Gorgo Go for Java Programmers

June 11, 2026

Contents

Chapter 10

Goroutines and Channels

Java programmers think about concurrency in terms of threads, locks, and shared mutable state. Go takes a fundamentally different approach: lightweight goroutines communicate through typed channels instead of fighting over shared memory. In Java you reach for an `ExecutorService` or a thread pool to run work concurrently, and as soon as two threads touch the same data you are reasoning about locks, memory visibility, and race conditions — shared-memory threading is notoriously easy to get subtly wrong. Go flips the model: goroutines are so cheap you can launch one per task, and instead of guarding shared state you pass values over channels, letting the type system and the runtime handle the handoff. This makes a whole class of concurrency bugs harder to write in the first place. This chapter covers how to launch goroutines with `go f()`, how channels act as typed conduits with synchronization built in, how `select` coordinates multiple channels, and the Go proverb that ties it all together.

Goroutines

A **goroutine** is a function executing concurrently with the rest of your program. You launch one by placing the `go` keyword in front of a function call:

```
go f()           // launch f in a new goroutine
go pkg.Method()  // works with any callable
go func() {      // anonymous function launched immediately
    fmt.Println("Escape")
}()
```

That is the entire syntax. There is no `Thread` class, no `Runnable`, no `ExecutorService`. There are also no handles to goroutines! That means **you cannot query if a goroutine has completed**. If you need to track the status of goroutines, you have to do it yourself! **sync.WaitGroup**, a common way to do it, is explained in the next chapter.

Goroutines vs Java Threads

Java threads are OS threads under the hood — each one starts with a fixed stack (typically 512 KB to 1 MB) and is managed by the operating system scheduler. Creating tens of thousands of Java threads is impractical because each one reserves a large, fixed block of memory and requires an OS context switch to schedule.

Goroutines are very different:

	Java thread	Go goroutine
Starting stack	512 KB – 1 MB	~2 KB
Scheduler	OS kernel	Go runtime (user space)

	Java thread	Go goroutine
Context switch	kernel mode, microseconds	cooperative + preemptive, nanoseconds
Practical ceiling	thousands	millions

A Go program can run a million goroutines comfortably on a laptop. The runtime multiplexes them onto a small number of OS threads (`GOMAXPROCS`, default equal to CPU count, caps how many threads execute Go code at once — the runtime spins up extra OS threads for goroutines blocked in syscalls or cgo). This is called **M:N scheduling** — M goroutines mapped onto N OS threads.

Tip: `GOMAXPROCS` defaults to the number of CPU cores, but since Go 1.25 the runtime also respects the container's cgroup CPU limit. So inside a container with a CPU quota, `GOMAXPROCS` may be lower than the host's core count — which is usually what you want.

Tip: Because goroutines are cheap, Go code often launches a goroutine for every incoming request or every item in a work queue. Java code typically uses a thread pool to amortize the cost of thread creation; in Go that concern largely disappears.

Goroutine stacks start small (~2 KB) and **grow automatically** when needed. The runtime detects that the stack is about to overflow and copies the goroutine's stack to a larger allocation. You never set a stack size — it is invisible to your code.

Wut: Goroutine stacks can grow and shrink at runtime. This means a pointer into a goroutine's stack frame may become invalid after the stack is relocated. The Go compiler and runtime manage this transparently — you never see it — but it is one reason Go does not allow pointer arithmetic (see Chapter 2).

A Simple Goroutine Example

```go
package main

import (
    "fmt"
    "time"
)

func playTrack(title string) {
    fmt.Println("Playing:", title)
}

func main() {
    go playTrack("Escape")            // runs concurrently
    go playTrack("Legend")            // runs concurrently
    time.Sleep(10 * time.Millisecond) // give goroutines time to run
    fmt.Println("done")
}
```

Trap: When `main` returns, the program exits — even if goroutines are still running. The `time.Sleep` above is a toy fix; real programs use channels or `sync.WaitGroup` (Chapter 11) to wait for goroutines to finish. [*goroutine-must-exit*]

Channels

A **channel** is a typed conduit through which goroutines send and receive values. The zero value of a channel is `nil`; use `make` to create one.

Wut: A send or receive on a `nil` channel blocks forever. This sounds like a bug waiting to happen, but it is handy in a `select`: setting a case's channel variable to `nil` disables that case until you set it back to a real channel.

```
ch := make(chan string)    // unbuffered channel of strings
ch := make(chan int, 10)   // buffered channel of ints, capacity 10
```

The send operator `<-` puts a value into the channel. The receive operator `<-` takes a value out:

```
ch <- "Turn Me On"   // send "Turn Me On" into ch
msg := <-ch          // receive from ch; assign to msg
```

Both the send and receive operators block until the other side is ready, unless the channel is buffered (covered in the next section).

Tip: Channels are first-class values — you can pass them to functions, store them in structs, and return them. The `<-` operator is the same for send and receive; the position relative to the channel name determines the direction: `ch <- v` sends, `v := <-ch` receives.

A Channel-Coordinated Goroutine

```
package main

import "fmt"

func fetchLyrics(out chan<- string) {
    out <- "Do you think you're better off alone?"  // send into channel
}

func main() {
    ch := make(chan string) // unbuffered channel
    go fetchLyrics(ch)      // goroutine sends one value
    lyric := <-ch           // main goroutine receives it
    fmt.Println(lyric)
}
```

Output:

```
Do you think you're better off alone?
```

`main` blocks on `<-ch` until `fetchLyrics` sends. There is no mutex, no lock, no explicit signal — the channel synchronizes the two goroutines automatically.

Buffered vs Unbuffered Channels

An **unbuffered channel** (`make(chan T)`) has no internal queue. A send blocks until a receiver is waiting; a receive blocks until a sender is ready. They synchronize at the point of exchange. This is like a relay baton hand-off: both runners must be at the exchange point at the same time.

A **buffered channel** (`make(chan T, n)`) has an internal queue of capacity `n`. A send blocks only when the queue is full; a receive blocks only when the queue is empty.

```go
ch := make(chan string, 3)  // capacity 3

ch <- "Turn Me On"    // queued immediately, no receiver needed
ch <- "Legend"        // queued
ch <- "Escape"        // queued

fmt.Println(<-ch)  // Turn Me On
fmt.Println(<-ch)  // Legend
fmt.Println(<-ch)  // Escape
```

Tip: Use an unbuffered channel when you want a **synchronization point** — when you need to know that the receiver has received the value. Use a buffered channel when you want to decouple the sender's pace from the receiver's pace, up to a known limit.

Trap: Sending to a full buffered channel blocks just like sending to an unbuffered channel. A channel with capacity 1 is not the same as "fire and forget."

Directional Channel Types

You can restrict a channel to send-only or receive-only in a function signature:

```go
func produce(out chan<- string) { // out is send-only
    out <- "$100 Bills"
}

func consume(in <-chan string) {  // in is receive-only
    fmt.Println(<-in)
}
```

The full bidirectional `chan string` converts to either directional type automatically. Directional types document intent and prevent bugs: a function that only receives cannot accidentally send and vice versa.

```go
ch := make(chan string)  // bidirectional
go produce(ch)           // ch narrows to chan<- inside produce
consume(ch)              // ch narrows to <-chan inside consume
```

Tip: Always use directional channel types in function parameters. If a function only reads from a channel, declare the parameter as `<-chan T`. If it only writes, declare it as `chan<- T`. This is self-documenting and lets the compiler catch mistakes at compile time.

Closing a Channel

Calling `close(ch)` marks the channel as closed. After that:

- **Receiving** from a closed channel returns immediately. If there are buffered values, they are drained first. Once empty, receives return the zero value of the channel's element type.
- **Sending** to a closed channel panics.

The idiomatic way to receive from a channel until it is closed is `range`:

```go
ch := make(chan string, 3)
ch <- "Escape"
ch <- "$100 Bills"
ch <- "Legend"
```

```
close(ch)

for msg := range ch {  // drains until channel is closed and empty
    fmt.Println(msg)
}
```

Output:

```
Escape
$100 Bills
Legend
```

You can also use the comma-ok idiom (introduced in Chapter 7 for maps) to detect a closed channel:

```
msg, ok := <-ch  // ok is false when ch is closed and empty
if !ok {
    fmt.Println("channel closed")
}
```

Trap: Only the sender should close a channel. Closing a channel from the receiver's side is a data race if the sender is still running. Closing an already-closed channel also panics. The rule is simple: the goroutine that produces values owns the channel and is responsible for closing it.

Wut: You never *have* to close a channel. Unlike a file, an unclosed channel is not a resource leak — it will be garbage-collected once all goroutines holding a reference to it exit. Close a channel only when you need the receiver to know that no more values are coming. A goroutine that never exits, however, prevents GC of every value it has closed over. [***leaked-goroutine-grows-memory***]

A Producer-Consumer Pipeline

```
package main

import "fmt"

func generate(tracks []string, out chan<- string) {
    for _, t := range tracks {
        out <- t  // send each track
    }
    close(out)    // signal: no more tracks
}

func main() {
    playlist := []string{"Escape", "$100 Bills", "Legend", "Turn Me On"}
    ch := make(chan string)

    go generate(playlist, ch)

    for track := range ch {
        fmt.Println("Playing:", track)
    }
}
```

Output:

```
Playing: Escape
Playing: $100 Bills
```

```
Playing: Legend
Playing: Turn Me On
```

`generate` closes `ch` when the loop finishes, which causes the `range` in `main` to terminate.

The `select` Statement

`select` is to channels what `switch` is to values: it waits on multiple channel operations and runs the first one that is ready.

```
select {
case msg := <-ch1:
    fmt.Println("from ch1:", msg)
case msg := <-ch2:
    fmt.Println("from ch2:", msg)
}
```

If both `ch1` and `ch2` are ready simultaneously, Go picks one at **random**. If neither is ready, `select` blocks until one becomes ready.

Fan-In

Fan-in merges multiple channels into one. `select` makes this straightforward:

```
package main

import (
    "fmt"
    "time"
)

func source(name, msg string, out chan<- string, delay time.Duration) {
    time.Sleep(delay)
    out <- name + ": " + msg
}

func fanIn(ch1, ch2 <-chan string) <-chan string {
    merged := make(chan string, 2)
    go func() {
        for i := 0; i < 2; i++ {
            select {
            case msg := <-ch1:
                merged <- msg
            case msg := <-ch2:
                merged <- msg
            }
        }
        close(merged)
    }()
    return merged
}

func main() {
    ch1 := make(chan string, 1)
    ch2 := make(chan string, 1)
```

```
    go source("Jaroslav Beck", "Escape",     ch1, 10*time.Millisecond)
    go source("Jaroslav Beck", "Turn Me On", ch2, 5*time.Millisecond)

    for msg := range fanIn(ch1, ch2) {
        fmt.Println(msg)
    }
}
```

Whichever goroutine sends first will be received first, regardless of declaration order.

Timeouts

`time.After(d)` returns a `<-chan time.Time` that receives a value after duration `d`. Use it in a `select` to implement timeouts:

```
select {
case result := <-workCh:
    fmt.Println("got result:", result)
case <-time.After(500 * time.Millisecond):
    fmt.Println("timed out waiting for result")
}
```

Java programmers reach for `Future.get(timeout, unit)`; in Go you compose `select` with `time.After`.

Non-Blocking Send and Receive

Adding a `default` case makes a `select` non-blocking: if no channel is ready, the `default` case runs immediately.

```
select {
case msg := <-ch:
    fmt.Println("received:", msg)
default:
    fmt.Println("nothing ready, moving on")
}
```

Use `default` sparingly. A busy-polling loop around a `select` with `default` wastes CPU time; channels with a proper blocking `select` usually express the intent more clearly.

Tip: A common use of the non-blocking pattern is a **done channel** signal combined with a work channel:

```
select {
case work := <-workCh:
    process(work)
case <-doneCh:
    return  // shut down cleanly
default:
    // nothing to do right now
}
```

Share Memory by Communicating

The most famous Go concurrency proverb is:

> "Don't communicate by sharing memory; share memory by communicating."

In Java, you protect shared state with `synchronized`, `ReentrantLock`, `volatile`, and `AtomicInteger`. The data lives in a shared heap; the locks prevent conflicting access.

Go's channel model inverts this. Instead of several goroutines all reading and writing the same variable while holding a lock, you pass ownership of the data through a channel. At any instant, exactly one goroutine holds the value — the one that last received it from the channel. No lock is needed because the data is never shared simultaneously.

```
// Java style: shared mutable state with a lock
// mu.Lock(); count++; mu.Unlock()

// Go style: one goroutine owns the state; others send requests
type command struct {
    inc    bool
    result chan<- int
}

func counter(cmds <-chan command) {
    count := 0
    for cmd := range cmds {
        if cmd.inc {
            count++
        }
        if cmd.result != nil {
            cmd.result <- count
        }
    }
}
```

Only `counter` touches `count`; all other goroutines communicate with it through the `cmds` channel. There is no lock because there is no shared access.

This model does not replace mutexes entirely — Chapter 11 covers `sync.Mutex` and `sync.WaitGroup` for the cases where channels are the wrong tool. But for many concurrency problems, especially pipelines and fan-out/fan-in patterns, the channel model is cleaner and less error-prone.

Tip: A goroutine that owns a piece of state and exposes it only through a channel is called an **active object** or **actor**. The `counter` example above is one. This pattern eliminates data races on the owned state by construction — you cannot accidentally forget to take a lock. Keeping ownership simple also makes it obvious when the goroutine can exit, which is essential for avoiding leaks. [*obvious-goroutine-lifetimes*]

Try It

Type this in and run it a few times. It launches one goroutine per track, collects results over a channel, and uses `select` with `time.After` to bail out if the work takes too long. Notice that the order of the "fetched" lines changes between runs — that is the scheduler at work.

```
package main

import (
    "fmt"
    "time"
)
```

```go
// fetch sends a result for each track after a short delay.
func fetch(track string, out chan<- string) {
    time.Sleep(5 * time.Millisecond)
    out <- "fetched: " + track
}

func main() {
    tracks := []string{"Padam Padam", "Houdini", "Espresso"}
    results := make(chan string)

    for _, t := range tracks {
        go fetch(t, results) // one goroutine per track
    }

    timeout := time.After(500 * time.Millisecond)
    got := 0
    for got < len(tracks) {
        select {
        case r := <-results:
            fmt.Println(r)
            got++
        case <-timeout:
            fmt.Println("timed out waiting for fetches")
            return
        }
    }
    fmt.Println("all", got, "tracks fetched")
}
```

The final `all 3 tracks fetched` line is deterministic; the lines above it are not.

Try these modifications:

- Change `results` to a buffered channel (`make(chan string, len(tracks))`) and confirm the program still works.
- Lower the timeout to `1 * time.Millisecond` and watch the `select` take the timeout case instead.
- Add a `done := make(chan struct{})` channel and a third `select` case that returns when something closes it.

Key Points

- A goroutine is launched with `go f()`. Goroutines are cheap (~2 KB initial stack) and multiplexed onto OS threads by the Go runtime.
- Java threads are OS threads with large fixed stacks; Go goroutines are user-space and grow dynamically. You can run millions of goroutines where Java would permit only thousands of threads.
- `make(chan T)` creates an unbuffered channel; `make(chan T, n)` creates a buffered channel with capacity `n`.
- An unbuffered send blocks until a receiver is ready; an unbuffered receive blocks until a sender is ready. A buffered send blocks only when the buffer is full.
- Use directional types (`chan<- T`, `<-chan T`) in function parameters to document intent and catch mistakes at compile time.
- Only the sender should close a channel. Receiving from a closed channel returns the zero value; sending to a closed channel panics.
- You do not have to close a channel; close it only to signal "no more values."
- `range` over a channel drains it until it is closed.

- `select` waits on multiple channel operations; the first ready case runs. A `default` case makes `select` non-blocking.
- Use `time.After` in a `select` case to implement timeouts.
- The Go proverb: "Don't communicate by sharing memory; share memory by communicating." Pass ownership of data through channels instead of locking shared variables.

Exercises

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?

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)
}
```

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?

```go
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")
}
```

4. **Where is the bug?**

```
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)
    }
}
```

5. **Write a program:** Write a program that launches three goroutines. Each goroutine sleeps for a different duration (use `time.Sleep` with values like `10ms`, `20ms`, `30ms`) and then sends the string `"Jaroslav Beck & Crispin: Legend"`, `"Jaroslav Beck: $100 Bills"`, or `"Jaroslav Beck: Turn Me On"` 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.