is important: a deferred call with arguments evaluates the arguments immediately, but a deferred closure evaluates its captured variables lazily at return time.

Common Uses

Closing resources is the most common use of `defer`:

```go
f, err := os.Open("cancion.txt")
if err != nil {
    return err
}
defer f.Close() // guaranteed to run even if the rest of the function panics
```

This pattern ensures cleanup happens no matter how the function exits — return, error return, or panic.

Releasing locks:

```go
mu.Lock()
defer mu.Unlock()
```

Printing structured exit messages (useful during debugging):

```go
func process() {
    fmt.Println("process: start")
    defer fmt.Println("process: done")
    // ... work ...
}
```

defer Runs Even During a Panic

If a function panics, all of its deferred calls still run before the panic propagates up the call stack. This is why `defer f.Close()` is safe even when something unexpected happens.

```go
func riskyOp() {
    defer fmt.Println("cleanup always runs")
```

```
    panic("something went wrong")
}

func main() {
    riskyOp()
}
// prints: cleanup always runs
// then panics
```

Try It

Type this in and run it. It pulls together the four control-flow tools you will reach for most often: an `if` init statement, a `for range` over a map, an expression-less `switch`, and a `defer`.

```
package main

import (
    "fmt"
    "slices"
)

func main() {
    defer fmt.Println("done analyzing the playlist")

    plays := map[string]int{
        "Monaco":           120,
        "Where She Goes":   95,
        "Titi Me Preguntó": 200,
    }

    if total := len(plays); total > 0 {
        fmt.Println("tracks loaded:", total)
    }

    titles := make([]string, 0, len(plays))
    for title := range plays {
        titles = append(titles, title)
    }
    slices.Sort(titles) // map order is randomized, so sort for stable output

    for i, title := range titles {
        count := plays[title]
        switch {
        case count >= 150:
            fmt.Printf("%d. %s --- hit (%d plays)\n", i+1, title, count)
        case count >= 100:
            fmt.Printf("%d. %s --- popular (%d plays)\n", i+1, title, count)
        default:
            fmt.Printf("%d. %s --- deep cut (%d plays)\n", i+1, title, count)
        }
    }
}
```

Because the titles are sorted, the output is deterministic: the `deferd` line always prints last, after the three

ranked tracks.

Try these modifications:

- Add a `fallthrough` to one of the cases and observe how the output changes.
- Replace the expression-less `switch` with an equivalent `if/else if` chain.
- Remove the `slices.Sort` call, run it a few times, and watch the map iteration order shuffle.

Key Points

- `if` supports an init statement: `if err := f(); err != nil` — scopes the variable to the block.
- `for` is the only loop keyword in Go; it covers C-style, while-style, and infinite loops.
- `for range` iterates slices (index+value), maps (key+value), strings (byte-index+rune), channels, integers (Go 1.22+), and iterator functions (Go 1.23+).
- Map iteration order is deliberately randomized.
- `switch` does not fall through by default; use `fallthrough` explicitly when needed.
- An expression-less `switch` acts as a cleaner if/else chain.
- Labeled `break` and `continue` break out of nested loops.
- `defer` pushes a call onto a LIFO stack; all deferred calls run when the function returns.
- Defer arguments are evaluated immediately; closures in defer capture variables by reference.
- `defer` runs even when the function panics.

Exercises

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.

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")
}
```

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)
}
```

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))
    }
}
```

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.

6. **Calculation:** Consider this loop:

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

   How many times does the loop body execute, and what is the value of `i` when the loop condition is evaluated for the last time (and fails)?