Gorgo Go for Java Programmers

June 11, 2026

Contents

Chapter 19

Testing

Go's `testing` package is deliberately minimal: no annotations, no test runner configuration files, no assertion library. What it lacks in ceremony it makes up for in power — table-driven tests, subtests, benchmarks, and a built-in fuzzer all ship with the standard library. In Java, testing is a stack of dependencies and configuration: you add JUnit to your `pom.xml` or `build.gradle`, wire up Surefire or the Gradle test task, and lean on annotations like `@Test`, `@BeforeEach`, and `@ParameterizedTest` to tell the runner what to do. In Go, testing is first-class tooling that is already installed: `go test` discovers any `TestXxx` function in a `_test.go` file and runs it, with no build-tool plugin, no runner config, and no annotations. The same toolchain you use to build also benchmarks, fuzzes, measures coverage, and detects data races — so writing a test is as cheap as writing the function it covers. If you are used to JUnit 5, most concepts will map cleanly; the idioms are just different.

The `testing.T` Type

Every test function takes a single argument of type `*testing.T`. The naming rule is strict: a test function must be named `TestXxx` where `Xxx` does not begin with a lowercase letter (an uppercase letter by convention), and it must live in a file whose name ends in `_test.go`.

```
package music

import "testing"

func TestBadApple(t *testing.T) {
    got := normalize("bad apple!!")
    want := "Bad Apple!!"
    if got != want {
        t.Errorf("normalize(%q) = %q, want %q", "bad apple!!", got, want)
    }
}
```

The test lives in `package music` — the same package as the code — so it can call the unexported `normalize`. A test file may instead declare `package music_test` (an *external* test package); it then sees only exported names, which keeps the test honest about the public API.

Go discovers and runs test files automatically with `go test`. There is no `@Test` annotation, no test class — just a naming convention.

`t.Error`, `t.Fatal`, and `t.Log`

The three methods you will reach for most often are:

```
func (t *T) Error(args ...any)                 // mark test failed; continue running
func (t *T) Errorf(format string, args ...any) // like Error, with a format string
func (t *T) Fatal(args ...any)                 // mark test failed; stop this test immediately
func (t *T) Fatalf(format string, args ...any) // like Fatal, with a format string
func (t *T) Log(args ...any)                   // log a message; shown only on failure or -v
func (t *T) Logf(format string, args ...any)   // record a formatted message
```

The key difference between `Error` and `Fatal` mirrors the difference between a recoverable and an unrecoverable condition. `t.Error` marks the test as failed but lets it keep running, which is useful when you want to report multiple independent failures in one pass. `t.Fatal` marks the test as failed and stops the current test function immediately.

Tip: Use `t.Fatal` when subsequent checks depend on a previous one passing. If you set up a database connection in a test and it fails, there is no point running the 20 assertions that follow — use `t.Fatal` to stop early. Use `t.Error` when each check is independent and you want a full picture of all failures.

Trap: `t.Fatal` calls `runtime.Goexit()` under the hood, which unwinds deferred functions in the current goroutine. If you call `t.Fatal` from a goroutine that is not the test's own goroutine, it will **not** stop the test — it will stop only that goroutine. Call `t.Fatal` only from the goroutine that received `t` directly (the test function itself or a helper called from it, not a spawned goroutine).

In Java with JUnit 5, you use `Assertions.assertEquals`, `Assertions.assertTrue`, and `Assertions.assertAll`. Go has no assertion library in the standard library. The idiomatic style is a plain `if` that calls `t.Error` or `t.Fatal`. Third-party libraries like `github.com/stretchr/testify/assert` exist, but many Go teams prefer the standard approach. Whichever you use, your failure messages should state the input, the actual result, and the expected result so that a reader can diagnose the failure without re-running the test. [*test-failure-describes-wrong*]

Table-Driven Tests

Table-driven tests are Go's answer to JUnit 5's `@ParameterizedTest`. Instead of writing one test function per case, you define a slice of structs — each struct is a test case — and loop over them. [*table-driven-tests*]

```
package music

import (
    "testing"
)

func TestBetterOffAlone(t *testing.T) {
    cases := []struct {
        name  string
        input int
        want  string
    }{
        {name: "zero",     input: 0,   want: "zero stars"},
        {name: "one star", input: 1,   want: "one star"},
        {name: "max",      input: 5,   want: "five stars"},
        {name: "negative", input: -1,  want: "zero stars"},
```

```
    }

    for _, tc := range cases {
        t.Run(tc.name, func(t *testing.T) {
            got := starRating(tc.input)
            if got != tc.want {
                t.Errorf("starRating(%d) = %q, want %q", tc.input, got, tc.want)
            }
        })
    }
}
```

The outer test function iterates over the cases and calls `t.Run` for each one. `t.Run` creates a **subtest**: a named, independently tracked test run.

`t.Run` Subtests

`t.Run(name, func(t *testing.T))` registers and runs a named subtest.

```
func (t *T) Run(name string, f func(t *T)) bool // run f as a named subtest; true if f passes
```

Each subtest:

- Has its own pass/fail state.
- Can be run individually with `-run TestFoo/subtest_name`.
- Appears in failure output as `TestFoo/subtest_name`, making it easy to identify which case failed.

Tip: Name your test cases with a `name` field and pass it as the first argument to `t.Run`. When a case fails, the output shows `--- FAIL: TestBetterOffAlone/negative (0.00s)` so you immediately know which row broke. Without subtests you would only see `--- FAIL: TestBetterOffAlone` and have to dig through the output to find which input caused it.

In JUnit 5, parameterized tests use `@ParameterizedTest` with `@MethodSource` or `@CsvSource`. Go's table-driven approach with `t.Run` is more explicit — you write a slice literal and a loop — but it gives you the same per-case naming and isolation. Format each `t.Errorf` call with the actual result before the expected result: `got %q, want %q`. [***actual-before-expected***]

`t.Helper()`

When you extract repeated assertion logic into a helper function, Go's test output points to the **helper** as the failure site rather than the call site in the test. That is rarely what you want. `t.Helper()` fixes this: calling it at the top of a helper function tells the testing framework to attribute any failure in that function to the **caller**.

Without `t.Helper()`:

```
// checkEqual reports whether got == want, but failure points to this line, not the caller.
func checkEqual(t *testing.T, got, want string) {
    if got != want {
        t.Errorf("got %q, want %q", got, want)  // file:line points here
    }
}
```

With `t.Helper()`:

```
// checkEqual reports whether got == want.
func checkEqual(t *testing.T, got, want string) {
    t.Helper()  // attribute failures to the caller, not this function
```

```
    if got != want {
        t.Errorf("got %q, want %q", got, want)  // file:line now points to the caller
    }
}
```

Now when `checkEqual` fails, the reported line is the line in your test function that called `checkEqual`, not the line inside `checkEqual` itself.

```
func TestCrazyTrain(t *testing.T) {
    checkEqual(t, normalize("crazy train"), "Crazy Train")                    // failure here
    checkEqual(t, normalize("THE SOUND OF SILENCE"), "The Sound of Silence") // failure here
}
```

Tip: Any function that calls `t.Error`, `t.Fatal`, `t.Log`, or another helper should call `t.Helper()` as its first statement. Forgetting `t.Helper` is a common oversight that makes failure output point to the wrong file and line. [***t-helper-for-helpers***]

`t.Cleanup`: Teardown the Idiomatic Way

JUnit 5 has `@AfterEach` to undo whatever `@BeforeEach` set up. Go's analog is `t.Cleanup`, which registers a function to run when the test (or subtest) finishes.

```
func (t *T) Cleanup(f func())  // register f to run when the test ends (LIFO order)
```

You register the cleanup right next to the code that needs it, so the setup and its teardown live together:

```
func TestPlaylistFile(t *testing.T) {
    f, err := os.CreateTemp("", "playlist-*.txt")
    if err != nil {
        t.Fatal(err)
    }
    t.Cleanup(func() {
        os.Remove(f.Name())  // runs when the test ends, pass or fail
    })

    // ... use f ...
}
```

Cleanups run in **last-in, first-out** order, just like deferred functions, and they run whether the test passes, fails, or calls `t.Fatal`.

Tip: `t.Cleanup` beats a bare `defer` for teardown that lives inside a helper. A `defer` in a helper fires when the *helper* returns, which is too early. `t.Cleanup` registered inside that helper fires when the *test* ends, so a `newTestServer(t)` helper can register its own shutdown and the caller never has to remember to close anything.

`t.Parallel`: Running Tests Concurrently

By default tests in a package run one after another. Calling `t.Parallel` signals that a test is safe to run alongside other parallel tests.

```
func (t *T) Parallel()  // mark this test to run in parallel with other parallel tests
```

It pairs naturally with `t.Run`: each subtest calls `t.Parallel` as its first statement, the runner pauses it, and once the loop finishes registering subtests it runs the paused ones together.

```go
func TestNormalizeParallel(t *testing.T) {
    titles := []string{"Monaco", "La Bachata", "Bad Apple!!"}
    for _, title := range titles {
        t.Run(title, func(t *testing.T) {
            t.Parallel()  // run this subtest concurrently with its siblings
            if normalize(title) == "" {
                t.Errorf("normalize(%q) was empty", title)
            }
        })
    }
}
```

Trap: Before Go 1.22, the loop variable was shared across iterations, so a parallel subtest that captured `title` would see the *last* value by the time it actually ran. Go 1.22 changed loop variables to be per-iteration, so the capture above is safe. If you target older toolchains, copy the variable inside the loop: `title := title`.

Benchmarks

A benchmark measures the performance of a piece of code. Benchmarks follow the same file convention as tests — `_test.go` — but the function name starts with `Benchmark` and the argument is `*testing.B`.

```go
func BenchmarkNormalize(b *testing.B) {
    for range b.N {
        normalize("bad apple!!")
    }
}
```

`b.N` is the number of iterations the framework chooses. The benchmark runner starts with a small `N` and increases it until the total run time is stable enough to report a reliable per-operation time. You do not set `b.N`; the framework sets it for you.

Run benchmarks with `-bench`:

```
go test -bench=. -benchmem ./...
```

The `-benchmem` flag adds allocation counts to the output.

`b.Loop`: the Modern Idiom

Go 1.24 added `b.Loop`, which is now the preferred way to write the benchmark loop.

```go
func (b *B) Loop() bool  // true until enough iterations have run; drives the benchmark loop
```

Instead of `for range b.N`, you write `for b.Loop()`:

```go
func BenchmarkNormalize(b *testing.B) {
    for b.Loop() {
        normalize("bad apple!!")
    }
}
```

`b.Loop` returns `true` until the framework has run enough iterations, then returns `false` to end the loop. It has two advantages over `b.N`. First, any setup that runs **before** the loop and any teardown **after** it are automatically excluded from the timing — you no longer need `b.ResetTimer` for the common case. Second, `b.Loop` keeps the benchmarked call alive so the compiler cannot optimize it away, a footgun that `b.N` loops sometimes hit.

```go
func BenchmarkTopTrack(b *testing.B) {
    data := buildLargePlaylist(10_000)  // setup outside the loop, not timed
    for b.Loop() {
        _ = findTopTrack(data)
    }
}
```

`b.N` still works and you will see it in older code, but reach for `b.Loop` in new benchmarks.

Tip: With `b.Loop`, put expensive setup before the loop and teardown after it; both are excluded from the measurement automatically. Only fall back to `b.ResetTimer` (next section) when you are still writing a `b.N`-style loop or need to reset the timer partway through.

`b.ResetTimer`

If your benchmark has expensive setup before the measured loop, use `b.ResetTimer()` to exclude the setup time from the measurement.

```go
func BenchmarkSoundOfSilence(b *testing.B) {
    data := buildLargePlaylist(10_000)  // expensive setup
    b.ResetTimer()                      // start timing only from here
    for range b.N {
        _ = findTopTrack(data)
    }
}
```

`b.ResetTimer` zeroes the elapsed time and allocation counters so that only the measured loop is included in the reported numbers.

Tip: Always call `b.ResetTimer()` after any setup that you do not want included in the benchmark. Without it, a slow setup inflates the per-operation time, making the benchmark misleading.

Trap: A plain `go test ./...` does not run benchmarks. Benchmarks only run when you pass `-bench=<pattern>`; without it, benchmark functions are compiled but never executed.

In JUnit 5, there is no built-in benchmarking; you need JMH or similar. Go ships a benchmarking harness in the standard library.

Fuzzing

Fuzzing automatically generates inputs to find crashes and panics. A fuzz test is a function named `FuzzXxx` that takes `*testing.F`.

```go
func FuzzBetterOffAlone(f *testing.F) {
    f.Add("Better Off Alone")   // seed corpus: start from known inputs
    f.Add("")
    f.Add("BAD APPLE!!")

    f.Fuzz(func(t *testing.T, s string) {
        result := normalize(s)
        if len(result) > 0 && result[0] >= 'a' && result[0] <= 'z' {
            t.Errorf("normalize(%q) starts with lowercase: %q", s, result)
```

```
        }
    })
}
```

`f.Add` seeds the fuzzer with known inputs. The fuzzer uses those seeds as a starting point and then mutates them to generate new inputs.

`f.Fuzz` registers the function that is called for each generated input. The first argument is always `*testing.T`; the remaining arguments match the types passed to `f.Add`.

Run fuzzing with `-fuzz`:

```
go test -fuzz=FuzzBetterOffAlone -fuzztime=30s .
```

Unlike `-run` and `-bench`, `-fuzz` accepts only a single package, so point it at one directory rather than `./...`. Without `-fuzz`, a fuzz test runs only the seed corpus — just like a regular test. With `-fuzz`, the engine runs indefinitely (or until `-fuzztime` elapses) mutating inputs until it finds a failure. When a failure is found, the engine writes the failing input to `testdata/fuzz/FuzzXxx/` so you can reproduce it.

Tip: Fuzz tests double as regression tests. The seed corpus (`f.Add` calls) runs every time `go test` runs — no `-fuzz` flag needed. Add the `testdata/fuzz/` directory to version control so that previously found failures are always checked.

Wut: Fuzzing is not available as a built-in in JUnit 5; you need a separate library. Go 1.18 added native fuzzing to the standard toolchain.

Example Tests

An example test is a function named `ExampleXxx` whose body ends in an `// Output:` comment. `go test` runs the function, captures what it prints to standard output, and fails if the captured output does not match the comment.

```
func ExampleNormalize() {
    fmt.Println(Normalize("bad apple!!"))
    // Output: Bad Apple!!
}
```

The payoff is double duty: the function is a verified test *and* it shows up in the package's generated documentation as runnable sample code. If someone changes `Normalize` so the output drifts, the example test fails like any other test — your docs cannot rot.

The name links the example to what it documents: `ExampleNormalize` attaches to the `Normalize` function, `ExampleClient_Get` to the `Get` method on `Client`, and a bare `Example` to the package itself. For output whose line order is not guaranteed (for example, ranging over a map), use `// Unordered output:` instead, which compares lines as a set.

Wut: An `ExampleXxx` function only runs if it has an `// Output:` (or `// Unordered output:`) comment. Without that comment, `go test` still compiles the example — so it must build — but never executes it, and it never fails.

JUnit 5 has no real analog here; Javadoc snippets are not executed or verified by the test runner.

Testing HTTP Handlers with `httptest`

Chapter 15 built HTTP handlers; the `net/http/httptest` package tests them without binding a real port or making you guess at a free one. There are two main styles.

The lightweight style uses `httptest.NewRecorder`, an `http.ResponseWriter` that records the status, headers, and body so you can assert on them. You call the handler directly — no network involved.

```
func NewRecorder() *httptest.ResponseRecorder                          // records the response
func NewRequest(method, target string, body io.Reader) *http.Request   // a request for tests
```

```
func greet(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hola, %s", r.URL.Query().Get("name"))
}

func TestGreet(t *testing.T) {
    req := httptest.NewRequest(http.MethodGet, "/?name=Mundo", nil)
    rec := httptest.NewRecorder()

    greet(rec, req)  // call the handler directly

    resp := rec.Result()
    if resp.StatusCode != http.StatusOK {
        t.Errorf("status = %d, want %d", resp.StatusCode, http.StatusOK)
    }
    body, _ := io.ReadAll(resp.Body)
    if string(body) != "Hola, Mundo" {
        t.Errorf("body = %q, want %q", body, "Hola, Mundo")
    }
}
```

The full-stack style uses `httptest.NewServer`, which starts a real HTTP server on a random local port and hands you its URL. This is the right tool when you need to exercise a client, middleware, or routing — anything that depends on a genuine round trip.

```
func NewServer(handler http.Handler) *httptest.Server // start a real server on a random port
```

```
func TestGreetServer(t *testing.T) {
    ts := httptest.NewServer(http.HandlerFunc(greet))
    defer ts.Close()  // shut the server down at the end

    resp, err := http.Get(ts.URL + "/?name=Mundo")
    if err != nil {
        t.Fatal(err)
    }
    defer resp.Body.Close()
    body, _ := io.ReadAll(resp.Body)
    if string(body) != "Hola, Mundo" {
        t.Errorf("body = %q, want %q", body, "Hola, Mundo")
    }
}
```

Tip: Prefer `httptest.NewRecorder` for unit-testing a single handler — it is faster and needs no port. Reach for `httptest.NewServer` only when something under test must make a real HTTP request, such as a client library or a reverse proxy.

Race Detector

Chapter 11 introduced the race detector briefly. Here is how to use it in tests.

Run `go test -race` to enable the race detector:

```
go test -race ./...
```

The race detector instruments the binary to track every memory access. If two goroutines access the same variable concurrently and at least one of them is a write, the detector prints a detailed report showing the goroutine stacks at both access sites.

Tip: Run `go test -race` in CI on every push. The race detector has a noticeable runtime cost (typically a 2–20x slowdown and 5–10x more memory), which is acceptable in CI but may be too slow for production. The cost of finding a race in production is far higher than the cost of running a slower test suite.

Trap: The race detector only catches races that **actually occur at runtime**. It cannot prove the absence of races. A test suite with low concurrency coverage might miss a race that the detector would catch under higher load. Write tests that exercise concurrent code paths.

Here is a concise example of a race the detector will catch:

```
func TestCounterRace(t *testing.T) {
    var count int
    var wg sync.WaitGroup

    for i := 0; i < 100; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            count++  // data race: concurrent unsynchronized write
        }()
    }
    wg.Wait()
    t.Log("count:", count)
}
```

This test imports `sync` for `sync.WaitGroup` (see Chapter 11) alongside `testing`. Run this with `go test -race` and the detector reports the race immediately.

Goroutine Leak Detection

Chapter 12 covered goroutine leaks in depth: what causes them, how to prevent them with context cancellation and `done` channels, and how `go.uber.org/goleak` detects them in tests. Every goroutine you spawn should have a clear exit path; if it has none, it is a leak. [***goroutine-must-exit***] The testing-specific summary: place `goleak.VerifyTestMain(m)` in `TestMain` to check every test in the package, or `defer goleak.VerifyNone(t)` in individual tests.

```
// Check all tests in the package --- preferred.
func TestMain(m *testing.M) {
    goleak.VerifyTestMain(m)
}

// Check a single test --- use when you cannot modify TestMain.
func TestBadAppleStream(t *testing.T) {
```

```
    defer goleak.VerifyNone(t)
    // ...
}
```

Tip: `TestMain` is Go's equivalent of JUnit 5's `@BeforeAll` / `@AfterAll` at the package level. It runs once per test binary rather than once per test function — the right place for global setup such as opening a shared database connection or installing goleak.

Wut: `goleak.VerifyTestMain` calls `m.Run()` internally. Do not call `m.Run()` yourself before passing `m` to it — the tests would run twice.

Integration Tests and Build Tags

Unit tests run quickly and need no external dependencies. Integration tests talk to real databases, real HTTP servers, or real message queues — they are slower and require infrastructure. Build tags let you separate the two so that `go test ./...` runs only the fast unit tests by default.

A build tag is a comment at the very top of a file, before the package declaration:

```
//go:build integration

package music_test
```

This file is excluded from the build unless the `integration` tag is provided. To run integration tests:

```
go test -tags=integration ./...
```

To run unit tests only:

```
go test ./...
```

Tip: Use separate CI pipeline stages: one stage runs `go test ./...` on every commit (fast, no infrastructure), and a second stage runs `go test -tags=integration ./...` before merging to main (slower, requires a test database or test container).

Wut: In Go 1.16 and earlier, build tags used a different syntax: `// +build integration` (a comment, not a `//go:build` directive). Go 1.17 introduced the `//go:build` form, which is now the standard. `gofmt` will add the `//go:build` form for you if you only write the old form. Both can coexist in the same file for backward compatibility.

Running Tests

The `go test` command is the entry point for all test-related tasks.

`go test ./...`

Run all tests in the current module:

```
go test ./...
```

The `./...` pattern matches the current directory and all subdirectories.

`-count=1` (Disable Caching)

Go caches test results. If the test sources and dependencies have not changed, `go test` reuses the cached result rather than re-running the tests. This is usually helpful, but sometimes you want to force a fresh run — for example, when testing code that depends on external state.

```
go test -count=1 ./...
```

`-count=1` tells the test runner to run each test exactly once and bypass the cache.

Tip: Use `-count=1` in CI to guarantee that tests always run, even when nothing has changed in the source. Cached test results in CI can hide flaky tests.

`-timeout`

By default `go test` applies a 10-minute timeout to the entire test binary. You can override this:

```
go test -timeout=30s ./...
```

Set a tight timeout in CI so that a hanging test (for example, a goroutine waiting on a channel that is never closed) fails fast rather than blocking your pipeline for 10 minutes.

`-run` and `-bench`

`-run` filters which test functions run by matching against a regular expression. `-bench` does the same for benchmarks.

```
go test -run=TestBetterOffAlone ./...           # run only tests matching "TestBetterOffAlone"
go test -run=TestBetterOffAlone/zero ./...      # run only the "zero" subtest
go test -bench=BenchmarkNormalize ./...     # run only this benchmark
go test -bench=. -benchmem ./...           # run all benchmarks with allocation stats
```

A Typical CI Invocation

A minimal, correct CI test command:

```
go test -race -count=1 -timeout=120s ./...
```

This runs all tests with the race detector, bypasses the cache, and fails if anything takes more than two minutes.

Try It

Time to type something in. The program below pairs a tiny `normalizeTitle` function with a table-driven test, a subtest per case, and a `t.Helper()` assertion — the three things you will use in almost every Go test you write. Save both files in the same directory, then run `go test -v` and watch each subtest report on its own line.

```
// normalize.go
package main

import "strings"

// normalizeTitle trims surrounding spaces and title-cases each word.
func normalizeTitle(s string) string {
    fields := strings.Fields(s)
```

```
    for i, w := range fields {
        fields[i] = strings.ToUpper(w[:1]) + strings.ToLower(w[1:])
    }
    return strings.Join(fields, " ")
}

func main() {}
```

```
// normalize_test.go
package main

import "testing"

// checkTitle reports a mismatch at the caller's line, not this helper's.
func checkTitle(t *testing.T, got, want string) {
    t.Helper()
    if got != want {
        t.Errorf("got %q, want %q", got, want)
    }
}

func TestNormalizeTitle(t *testing.T) {
    cases := []struct {
        name  string
        input string
        want  string
    }{
        {name: "padded",     input: "  paint the town red  ", want: "Paint The Town Red"},
        {name: "all caps",   input: "MONACO",                 want: "Monaco"},
        {name: "mixed case", input: "la BACHATA",             want: "La Bachata"},
        {name: "empty",      input: "",                       want: ""},
    }

    for _, tc := range cases {
        t.Run(tc.name, func(t *testing.T) {
            checkTitle(t, normalizeTitle(tc.input), tc.want)
        })
    }
}
```

Running `go test -v` reports `PASS` for `TestNormalizeTitle` and each named subtest below it. Then try these modifications:

- Add a case that you expect to fail (say, `want: "monaco"`) and confirm the failure output points to the `t.Run` line, not inside `checkTitle` — that is `t.Helper()` at work.
- Add a `BenchmarkNormalizeTitle` and run `go test -bench=. -benchmem` to see ns/op and allocation counts.
- Delete the `t.Helper()` line and re-run the failing case to see the reported line number move into the helper.

Key Points

- Test functions are named `TestXxx` and live in `_test.go` files; no annotations required.
- `t.Error` marks a failure and continues; `t.Fatal` marks a failure and stops the test immediately.
- Table-driven tests use a slice of anonymous structs; `t.Run` creates named subtests for each case.

- `t.Helper()` must be called at the top of any helper function so that failure output points to the call site, not the helper.
- `t.Cleanup(f)` registers teardown that runs when the test ends (LIFO), the idiomatic analog of JUnit's `@AfterEach`; `t.Parallel()` marks a (sub)test to run concurrently.
- Benchmarks are named `BenchmarkXxx` and receive `*testing.B`; prefer the `for b.Loop() { ... }` idiom (Go 1.24), which excludes setup and teardown from timing automatically. `b.N` still works.
- Call `b.ResetTimer()` after setup when you are still writing a `b.N`-style loop.
- Example tests (`ExampleXxx` with a trailing `// Output:` comment) are verified, runnable documentation.
- Test HTTP handlers with `httptest.NewRecorder` (call the handler directly) or `httptest.NewServer` (a real server on a random port).
- Fuzz tests are named `FuzzXxx` and receive `*testing.F`; `f.Add` seeds the corpus, `f.Fuzz` registers the test body.
- Without `-fuzz`, a fuzz test runs only the seed corpus, acting as a standard test.
- Run `go test -race` in CI on every push to catch data races; the race detector only reports races that actually occur.
- `goleak.VerifyTestMain(m)` in `TestMain` detects goroutine leaks after the test suite finishes.
- Use `//go:build integration` to gate slow integration tests; run them with `-tags=integration`.
- `go test ./...` runs all tests; `-count=1` disables caching; `-timeout` caps the run time.

Exercises

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`.

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")
}
```

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?

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")
}
```

5. **Write a program:** Write a table-driven test for the following function. 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
```