Testing

Test stdlib, testsets, test items, Aqua, JET

The stdlib

```
using Test

@testset "Money" begin
    @test Money(100) + Money(50) == Money(150)
    @test Money(0) == zero(Money)
    @test_throws ArgumentError Money(-1)
    @test_throws "below absolute zero" Celsius(-300)   # match the message (1.8+)
    @test 0.1 + 0.2 ≈ 0.3
    @test isapprox(a, b; atol = 1e-10)
    @test_broken known_bug()                  # must fail; shouts if it starts passing
    @test_skip flaky()
    @test_logs (:warn, "retrying") do_thing()
    @test_nowarn quiet_thing()
    @inferred mysum(v)                                  # asserts type stability
end
```

`@testset` catches failures, keeps going, and prints a summary table at the end. Nesting works and the summary is nested too.

Parameterised testsets — a `for` clause on the testset itself, which re-runs the body per value and labels each:

```
@testset "roundtrip $T" for T in (Int, Float64, Rational{Int})
    @test parse_as(T, string(one(T))) == one(T)
end

@testset "$name" for (name, input, expected) in [
        ("empty", "", 0),
        ("single", "1", 1),
        ("many", "1,2,3", 6),
    ]
    @test parse_sum(input) == expected
end
```

Layout and running

```
test/
├── Project.toml        # test-only dependencies
├── runtests.jl
├── test_parsing.jl
└── test_arithmetic.jl
```

```
# test/runtests.jl
using MyPkg, Test

@testset "MyPkg" begin
    include("test_parsing.jl")
    include("test_arithmetic.jl")
end
```

Run with `] test` (fresh process, uses `test/Project.toml`), or `Pkg.test("MyPkg")`. Individual files run directly with `julia --project=test test/test_parsing.jl` if they're self-contained.

Test dependencies go in `test/Project.toml` (modern) or the `[extras]`/`[targets]` sections of the root `Project.toml` (older style; still seen everywhere).

Test items — running one test from the editor

```
using TestItems

@testitem "addition" begin
    @test 1 + 1 == 2
end

@testitem "needs setup" setup=[SharedFixtures] tags=[:slow] begin
    @test length(SharedFixtures.data) == 100
end

@testmodule SharedFixtures begin
    const data = collect(1:100)
end
```

Test items can live *next to the code they test* in `src/`, and each runs in its own module. The point is editor integration: JETLS and the VSCode extension put a run button on each `@testitem`, so you execute one test without a full `] test` cycle. Run them in bulk with `ReTestItems.runtests()`, which also parallelises across processes.

If you're starting a package now, this is the layout to pick.

Package-quality checks

```
using Aqua
Aqua.test_all(MyPkg)
```

Catches: method ambiguities, type piracy, unbound type parameters, stale dependencies, missing `compat` entries, undefined exports. Cheap to add, catches a class of bug that tests never will.

```
using JET
JET.test_package(MyPkg; target_defined_modules = true)
```

Static analysis: possible `MethodErrors`, undefined variables, type errors on paths your tests didn't exercise. Noisy on first run; worth triaging once.

Doctests

Examples in docstrings, executed as tests:

````
"""
    double(x)

# Examples
```jldoctest
julia> double(2)
4
```
"""
double(x) = 2x
````

```
using Documenter
DocMeta.setdocmeta!(MyPkg, :DocTestSetup, :(using MyPkg); recursive = true)
doctest(MyPkg)
```

Your documentation examples then cannot rot silently, which is worth more than it sounds.

Property-based and fuzz testing

```
using Supposition        # or PropCheck.jl

@check function roundtrip(s = Data.Text(Data.Characters()))
    parse_it(render_it(s)) == s
end
```

Not as mature as StreamData or Hypothesis, but usable for parser/serialiser pairs and numerical invariants.

Coverage

```
julia --project --code-coverage=user -e 'using Pkg; Pkg.test()'
```

```
using Coverage
coverage = process_folder()
covered, total = get_summary(coverage)
```

In CI, `julia-actions/julia-processcoverage` plus Codecov is the standard chain.

CI

The `julia-actions` set of GitHub Actions is the ecosystem default; they also run under Forgejo Actions and Gitea Actions, since those consume GitHub Action definitions.

```
- uses: julia-actions/setup-julia@v2
  with:
    version: '1'          # '1' = latest 1.x, 'lts', or '1.10'
- uses: julia-actions/cache@v2
- uses: julia-actions/julia-buildpkg@v1
- uses: julia-actions/julia-runtest@v1
- uses: julia-actions/julia-processcoverage@v1
```

Matrix over `['1', 'lts', 'nightly']` and over OS if you have platform-specific code. Allow `nightly` to fail.

Test-writing notes specific to Julia

- **`@inferred`** is a test, not a benchmark. Wrapping your core functions in `@inferred` in the test suite catches type instability regressions automatically, which is the cheapest performance regression test available.
- Comparing floats: use `≈` or `isapprox` with an explicit tolerance. `==` on floats in a test is almost always a latent failure.
- Testing errors: prefer `@test_throws SpecificError` over `@test_throws Exception`.
- A `@testset` swallows the return value of its body, so don't rely on it evaluating to anything.
- Randomness: seed it (`Random.seed!(42)`) inside the testset, or use `StableRNGs.jl` — the default RNG algorithm is allowed to change between Julia versions, so `seed!` alone is not reproducible across releases.
- Tests run in a separate process from your REPL session, so a passing `] test` after a failing interactive run usually means you had stale state in the REPL, not a flaky test.