Julia cheat sheet

For the experienced polyglot, 2026

2026-08-26

Contents

Introduction

Written for someone who already writes Go, Elixir, Python, Java and Node, and is starting Julia in 2026 (Julia 1.12 era). It assumes you know what a loop is; it does not assume you know what a `Vector{<:Real}` is or why `A[:, 1]` is faster than `A[1, :]`.

Section 01 gets a working environment. Read 02 and 03 back to back, then write something. Come back to the rest as needed.

Conventions used here

- `julia>` prompts are omitted; code blocks are things you can paste.
- `#` comments after a line show the result where useful.
- Anything marked **trap** is a mistake most people make exactly once.

The one-paragraph orientation

Julia is a dynamically typed language with an aggressive JIT, whose organising abstraction is multiple dispatch rather than objects or modules. Types exist mainly to select methods and to let the compiler generate specialised machine code; you can ignore them and the program still runs, but you'll leave a factor of 50 on the table. There is no class, no inheritance of data, no interface keyword, no null, and no BEAM. The standard library is large and numerics-heavy. Packages are just modules with a `Project.toml`, and the package manager is genuinely good.

01. Setting up

Linux, August 2026. Current stable Julia is **1.12.6**, LTS is **1.10.11**, and **1.13** is in release candidates.

Toolchain

```
curl -fsSL https://install.julialang.org | sh
```

That installs `juliaup` and the current stable Julia into `~/.juliaup`, and puts a `julia` shim on your PATH. Nothing else is required — Julia binaries are self-contained tarballs with their own LLVM, BLAS and libgit2. No compiler toolchain, no system BLAS, no Python.

Two environmental notes that aren't obvious:

- **Headless plotting.** The default `GR` backend links against X libraries and will try to open a window. If the machine has no display, `export GKSwstype=100` makes it write to file instead. Without it you get an obscure failure deep inside a plotting call, and you will not guess why.
- **Don't install Julia from your distro's repository.** Distro packages lag badly and confuse `juliaup`'s shim. If one is already installed, remove it.

Version management: juliaup, not asdf

```
juliaup add lts              # a channel, not a pin — tracks LTS as it moves
juliaup add 1.13-rc
juliaup add nightly
juliaup status
juliaup update               # updates every channel you track
juliaup default release

julia +lts                   # run a specific channel ad hoc, no switching
julia +1.10 --project -e 'using Pkg; Pkg.status()'
```

`julia +channel` is the feature that makes this worth using over a generic version manager. Testing a package against LTS and stable in the same terminal is one token of difference, not an `asdf set` dance and back. Channels also mean patch releases arrive with `juliaup update` and no version strings to edit.

The reproducibility argument for `.tool-versions` is weaker here than elsewhere, because Pkg already covers it a layer down: `Project.toml` carries a `[compat]` bound on `julia` itself, and `Manifest.toml` pins the entire resolved dependency graph.

If you want one tool across all your runtimes anyway, there's a community plugin:

```
asdf plugin add julia https://github.com/rkyleg/asdf-julia.git
asdf install julia 1.12.6
asdf set julia 1.12.6            # post-Go-rewrite command; `asdf local` is gone

mise use julia@1.12.6            # mise reads asdf's plugin registry
```

What that costs you: channels, `julia +ver`, and same-day patch releases (you wait for the plugin maintainer). It also adds a shim layer that occasionally confuses tools which shell out to `julia` — the `jetls` and `runic` executables installed by `Pkg.Apps` are the usual casualties.

The hybrid worth adopting: juliaup for the toolchain, direnv (or `mise`'s `[env]` block) for per-project environment.

```
# .envrc in a project root
export JULIA_PROJECT=@.
export JULIA_NUM_THREADS=auto
```

Then bare `julia` in that directory is already in the right project with the right thread count, and Zed picks the `.envrc` up automatically.

Environment variables

Variable	Why it matters
`JULIA_NUM_THREADS=auto`	Default is **1**. Everything "multithreaded" silently runs serially otherwise.
`JULIA_PROJECT=@.`	Activates the nearest `Project.toml`, walking upward. Without it you're installing into a global environment.
`PATH` += `~/.julia/bin`	**Required.** `Pkg.Apps` installs `jetls`, `runic`, `jlfmt` here.
`JULIA_DEPOT_PATH`	Relocate the depot. Colon-separated; the first entry is where writes go.
`JULIA_PKG_SERVER`	Point at a self-hosted package server (see below).
`JULIA_EDITOR`	What `@edit` and `InteractiveUtils.edit` open. `"zed --wait"`, `"nvim"`, etc.
`GKSwstype=100`	Headless plotting, as above.

The depot

`~/.julia` holds registries, package source, and precompilation caches — large, written constantly, and mmap'd.

- **Never put it on a network mount.** SMB in particular: precompile caches are lock-sensitive and you get corruption plus glacial startup. Local disk only. Back up `Project.toml`/`Manifest.toml` instead — everything else is reconstructible with `] instantiate`.
- Give each VM or container its own depot rather than sharing one. Multiple Julia *versions* share a depot fine (they namespace by version); multiple *machines* do not.
- `] gc` reclaims space from package versions nothing references any more. It grows to tens of gigabytes if you never run it.

Editors

The landscape, honestly

`julia-vscode` remains the most featureful integration by a distance: inline evaluation, a workspace/variable explorer, an integrated plot pane, a profiler viewer, and the only working step debugger. Everything below trades some of that away.

Editor	LSP	Inline eval	Plot pane	Debugger	Notes
VSCode + julia-vscode	LanguageServer.jl	yes	yes	yes	the benchmark
Zed	both, JETLS via dev extension	yes, via Jupyter kernel	via ZedPlotPane or Jupyter	no	fast, remote-over-SSH is excellent
Neovim	either, trivially configured	via a terminal plugin	terminal or file-watch	no	best LSP flexibility; most assembly required
Helix	either, four lines of TOML	no	no	no	zero-config, zero-extension
Emacs + eglot	either	julia-repl / julia-snail	julia-snail	no	julia-snail is the closest thing to the VSCode experience outside VSCode
Pluto.jl	no	inherent	inherent	no	not an editor; a different way to work (see below)

Zed and Neovim both have vim keybindings if that's the draw — Zed's vim mode is good, and it also ships a Helix mode.

The new language server

JETLS is built on the compiler's own machinery — JET.jl for abstract interpretation, JuliaLowering.jl for byte-precise, macro-aware lowering. That gets you type-sensitive diagnostics and go-to-definition that works through macros, neither of which `LanguageServer.jl` can do structurally. It's officially experimental, and the pace of change is high.

```
# needs Julia >= 1.12.2 and < 1.14 — check with `julia --version` first
julia -e 'using Pkg; Pkg.Apps.add(;
    url="https://github.com/aviatesk/JETLS.jl", rev="release")'
jetls --help          # verifies installation and that ~/.julia/bin is on PATH
```

Three things to know before you commit to it:

1. Installation precompiles JETLS and its dependencies — several minutes, once. Do this *before* first opening a `.jl` file in your editor, or the LSP `initialize` request can time out while it compiles.
2. **The `jetls` app is pinned to the Julia version that installed it.** After every `juliaup update` to a new minor, re-run the install command or the binary stops working. Put it in a shell function so you don't forget.
3. JETLS analyses by *running* parts of your code, including dependencies. Don't point it at a repository you don't trust.

To pin a version rather than track the release branch: `rev="2026-08-07"` (release tags are dates).

Update the same way you installed:

```
julia -e 'using Pkg; Pkg.Apps.add(;
    url="https://github.com/aviatesk/JETLS.jl", rev="release")'
```

Zed

The published extension:

1. `zed: extensions` from the command palette, search `julia`, install, restart.
2. Make sure `julia` is on PATH — the extension looks in standard locations.

That gives you syntax, tasks, and `LanguageServer.jl`. Configure its linting in `~/.config/zed/settings.json`:

```
{
  "lsp": {
    "julia": {
      "settings": { "julia.lint.missingrefs": "none" }
    }
  }
}
```

JETLS in Zed needs the dev-extension route, because Zed can only register a language server through an extension — unlike Helix or Neovim, you can't point it at an arbitrary LSP binary from settings. The JETLS-enabled fork lives on a branch:

```
# prerequisite: Rust via rustup specifically (a package-manager Rust will NOT work;
# Zed compiles extensions to wasm32-wasip2 and needs rustup to add the target)
git clone -b avi/JETLS https://github.com/aviatesk/zed-julia ~/src/zed-julia-jetls
```

Then in Zed: command palette ⟶ `zed: install dev extension` ⟶ select that directory. It overrides the published Julia extension, and the extensions page will say so. Check the branch README for current specifics — it moves, and it will eventually be merged upstream, at which point all of this collapses to "install the extension".

Debugging a failed install: `zed: open log`, or relaunch with `zed --foreground` for verbose output.

Inline evaluation in Zed. The zed-julia README says there's no inline execution, and for the extension's own integration that's true — but Zed has a separate built-in REPL that speaks Jupyter kernels, and Julia is a supported language:

```
using Pkg; Pkg.add("IJulia")     # registers a Julia kernelspec with Jupyter
```

Then `repl: run` (bind it, or `ctrl-shift-enter`) evaluates the selection, line, or block and renders the result inline beneath it. `repl: sessions` shows what's running; `repl: refresh kernelspecs` after installing a new kernel.

Understand what this is, though: a **Jupyter kernel**, not your `julia --project` session. State lives in the kernel, Revise isn't in the loop by default, and it's a separate process from any terminal REPL you have open. It's excellent for exploration and plots; it is not the edit-save-and-the-running-session-updates workflow. For that, keep a terminal REPL with Revise and use the send-selection bindings below.

Extra threads for the kernel:

```
using IJulia
IJulia.installkernel("Julia (auto threads)", env = Dict("JULIA_NUM_THREADS" => "auto"))
```

Terminal REPL send-selection bindings, for the Revise workflow:

```
// ~/.config/zed/keymap.json
[
  {
    "context": "Terminal",
    "bindings": { "ctrl-shift-`": "terminal_panel::ToggleFocus" }
  },
  {
    "context": "Editor && mode == full",
    "bindings": {
      "ctrl-shift-`": "workspace::OpenInTerminal",
      "ctrl-shift-f11": ["action::Sequence", [
        "editor::SelectEnclosingSymbol",
        "editor::CopyAndTrim",
        ["workspace::SendKeystrokes", "ctrl-` ctrl-shift-v enter ctrl-shift-`"]
      ]]
    }
  }
]
```

Adjust the paste keystroke to your terminal. Start the REPL with `julia --project -t auto`.

Plot pane without Jupyter:

```
using Pkg; Pkg.add("ZedPlotPane")
using ZedPlotPane      # load BEFORE Plots or any other plotting package
```

First plot creates `~/.cache/zed-julia/current-plot.png`; drag that tab into a side pane and it updates in place.

Zed as Julia's editor, so `@edit` opens the right file at the right line:

```
# ~/.julia/config/startup.jl
atreplinit() do repl
    InteractiveUtils.define_editor("zed") do cmd, path, line, column
        `$cmd $path:$line:$column`
    end
end
```

with `export JULIA_EDITOR="zed --wait"`.

Remote work. If the machine you compute on is headless, run Zed locally and use its SSH remote projects. The LSP, terminal, and tasks all execute on the remote host; only the UI is local. Install juliaup, the depot and `jetls` **on the remote machine** — the local Zed needs no Julia at all. This is materially better than the alternatives for a headless workstation.

Neovim

The most flexible LSP setup of the lot, because you point it directly at the binary.

```
-- JETLS (Neovim 0.11+)
vim.lsp.config("jetls", {
  cmd = { "jetls", "serve" },
  filetypes = { "julia" },
  root_markers = { "Project.toml" },
})
vim.lsp.enable("jetls")
```

`--stdio` instead of the default socket if you're editing over TRAMP-like remote schemes; sockets are the more stable transport otherwise.

Worth installing alongside:

- **`julia-vim`** — the LaTeX-to-Unicode tab substitution (`\alpha<tab>` becomes α) that the REPL has built in. Without it, typing mathematical Julia is painful. Also block-wise `%` matching for `if`/`end`.
- **`iron.nvim`**, **`vim-slime`**, or **`toggleterm`** — send-region-to-REPL. Slime is the simplest and works with tmux, which pairs well with a long-lived Julia session.
- **ZedPlotPane's equivalent**: `using UnicodePlots` for terminal plots, or watch the PNG with an image-capable terminal.

Helix

Four lines and no extension system to fight:

```
# ~/.config/helix/languages.toml
[[language]]
name = "julia"
language-servers = ["jetls"]

[language-server]
jetls = { command = "jetls", args = ["serve"], timeout = 60 }
```

The `timeout = 60` matters: Helix aborts `initialize` after 20 seconds by default, and JETLS's first startup compiles enough Julia code to blow past that on a cold cache. Without it you get `request 0 timed out` and no server.

Emacs

```
(with-eval-after-load 'eglot
  (add-to-list 'eglot-server-programs
               '(((julia-mode :language-id "julia")
                  (julia-ts-mode :language-id "julia"))
                 "jetls" "serve" "--socket" :autoport)))
```

Use `--stdio` rather than the socket if you edit over TRAMP. Add `julia-snail` for REPL integration — it's the closest thing to the VSCode experience outside VSCode, with module-aware evaluation and a plot pane.

Formatting, linting, analysis

```
julia -e 'using Pkg; Pkg.Apps.add("Runic")'          # opinionated, zero config
julia -e 'using Pkg; Pkg.Apps.add("JuliaFormatter")' # configurable, installs `jlfmt`
```

Both land in `~/.julia/bin`. **Runic** is JETLS's default and has no options at all, which ends the formatting argument permanently — pick it unless you have a specific reason not to. To switch, put a `.JETLSConfig.toml` in your project root:

```
formatter = "JuliaFormatter"     # or "Runic" (default)
```

Static analysis and package hygiene are covered in section 17, but install them now so they're there: `JET.jl` (type-level bug finding, the Dialyzer analogue), `Aqua.jl` (ambiguities, piracy, stale deps). JETLS also ships a diagnostic CLI, so the same analysis runs in CI without an editor.

REPL setup

Detail is in section 20; the minimum to do today:

```
julia -e 'using Pkg;
    Pkg.activate("v$(VERSION.major).$(VERSION.minor)"; shared=true);
    Pkg.add(["Revise", "OhMyREPL", "BenchmarkTools"])'
```

These go in the **shared** default environment, not in any project, so they're available everywhere without being anyone's dependency.

```
# ~/.julia/config/startup.jl
try
    using Revise
catch e
    @warn "Revise failed to load" exception = (e, catch_backtrace())
end
```

Revise is not optional. Julia's first-call compilation makes restarting a session expensive, so the workflow is: keep one REPL alive for hours, edit files, let Revise apply the changes. The exception is redefining a `struct`, which still requires a restart.

Self-hosted, Julia-adjacent

If you run your own infrastructure, four of these earn their keep.

Pluto.jl — reactive notebooks; the Livebook analogue, and a close one. No hidden state: the program state is always fully described by the visible code, and changing a cell re-runs its dependents. It manages a package environment per notebook automatically, and notebooks are plain `.jl` files that also run as scripts, which behaves better in git than `.livemd` or `.ipynb`.

```
using Pkg; Pkg.add("Pluto")
using Pluto; Pluto.run(host = "0.0.0.0", port = 1234)
```

A Pluto server is arbitrary code execution by design — private network only. For a container deployment, bake the depot and precompilation into the image layer; that's the slow part, and doing it at build time turns a two-minute cold start into seconds.

PlutoSliderServer.jl — runs only the `@bind` parts of notebooks, statelessly, over plain HTTP GET. Visitors get working sliders and recomputed results; they can't edit code, and there's no per-visitor Julia session. It watches a git repo and re-exports on push. This has no Livebook equivalent, and it's what makes Pluto interesting as a hosted service rather than a local tool. `PlutoPages.jl` goes further and renders a directory of notebooks into a static site.

LocalRegistry.jl + LocalPackageServer.jl — a private registry (just a git repo) so `] add MyInternalThing` works, plus a package server that serves it *and* caches the General registry. Point `JULIA_PKG_SERVER` at it and every machine on your network pulls each package once. Small, TOML-configured, and it makes CI and multi-VM setups noticeably faster.

Genie / Stipple — Phoenix and LiveView, respectively: full-stack MVC with an ORM, plus a reactive UI layer holding state server-side and syncing over JSON to a Vue3/Quasar client. Genie Builder, the visual editor, is a VSCode plugin — irrelevant here; the low-code API is fine hand-written.

Also worth knowing about: **Oxygen.jl** (Plug-shaped minimal HTTP), **Franklin.jl** (static site generator with live Julia evaluation in pages), **Documenter.jl** (the docs standard), **IJulia** (Jupyter kernel — already needed above for Zed's REPL), **Dagger.jl** (distributed task scheduler).

Verification

```
juliaup status
julia --version                      # 1.12.6
julia +lts --version                 # 1.10.11
echo $PATH | tr ':' '\n' | grep .julia/bin
which jetls runic
julia -e 'using Revise; println("revise ok")'
julia -t auto -e 'println(Threads.nthreads())'
julia -e 'using IJulia; println("kernel ok")'
```

In the editor: open a `.jl` file, confirm the LSP reports as running, hover a function and check you get a docstring, and introduce a deliberate type error to confirm diagnostics arrive.

02. Syntax basics

The parts you'd otherwise pick up by osmosis, compressed. If a construct behaves exactly like its equivalent in Python or Go, it gets one line.

Variables and assignment

```
x = 5                    # no declaration keyword, no type needed
x::Int = 5               # optional type assertion on the binding (1.8+)
const MAX = 100          # binding cannot be reassigned (value may still mutate)
a = b = 0                # chained
a, b = 1, 2              # tuple destructuring
a, b = b, a              # swap
x += 1; x *= 2           # compound assignment (no ++ or --)
_ = f()                  # conventional throwaway
```

Naming convention: `lower_snake` for variables and functions (often just `lowercase` with no separator for short names), `UpperCamel` for types and modules, `SCREAMING` for constants only when it aids readability, trailing `!` for functions that mutate their arguments.

Unicode identifiers are legal and idiomatic in numerical code: `α`, `Δt`, `∇f`, `σ²`. See section 20 for how to type them.

Comments

```
# line comment

#=
block comment,
which nests properly
=#

"""
    myfunc(x, y)

Docstrings go *before* the definition and are Markdown. They're
retrievable with `?myfunc` and become the package docs.
"""
myfunc(x, y) = x + y
```

Numbers and literals

```
1                # Int64 on 64-bit
1.0              # Float64
```

```
1f0                 # Float32
0x1f                # UInt8 — hex literals are unsigned, width from digit count
0b1010              # UInt8 binary
1_000_000           # underscores as digit separators
1//3                # Rational{Int}
1 + 2im             # Complex{Int}
big"123456789012345678901"   # BigInt
3.0e-5
Inf, -Inf, NaN
2x                  # numeric literal juxtaposition: means 2*x. So is 3(a+b).
```

Type conversion and parsing:

```
Int(3.0)            # 3 — errors if not exactly representable
round(Int, 3.7)     # 4;  floor/ceil/trunc take a type too
float(3)            # 3.0
parse(Int, "42")
parse(Float64, "1.5")
tryparse(Int, "x") # nothing instead of throwing
string(42)
```

Operators

```
+  -  *  /  \  ^  %        # \ is left division: A \ b solves Ax = b
÷                          # integer division, also div(a,b)
==  !=  <  <=  >  >=       # comparison; chainable: 1 < x <= 10
===  !==                   # identity / egal
&&  ||  !                  # short-circuiting boolean
&  |  ⊻  ~  >>  <<         # bitwise (xor is ⊻ or xor(a,b))
∈  ∉  ⊆                    # in, not-in, subset (ASCII: in, issubset)
```

^ is exponentiation, never xor. && and || short-circuit and are commonly used as one-line guards:

```
x < 0 && throw(ArgumentError("negative"))
isempty(v) || process(v)
```

Ternary and the "return this or that" forms:

```
y = x > 0 ? "pos" : "nonpos"
y = ifelse(x > 0, a, b)     # evaluates both branches, no branch misprediction
```

Control flow

```
if x > 0
    ...
elseif x < 0
    ...
else
    ...
end

for i in 1:10 ... end
for (i, v) in enumerate(v) ... end
for i in 1:3, j in 1:3 ... end        # nested, one `end`, j varies fastest
while cond ... end
break; continue
```

There is no `switch`/`case`. Options: an if-chain, a `Dict` of functions, or dispatch on a type or a `Val`. `Match.jl` provides pattern matching if you miss Elixir's `case`.

No `do...while`. Use `while true ... cond || break; end`.

Blocks and scope

```
begin
    a = 1
```

```
    b = 2
end             # groups expressions, does NOT create a scope; value is last expr

let x = 1
    x + 1       # DOES create a scope; useful for capturing loop variables
end

(a = 1; a + 1)  # semicolon form of begin/end, value is last expr
```

Scope rules that matter:

- `function`, `for`, `while`, `let`, `struct`, `module`, comprehensions: **new scope**.
- `if`, `begin`, `&&`: **no new scope**. Variables assigned inside leak out.
- Inside a function, assigning to a name that exists globally creates a *local* unless you write `global x = ...`.
- **trap:** in a script (not the REPL), a `for` loop at top level cannot assign to a global without `global`. This is the single most common "why is my counter zero" question.

```
total = 0
for i in 1:10
    global total += i       # required at top level in a script
end
```

Everything is an expression

Assignments, `if`, `for` (returns `nothing`), and blocks all have values.

```
x = if cond; 1 else 2 end
f(x) = (y = 2x; y + 1)
```

Functions return the value of their last expression; `return` is optional but clearer in anything longer than a line.

Printing and output

```
print("no newline")
println("with newline")
println("interpolated: $x and $(x + 1)")
printf_style = @sprintf("%.3f", π)     # using Printf
@printf("%5.2f %s\n", 1.5, "ok")
display(x)                             # rich display (arrays get pretty-printed)
show(x)                                # developer-facing repr
@show x y                              # debug: prints "x = 1" and "y = 2", returns values
@info "message" key = value            # structured log to stderr
@warn "careful"; @error "bad"
```

`println` writes to stdout; `@info`/`@warn`/`@error` write to stderr with source location, which is what you want in libraries.

Functions, the short version

```
function add(x, y)
    x + y
end

add(x, y) = x + y            # assignment form, for one-liners
add(x, y::Int = 0) = x + y   # default argument
add(x; scale = 1) = scale*x  # keyword argument (after the semicolon)
(x -> x^2)(3)                # anonymous
```

Arguments are **passed by sharing**: you get a reference to the same object, so mutating an array argument is visible to the caller, but rebinding the name isn't. Same as Python or JS.

Full detail in section 07.

Collections, the short version

```
v = [1, 2, 3]                  # Vector{Int64}
m = [1 2; 3 4]                 # 2x2 Matrix — space separates columns, ; rows
```

```julia
t = (1, "two", 3.0)             # Tuple, heterogeneous, immutable
nt = (a = 1, b = 2)             # NamedTuple
d = Dict("a" => 1, "b" => 2)    # Dict{String, Int64}
s = Set([1, 2, 2])              # Set{Int64} with 2 elements
r = 1:10                        # UnitRange, lazy
r2 = 0:0.5:2                    # StepRange
```

Detail in section 08.

Type annotations, the short version

```julia
f(x::Int) = ...                 # method signature: this is DISPATCH, not a check
x::Int = 5                      # binding assertion: converts and enforces
v = Int[]                       # typed empty vector
Vector{Float64}(undef, 10)      # uninitialised, 10 elements
```

Annotating function arguments does not make code faster. Annotating struct fields does. Section 04.

Running Julia

```julia
julia                           # REPL
julia script.jl arg1 arg2       # ARGS == ["arg1","arg2"]
julia --project                 # activate the Project.toml in this dir tree
julia --project -t auto         # ... with all threads
julia -e 'println(1+1)'         # one-liner
julia -i script.jl              # run then drop into REPL
```

A script has no `main` and no entry point ceremony; top-level code runs top to bottom. The `if abspath(PROGRAM_FILE) == @__FILE__` idiom is the `if __name__ == "__main__"` equivalent, if you want a file that's both script and library.

03. Gotchas

Places where the correct Julia answer is not the one your fingers will type.

Indexing and memory layout

```julia
a = [1, 2, 3]
a[1]              # 1-based
a[end]; a[end-1]; a[2:end]
a[0]              # BoundsError

A = rand(1000, 1000)
A[:, 1]           # a column — CONTIGUOUS in memory, fast
A[1, :]           # a row — strided, slow
```

Julia is **column-major**, like Fortran, MATLAB and R; not row-major like C, NumPy or Go. Consequences:

- The innermost loop should vary the *first* index.
- `A[:, j]` is the cheap slice.
- Reshaping and flattening go down columns: `vec([1 2; 3 4]) == [1, 3, 2, 4]`.

```julia
for j in axes(A, 2), i in axes(A, 1)    # correct order
    A[i, j] = f(i, j)
end
```

Copies, references and views

```julia
a = [1, 2, 3]
b = a             # SAME object
b[1] = 99         # a is now [99, 2, 3]
c = copy(a)       # shallow copy
d = deepcopy(a)   # recursive

s = a[2:3]        # slicing ALLOCATES A COPY (unlike NumPy)
v = @view a[2:3]  # a view, no allocation, writes through to a
@views begin ... end   # every slice in this block becomes a view
```

trap: coming from NumPy you expect slices to be views; coming from Go you expect subslices to share. Julia copies. In hot code this is the number one source of surprise allocation.

Integer arithmetic

```
7 / 2          # 3.5 — always Float64, even Int/Int
7 ÷ 2          # 3   — integer division, div(7,2)
7 % 2          # 1   — rem, sign follows the dividend
mod(-7, 2)     # 1   — sign follows the divisor; rem(-7, 2) == -1
2^10           # 1024 — ^ is power
typemax(Int) + 1   # typemin(Int). SILENT WRAPAROUND, no exception.
```

If overflow matters, use `BigInt`, `Int128`, `SafeInt` from SaferIntegers.jl, or `Base.Checked.checked_add`.

`^` on integers with a negative exponent throws — `2^-1` is a `DomainError`. Write `2.0^-1` or `1//2`.

Equality

Operator	Semantics	`NaN`	Use for
`==`	value equality, may be `missing`	`NaN == NaN` is false	general comparison
`===`	identity / bit-identical	true for the same object	checking "same object"
`isequal`	like == but total; hash-consistent	`isequal(NaN, NaN)` is true	Dict keys, sorting
≈ (`isapprox`)	approximate float equality	—	tests, numerics

```
0.1 + 0.2 == 0.3      # false
0.1 + 0.2 ≈ 0.3       # true (\approx<tab>)
isapprox(a, b; atol = 1e-8, rtol = 1e-5)
```

Strings are UTF-8 and indexed by byte

```
s = "héllo"
length(s)        # 5 characters
ncodeunits(s)    # 6 bytes
s[1]             # 'h'
s[2]             # 'é'  (2 bytes)
s[3]             # StringIndexError — index 3 is mid-character
s[nextind(s, 2)] # 'l' — the correct way to step
collect(s)       # Vector{Char} if you need O(1) random access
for c in s ... end   # iteration is always character-wise and correct
```

Also: `"a" * "b"` concatenates; `+` on strings is an error. Repetition is `"ab"^3`.

Scope in loops at top level

```
# in a script:
count = 0
for i in 1:3
    count += 1        # UndefVarError — `count` here is a new local
end
```

Fix with `global count += 1`, or wrap in a function (better). The REPL has a softer rule since 1.5, so this bites only in scripts — which makes it worse, because it works when you test it interactively.

Mutation convention

A trailing `!` means "mutates the first argument". This is convention, not enforcement, but the entire ecosystem obeys it.

```
sort(v)      # returns a sorted copy
sort!(v)     # sorts in place, returns v
push!, append!, empty!, filter!, map!, replace!, mul!, copyto!
```

Argument passing

Pass-by-sharing. Rebinding inside a function does nothing to the caller; mutating does.

```
function f(v)
    v[1] = 99      # visible to caller
    v = [0, 0]     # NOT visible — rebinds the local name
end
```

Immutable types (`Int`, `Float64`, `Tuple`, non-`mutable struct`) cannot be mutated at all, so passing them is always safe.

Type annotations don't do what you think

```
f(x::Float64) = x^2     # a dispatch rule, not a performance hint
f(x) = x^2              # exactly as fast when called with a Float64
```

Julia compiles a specialised version per concrete argument type either way. Annotate for correctness, dispatch and documentation. The place annotations *do* matter for speed is struct fields (section 06) and, occasionally, `Ref`/container element types.

Vector{Int} is not a Vector{Real}

```
Vector{Int} <: Vector{Real}     # false — parametric types are INVARIANT
Vector{Int} <: Vector{<:Real}   # true
```

Write `f(v::AbstractVector{<:Real})`, not `f(v::Vector{Real})`. The latter accepts almost nothing. See section 04.

Ranges are lazy

```
r = 1:1_000_000     # 8 bytes, not 8 MB
collect(r)          # now it's 8 MB
sum(1:1_000_000)    # O(1), it uses the closed form
```

`1:n` where `n < 1` is empty, not an error, and not reversed. Descending is `n:-1:1` or `reverse(1:n)`.

Comprehension and generator parens

```
[f(x) for x in v]      # Vector, materialised
(f(x) for x in v)      # lazy generator, allocates nothing
sum(f(x) for x in v)   # no intermediate array — prefer this in hot code
```

Precompilation and first-call latency

The first call to any function compiles it. A fresh session plotting something can take ten seconds; the second plot is instant. This is normal, not a hang. The workflow implication is significant: **keep one REPL alive for hours and use Revise**, rather than re-running a script. Restarting is the Julia equivalent of a clean rebuild.

Similarly, `] add SomeBigPackage` triggers precompilation of the whole dependency graph, which can take minutes once and then never again.

Threads default to one

```
Threads.nthreads()    # 1, unless you started with -t auto
```

Nothing warns you. Everything "multithreaded" silently runs serially. Set `JULIA_NUM_THREADS=auto` in your shell profile.

Project environments are not the default

Plain `julia` puts you in the shared `@v1.12` environment, where installs are global. Use `julia --project` or set `JULIA_PROJECT=@.` so you're always in the nearest project. Otherwise you'll build a giant global environment whose dependency resolution eventually deadlocks.

Small things

- `1:3 == [1,2,3]` is **false** (different types); `1:3 == 1:3` is true; `collect(1:3) == [1,2,3]` is true.
- `nothing` prints as nothing at the REPL — a function returning `nothing` looks like it returned silently.
- `Dict` iteration order is unspecified and changes between runs; use `OrderedDict` (DataStructures.jl) or sort the keys.
- `x = [1,2,3]; x .+= 1` mutates in place; `x = x .+ 1` allocates a new array.
- Adjacent numeric literal and identifier means multiplication: `2π`, `3x`. So you cannot name a variable `2x`, and `x2` is fine but `2x` is `2*x`.

- The empty array literal `[]` is `Vector{Any}`. Write `Int[]` or `Float64[]`.
- `end` inside indexing means "last index of this dimension", and is also the block terminator. Context disambiguates.

04. Types

Julia's type system exists to select methods and to let the compiler specialise. It is not there to prove your program correct, and it will not stop you at compile time.

The lattice

```
Int64 <: Signed <: Integer <: Real <: Number <: Any
Float64 <: AbstractFloat <: Real
String <: AbstractString
Nothing, Missing, Bool, Char, Symbol
```

```
Int <: Real          # true — subtype operator
isa(3, Real)         # true
typeof(3)            # Int64
supertype(Int64)     # Signed
subtypes(Integer)    # list direct children
```

Two kinds of type:

- **Abstract** — no fields, cannot be instantiated, exists only as a node in the dispatch tree. `Number`, `AbstractArray`, your own `abstract type Shape end`.
- **Concrete** — has a memory layout, can be instantiated, and is *final*: nothing can subtype a concrete type. There is no inheriting from `Int64`, and no inheriting fields from another struct.

That last point is the big structural difference from Java or Python. **Julia has no implementation inheritance.** You inherit behaviour by dispatching on a shared abstract supertype, and you share data by composition (embedding a struct as a field) plus method forwarding.

```
abstract type Animal end

struct Dog <: Animal
    name::String
end
struct Cat <: Animal
    name::String
end

speak(::Dog) = "woof"
speak(::Cat) = "meow"
describe(a::Animal) = "$(a.name) says $(speak(a))"   # shared, on the abstract type
```

Parametric types

```
Vector{Int}              # concrete
Vector                   # UnionAll: Vector{T} for all T
Dict{String, Int}
Array{Float64, 3}        # element type, dimensionality
Tuple{Int, String}
```

Writing your own:

```
struct Box{T}
    value::T
end
Box(1)          # Box{Int64}
Box("a")        # Box{String}

struct Pair2{K, V}
    key::K
    value::V
end
```

```
struct Bounded{T <: Real}     # constrained parameter
    lo::T
    hi::T
end
```

The `where` syntax spells out what `{}` implies:

```
Vector{T} where T                # same as Vector
Vector{T} where T <: Real
f(x::Vector{T}) where {T <: Real} = zero(T)   # T is usable in the body
```

Invariance — the one that costs everyone an hour

```
Vector{Int} <: Vector{Real}      # FALSE
Vector{Int} <: Vector{<:Real}    # true
Vector{Int} <: AbstractVector    # true
```

Parametric types are invariant: `Box{Int}` and `Box{Real}` are unrelated, even though `Int <: Real`. This is not an oversight — a `Vector{Real}` is a genuinely different thing in memory (boxed pointers) from a `Vector{Int}` (packed 64-bit words), so allowing the substitution would be unsound.

Practical rule for writing signatures:

```
f(v::Vector{Real})           # accepts ONLY Vector{Real}. Almost never what you want.
f(v::Vector{<:Real})         # accepts Vector{Int}, Vector{Float64}, ...
f(v::AbstractVector{<:Real}) # ALSO accepts ranges, views, StaticArrays, GPU arrays. Best.
f(v)                         # accepts everything. Fine, and just as fast.
```

Default to `AbstractVector`/`AbstractArray`/`AbstractString` in public signatures. Using `Vector` in a signature quietly excludes every view and range in the ecosystem.

Unions

```
Union{Int, String}
Union{T, Nothing}         # the idiomatic "optional"
Union{}                   # the bottom type; no value has it. Return type of throw().
```

Small unions (a handful of concrete types) are heavily optimised — the compiler splits the branches and unboxes. `Union{Int, Nothing}` in a hot loop costs essentially nothing. Big or open-ended unions are slow. So `Union{T, Nothing}` returns are idiomatic and fine; `Union{Int, Float64, String, Missing, Nothing}` in a struct field is not.

Type unions vs abstract types

```
const Numbers = Union{Int, Float64}     # closed set, you enumerate it
abstract type Shape end                 # open set, anyone can extend it
```

Use an abstract type when third parties should be able to add cases. Use a `Union` when the set is genuinely closed (and you want exhaustiveness).

Value types

Occasionally you want to dispatch on a *value* rather than a type. `Val` lifts a value into the type domain.

```
f(::Val{:fast}) = "fast path"
f(::Val{:safe}) = "safe path"
f(Val(:fast))

# common in numerical code for compile-time dimensions
g(x, ::Val{N}) where {N} = ntuple(i -> x^i, Val(N))
```

Don't reach for this often. It forces recompilation per value and is easy to misuse; it's for when the value genuinely changes the generated code.

Type assertions, conversion, promotion

```
x = 3::Int                   # assertion — throws TypeError if wrong, no conversion
convert(Float64, 3)          # 3.0 — explicit, extensible
Float64(3)                   # constructor
```

```
promote(1, 2.0)              # (1.0, 2.0) — find a common type
promote_type(Int, Float64)   # Float64
oftype(x, 3)                 # convert 3 to the type of x
```

Extending conversion and promotion for your own numeric types:

```
Base.convert(::Type{Money}, x::Real) = Money(round(Int, 100x))
Base.promote_rule(::Type{Money}, ::Type{<:Real}) = Money
```

Note the difference between `x::Int` in a *signature* (dispatch), in an *assignment* (assertion plus implicit `convert`), and in a *struct field* (storage type plus implicit `convert` on construction).

Introspection

```
typeof(x)
eltype(v)                # element type of a collection
isa(x, T); x isa T
isconcretetype(T); isabstracttype(T)
fieldnames(MyStruct)
fieldtypes(MyStruct)
sizeof(Int)              # 8
typemin(Int), typemax(Int), eps(Float64)
zero(T), one(T)          # generic identity elements — use these in generic code
```

`zero(T)` and `one(T)` matter more than they look: writing `s = 0` in a generic accumulator makes the function type-unstable for `Float64` input. Write `s = zero(eltype(v))`.

Where types matter for speed

The rule is: **fields must be concretely typed, or parametric.**

```
struct Fast{T <: Real}
    x::T          # concrete once instantiated — unboxed, inlined
end

struct AlsoFast
    x::Float64
end

struct Slow
    x             # implicit ::Any — boxed pointer, kills inference downstream
end

struct AlsoSlow
    x::Real       # abstract — same problem
end
```

`Slow` and `AlsoSlow` will run, produce correct answers, and be an order of magnitude slower. Nothing warns you. `@code_warntype` is how you find out (section 16).

05. Multiple dispatch

This is the organising principle of the language, the thing that has no close equivalent in Go, Python, Java or Elixir, and the reason the ecosystem composes as well as it does.

The model

A **function** is a name. A **method** is one implementation of it for a particular tuple of argument types. Calling a function selects the most specific applicable method, based on the runtime types of **all positional arguments**.

```
combine(a::Int, b::Int)       = a + b
combine(a::String, b::String) = a * b
combine(a::Number, b::Number) = float(a) + float(b)
combine(a, b)                 = error("cannot combine $(typeof(a)) and $(typeof(b))")

combine(1, 2)        # 3       — the Int,Int method
```

```julia
combine(1, 2.0)      # 3.0     — falls back to Number,Number
combine("a", "b")    # "ab"
```

```julia
methods(combine)         # every method, with file and line
@which combine(1, 2)     # which one this call selects
@edit combine(1, 2)      # open that method in your editor
methodswith(Money)       # every method in scope taking a Money
```

How it differs from what you already know

vs OO single dispatch (Java, Python, Ruby). `a.combine(b)` dispatches on `a` only; `b` is just an argument. Adding an operation across a type hierarchy means editing every class, which is why the visitor pattern exists. In Julia, dispatch is symmetric across all arguments, so binary operations on mixed types are natural and the visitor pattern doesn't need to exist.

vs Go interfaces. Go dispatches on one receiver satisfying a declared interface. Julia has no interface declarations at all — a type "satisfies" a protocol by having the right methods defined, and there is nothing to declare or check. Looser, and unverified.

vs Elixir function clauses. Superficially similar — several `def` clauses for one name — but the differences are load-bearing:

	Elixir	Julia
Matches on	values, shapes, guards	types only
Order	top to bottom, first match wins	order-independent, most specific wins
Ambiguity	impossible (order decides)	possible, and an error
Extending from outside the module	no	yes, and that's the point

That last row is the whole game. Any package can add methods to any function for its own types. `Base.+` gets extended by every numeric package; `Base.iterate` by every container package. This is why `DifferentialEquations` works on `Measurements` numbers on GPU arrays without any of those three packages knowing about the others.

Specificity and ambiguity

The compiler picks the most specific method. `Int` beats `Integer` beats `Real` beats `Any`. When two methods are applicable and neither is more specific, you get an error:

```julia
f(x::Int, y::Any) = 1
f(x::Any, y::Int) = 2
f(1, 1)     # MethodError: f(::Int64, ::Int64) is ambiguous
```

The fix is to define the intersection explicitly:

```julia
f(x::Int, y::Int) = 3
```

`Test.detect_ambiguities` and Aqua.jl find these in a package before your users do.

Dispatch idioms

Fallback plus specialisation. Define the generic version, then specialise for types where you can do better.

```julia
distance(a, b) = sqrt(sum(abs2, a .- b))            # works for anything
distance(a::SparseVector, b::SparseVector) = ...    # exploits sparsity
```

Dispatching on the type itself (a "trait" argument or a constructor-like function). Note `::Type{T}`:

```julia
parse_as(::Type{Int}, s) = parse(Int, s)
parse_as(::Type{Date}, s) = Date(s)
parse_as(Int, "42")
```

Nothing-argument for the empty case:

```julia
process(x::Nothing) = default()
process(x) = actually_process(x)
```

Traits (Holy traits) — when you want to dispatch on a property that cuts across the type hierarchy, since you only get one supertype:

```
abstract type Sortability end
struct IsSortable <: Sortability end
struct NotSortable <: Sortability end

sortability(::Type) = NotSortable()          # default
sortability(::Type{<:Real}) = IsSortable()   # opt in
sortability(::Type{<:MyThing}) = IsSortable()

mysort(v) = _mysort(sortability(eltype(v)), v)
_mysort(::IsSortable, v) = sort(v)
_mysort(::NotSortable, v) = error("not sortable")
```

This is how `Base` implements `IndexStyle`, `IteratorSize`, and so on. It's the Julia answer to "I want an interface that isn't in my inheritance chain".

Extending other people's functions

```
import Base: show, ==, +          # `import` (not `using`) to add methods unqualified
# or, without importing:
Base.show(io::IO, m::Money) = print(io, "£", m.cents / 100)
Base.:(==)(a::Money, b::Money) = a.cents == b.cents
Base.:+(a::Money, b::Money) = Money(a.cents + b.cents)
Base.hash(m::Money, h::UInt) = hash(m.cents, h)
```

trap: `using Base` does *not* let you add methods to `show` by writing `show(...) = ...` — that defines a new local function that shadows it. Either `import Base: show` or write the qualified `Base.show(...) = ...`.

Type piracy

Defining a method where neither the function nor any of the argument types belongs to you:

```
# in YourPackage, where none of Base.+, Int, String are yours:
Base.:+(a::Int, b::String) = ...     # PIRACY
```

It changes behaviour globally for anyone who loads your package, including code that never asked. It also breaks precompilation invalidation in ugly ways. The legitimate versions: define your own type and dispatch on that, or define your own function.

Aqua.jl checks for piracy in your package.

Where dispatch costs you

- **Dynamic dispatch** happens when the compiler can't infer the argument types — then method lookup is at runtime, and it's slow. This is downstream of type instability (section 16), not of dispatch itself.
- **Method explosion**: dispatching on many type combinations generates many specialisations, which is compile time and memory. Usually fine; occasionally the reason a package takes 40 seconds to load.
- **`invoke`** lets you deliberately call a less specific method: `invoke(f, Tuple{Real}, 1)`. Rare, but useful in a specialised method that wants to call the generic one.

A worked shape of a design

```
abstract type Distribution end

struct Normal{T <: Real} <: Distribution
    μ::T
    σ::T
end
struct Uniform{T <: Real} <: Distribution
    a::T
    b::T
end

# the "interface": anything that is a Distribution should implement these
```

```
mean(d::Normal)  = d.μ
mean(d::Uniform) = (d.a + d.b) / 2
var(d::Normal)   = d.σ^2
var(d::Uniform)  = (d.b - d.a)^2 / 12

# generic code written once against the abstract type
std(d::Distribution) = sqrt(var(d))
zscore(d::Distribution, x) = (x - mean(d)) / std(d)
```

Someone else's package can now add `struct Cauchy <: Distribution` with `mean` and `var`, and `zscore` works on it without either package changing. That composability is what you're buying.

06. Structs

Immutable by default

```
struct Point
    x::Float64
    y::Float64
end

p = Point(1.0, 2.0)
p.x                    # 1.0
p.x = 5.0              # ERROR: immutable struct of type Point cannot be changed
```

Prefer immutable. An immutable struct of concrete fields is stored inline (in an array, in another struct, on the stack), needs no allocation, and can be freely shared without defensive copying. `mutable struct` becomes a heap-allocated object with a pointer.

To "change" one, rebuild it:

```
q = Point(5.0, p.y)

using Accessors            # if you do this a lot
q = @set p.x = 5.0         # returns a new Point
q = @set p.nested.deep.field = 1   # works through nesting
```

Mutable, and const fields

```
mutable struct Counter
    n::Int
    const name::String     # 1.8+: this field alone is immutable
end

c = Counter(0, "hits")
c.n += 1
c.name = "x"     # ERROR
```

Mutable structs also get identity: two `Counter(0,"x")` are `!==` each other, whereas two identical immutable `Points` may be indistinguishable.

Field types matter for performance

```
struct Fast
    x::Float64             # concrete
end

struct AlsoFast{T <: Real}
    x::T                   # concrete once instantiated
end

struct Slow
    x                      # ::Any — boxed
end
```

```
struct AlsoSlow
    x::Real               # abstract — boxed
end

struct SlowContainer
    v::AbstractVector     # abstract container type — inference dies here
end

struct FastContainer{V <: AbstractVector}
    v::V                  # parametric — keeps the concrete type
end
```

The parametric form is the one to reach for whenever a field holds something whose exact type varies. It costs you nothing at the call site and everything downstream stays inferable.

Constructors

Every struct gets a default constructor taking all fields in order. Two ways to add more.

Outer constructors are just extra methods. Defaults, alternative argument forms, computed fields.

```
struct Rect
    w::Float64
    h::Float64
end

Rect(s) = Rect(s, s)                  # square
Rect(; w, h) = Rect(w, h)             # keyword form
```

Inner constructors live inside the struct block, have access to `new`, and are the only place you can enforce invariants — because defining one *replaces* the default constructor, so no path exists around it.

```
struct Celsius
    deg::Float64

    function Celsius(d)
        d < -273.15 && throw(ArgumentError("below absolute zero"))
        new(d)
    end
end
```

`new` is special: it's the raw allocator, only callable inside an inner constructor. For parametric types, `new{T}(...)` when you need to name the parameter:

```
struct Ordered{T}
    lo::T
    hi::T
    function Ordered(lo::T, hi::T) where {T}
        lo > hi && ((lo, hi) = (hi, lo))
        new{T}(lo, hi)
    end
end
```

`new` also lets you build partially-initialised mutable structs (for self-referential data like linked lists) by passing fewer arguments than there are fields.

Keyword structs

```
Base.@kwdef struct Config
    host::String = "localhost"
    port::Int    = 8080
    tls::Bool    = false
    retries::Int                      # no default — required keyword
end
```

```
Config(retries = 3)
Config(host = "example.com", port = 443, tls = true, retries = 3)
```

This is the closest thing to a config object with defaults, and it composes with `@set` for overriding.

Composition instead of inheritance

You cannot subtype a concrete struct. To reuse data, embed and forward:

```
struct Base_
    id::Int
    name::String
end

struct Employee
    base::Base_
    salary::Float64
end

# forward the accessors you care about
id(e::Employee) = e.base.id
name(e::Employee) = e.base.name
```

If you're forwarding many methods, `Lazy.jl`'s `@forward` or `ReusePatterns.jl` automate it. Usually the honest answer is that you wanted an abstract type plus duck-typed accessors, not inheritance.

Custom display

Two methods, two contexts:

```
# compact, used inside arrays and interpolation
Base.show(io::IO, m::Money) = print(io, "£", m.cents / 100)

# verbose, used when the object is the REPL result
function Base.show(io::IO, ::MIME"text/plain", m::Money)
    println(io, "Money")
    println(io, "  cents: ", m.cents)
end
```

Respect the `io` argument rather than printing to stdout, and honour `get(io, :compact, false)` if you want to be polite inside arrays.

Equality and hashing

The default `==` for structs is field-wise for immutable structs and identity for mutable ones. If you define `==`, define `hash` to match, or `Dict` and `Set` will behave incorrectly:

```
Base.:(==)(a::Money, b::Money) = a.cents == b.cents
Base.hash(m::Money, h::UInt) = hash(m.cents, hash(:Money, h))
Base.isequal(a::Money, b::Money) = isequal(a.cents, b.cents)
```

Callable structs (functors)

Any object can be made callable by defining a method on its type. This is how closures, neural network layers, and optimisation objectives are usually modelled.

```
struct Polynomial{T}
    coeffs::Vector{T}
end

(p::Polynomial)(x) = evalpoly(x, p.coeffs)

p = Polynomial([1, 2, 3])
p(2.0)      # 17.0
map(p, 1:5)
```

Enums

```
@enum Suit hearts diamonds clubs spades
@enum Status::UInt8 pending = 1 active = 2 done = 3

s = hearts
s isa Suit
Int(spades)          # 3
instances(Suit)
Suit(0)              # hearts
```

`@enum` values are their own type, so you can dispatch on them: `handle(::Val{pending})` or more simply `handle(s::Status)` with an if-chain. For richer sum types, define an abstract type with singleton structs, or use `SumTypes.jl`.

Singletons

```
struct Verbose end
struct Quiet end

log(::Verbose, msg) = println(msg)
log(::Quiet, msg) = nothing
```

Zero-size, zero-cost, dispatch-only. Used constantly for options, traits, and flags — this is how you avoid passing boolean flags that force runtime branches.

07. Functions

Definition forms

```
function add(x, y)
    return x + y          # `return` optional; last expression is the value
end

add(x, y) = x + y         # assignment form

const add2 = (x, y) -> x + y     # anonymous, bound to a name (slightly slower to dispatch)

function noargs()::Nothing       # return type annotation: CONVERTS the result
    nothing
end
```

A return type annotation is not a check, it's a `convert`. `f(x)::Int = x` will happily turn `2.0` into `2` and throw on `2.5`.

Arguments

```
f(a, b = 2; c, d = 4, kwargs...) = (a, b, c, d, kwargs)
#  ^  ^-- optional positional    ^-- required keyword (no default)
#         everything after ; is keyword-only
```

- Positional arguments participate in dispatch. **Keyword arguments do not.**
- Optional positional arguments generate extra methods behind the scenes (`f(a)` calls `f(a, 2)`).
- Keyword arguments are matched by name, order-independent, and can be splatted from a NamedTuple or Dict with `String` or `Symbol` keys.

```
opts = (c = 3, d = 5)
f(1; opts...)

# passing keywords through
wrapper(args...; kwargs...) = f(args...; kwargs...)
```

Splatting and slurping

```
f(args...) = length(args)      # slurp into a Tuple
t = (1, 2, 3)
```

```
f(t...)                         # splat
maximum([1,2,3]...)             # works, but prefer maximum([1,2,3])
```

Splatting a *long* runtime-length collection is slow (it forces a heap-allocated tuple and defeats inference). Splatting short, statically-known tuples is free.

Anonymous functions and closures

```
map(x -> x^2, 1:3)
filter(x -> x > 2, v)

function counter()
    n = 0
    () -> (n += 1)        # closure over n; n is boxed because it's reassigned
end
```

trap: closures that reassign captured variables box them, which is a performance cliff. If it matters, use a mutable struct or a `Ref` explicitly, or restructure so nothing is reassigned.

Multi-line anonymous:

```
f = function (x, y)
    x + y
end
```

`do` blocks

`do` is sugar that turns a trailing block into the *first* argument. Anywhere a function takes a function first, `do` reads better.

```
open("file.txt", "r") do io
    read(io, String)
end
# equivalent to: open(io -> read(io, String), "file.txt", "r")
```

This is Julia's resource-management idiom — the file closes even if the block throws. Same pattern for locks, temp dirs, connections:

```
lock(mylock) do
    shared_state += 1
end

mktempdir() do dir
    ...
end

Channel{Int}(10) do ch
    for i in 1:100; put!(ch, i); end
end
```

And for anything with a function argument:

```
map(1:3) do x
    y = expensive(x)
    y^2 + 1
end

sum(v) do x
    x^2
end
```

Composition and piping

```
(sqrt ∘ abs)(-4)          # 2.0;  ∘ is \circ<tab>
-4 |> abs |> sqrt         # 2.0
[-4, -9] .|> abs .|> sqrt # broadcast the pipe
```

```
filter(!isempty, lines)   # ! composes with a predicate to negate it
findfirst(==(3), v)       # ==(3) is a curried predicate; also <(5), in(set), isequal(x)
```

The curried-comparison forms (`==(x)`, `<(x)`, `in(c)`, `isequal(x)`) are used constantly with `findfirst`, `filter`, `count`, `any`, `all`.

Multiple return values and destructuring

```
minmax(3, 1)              # (1, 3) — just a tuple
lo, hi = minmax(3, 1)
a, b... = [1,2,3]         # a = 1, b = [2,3]
(; host, port) = config   # 1.7+: destructure by property name from a NamedTuple or struct

function stats(v)
    (mean = sum(v)/length(v), n = length(v))   # NamedTuple return: self-documenting
end
s = stats(v); s.mean
```

Returning a NamedTuple is the idiomatic alternative to out-parameters or a bespoke result struct, and it's type-stable and free.

Mutating functions

Convention: `!` suffix, mutated argument first.

```
function normalize!(v)
    v ./= sum(v)
    return v            # return the mutated argument, for chaining
end
```

Many `Base` functions have both forms, and the in-place one usually takes the destination as the *first* argument even when the non-mutating one takes it last:

```
map!(f, dest, src)
copyto!(dest, src)
mul!(C, A, B)          # C .= A * B without allocating
sort!(v); push!(v, x); append!(v, w); empty!(v)
```

Generic functions and `where`

```
f(x::T, y::T) where {T} = ...              # both arguments the same type
f(v::AbstractVector{T}) where {T <: Real} = zero(T)
f(::Type{T}) where {T} = T                 # dispatch on a type argument
```

Use `T` in the body when you need to construct something of the right type — `zero(T)`, `one(T)`, `similar(v)`, `T[]`. Hard-coding `0` or `Float64[]` in a generic function is the usual cause of "works for Float64, breaks for Complex or Measurement or Dual".

Recursion and iteration helpers

```
foldl(+, 1:5); foldr(-, 1:5)
reduce(+, v; init = 0)
mapreduce(abs2, +, v)          # fused map-then-reduce, no intermediate
accumulate(+, v); cumsum(v)
```

Julia has no tail-call optimisation. Deep recursion overflows the stack; convert to iteration or increase the stack with a `Task`:

```
t = Task(deep_recursive_thing)   # tasks get their own, growable stack
```

Documenting

```
"""
    solve(A, b) -> x

Solve `A x = b`.

# Arguments
```

```
- `A::AbstractMatrix`: the system matrix.
- `b::AbstractVector`: the right-hand side.

# Examples
```jldoctest
julia> solve([1 0; 0 1], [1, 2])
2-element Vector{Float64}:
 1.0
 2.0
```
"""
function solve(A, b) ... end
```

The `jldoctest` blocks are executed as tests by Documenter.jl, so your examples can't rot.

Function-like odds and ends

```
applicable(f, 1, 2)       # would this call find a method?
hasmethod(f, Tuple{Int})
Base.@nospecialize x      # tell the compiler NOT to specialise on this argument
@inline f(x) = ...        # hints; the compiler mostly knows better
@noinline
Base.@propagate_inbounds  # forward @inbounds through a wrapper
```

08. Arrays and broadcasting

Construction

```
[1, 2, 3]                       # Vector{Int64} (3-element, 1-D)
[1 2 3]                         # 1x3 Matrix — spaces make columns
[1 2; 3 4]                      # 2x2 Matrix — semicolons make rows
[1, 2, 3]'                      # adjoint: 1x3 row (lazy wrapper, not a copy)

zeros(3); zeros(Int, 2, 2); ones(2, 3); fill(7, 2, 2)
trues(3); falses(3)
Vector{Float64}(undef, 100)     # uninitialised — fastest, contains garbage
similar(A); similar(A, Float64); similar(A, 3, 3)
rand(3, 3); randn(100); rand(1:6, 10)
collect(1:5); range(0, 1; length = 11); range(0, 1; step = 0.1)
reshape(1:12, 3, 4)
repeat([1,2], 3); repeat([1 2], 2, 2)
Int[]                           # typed empty. NOT []
```

`[]` alone is `Vector{Any}` — a common accidental performance sink.

Indexing

```
v[1]; v[end]; v[end-1]; v[2:end]; v[[1,3]]; v[v .> 2]     # boolean mask
A[i, j]; A[:, j]; A[i, :]; A[1:2, 2:3]; A[:]              # A[:] flattens (column-major)
A[CartesianIndex(1,2)]
first(v); last(v); v[begin]                                # `begin` works as an index too
```

Assignment mirrors it, with broadcast on the right-hand side:

```
A[:, 1] .= 0            # fill a column
A[:, 1] = zeros(3)      # same, but allocates the RHS
v[v .< 0] .= 0          # clamp negatives
```

trap: `A[:, 1] = 0` (scalar, no dot) is an error. Use `.=` when assigning a scalar into a slice.

Views vs copies

```
B = A[:, 2:3]           # COPY
B = @view A[:, 2:3]     # view (SubArray), shares memory
B = view(A, :, 2:3)     # function form
```

```
@views function f(A)    # every slicing operation in this function becomes a view
    s = 0.0
    for j in axes(A, 2)
        s += sum(A[:, j])
    end
    s
end
```

`@views` on a function or block is usually the right move in numerical code. The exception: if you're going to keep the result around, a view holds the whole parent array alive.

Iteration

```
for x in v end                    # elements
for i in eachindex(v) end         # correct indices for ANY array type
for (i, x) in pairs(v) end        # index => element
for (i, x) in enumerate(v) end    # 1,2,3... regardless of actual indices
for c in eachcol(A) end
for r in eachrow(A) end
for I in CartesianIndices(A) end  # A[I] works, I[1], I[2] are the components
for (a, b) in zip(v, w) end
axes(A, 1); size(A); size(A, 2); length(A); ndims(A)
```

Prefer `eachindex`/`axes` over `1:length(v)`. It's correct for views, offset arrays, and sparse arrays, and it enables bounds-check elision.

Growing and mutating

```
push!(v, x); pop!(v)
pushfirst!(v, x); popfirst!(v)
append!(v, w); prepend!(v, w)
insert!(v, 2, x); deleteat!(v, 2); splice!(v, 2)
resize!(v, 10); empty!(v)
sizehint!(v, 10_000)          # preallocate capacity before a push! loop
```

Only `Vector` grows; matrices are fixed-size (use `vcat`/`hcat` or build a vector of rows and `reduce(hcat, ...)`).

Combining

```
vcat(a, b); hcat(a, b); cat(a, b; dims = 3)
[a; b]        # vcat
[a b]         # hcat
[a b; c d]    # block matrix
reduce(vcat, list_of_vectors)     # much faster than repeated vcat in a loop
stack(list_of_vectors)            # 1.9+: list of vectors -> matrix
```

Broadcasting: the dot

Any function becomes elementwise with a dot, and **adjacent dotted operations fuse into a single loop with no temporary arrays**.

```
x = [1.0, 2.0, 3.0]
sqrt.(x)
x .+ 1
x .* y
f.(x, y)

z = 2 .* x .+ 1 .- sqrt.(x)     # ONE pass, ONE allocation (the result)
@. z = 2x + 1 - sqrt(x)         # dots everything; writes into z, ZERO allocation
```

The fusion is the point. In NumPy, `2*x + 1 - np.sqrt(x)` builds three temporaries; in Julia it builds none.

Shape rules: dimensions of length 1 are stretched.

```
[1,2,3] .+ [10 20]        # 3x2 matrix — column plus row is an outer operation
A .- mean(A; dims = 1)    # centre each column
```

Protect an argument you *don't* want iterated with `Ref` or a 1-tuple:

```
in.([1,2,3], Ref([1,5]))          # elementwise membership in ONE set -> [true,false,false]
f.(x, Ref(some_big_config))
```

In-place broadcast:

```
x .= 0.0
y .= f.(x)          # writes into existing y
x .+= 1             # in place
x = x .+ 1          # allocates a new array — not the same thing
```

Custom types opt in via `Base.broadcastable`; scalars-by-default via `Base.broadcastable(x::MyType) = Ref(x)`.

Comprehensions and generators

```
[x^2 for x in 1:10]
[x^2 for x in 1:10 if isodd(x)]
[i*j for i in 1:3, j in 1:4]              # 3x4 Matrix — multiple `for` make dimensions
[(i,j) for i in 1:3 for j in 1:i]         # nested (dependent), flattens to a Vector
Dict(k => f(k) for k in keys)
Set(x % 3 for x in 1:10)
Float64[x for x in 1:3]                   # force the element type

(x^2 for x in 1:10)                       # generator: lazy, allocates nothing
sum(x^2 for x in 1:10)                    # consumed without materialising
```

Use a generator whenever the result is immediately reduced.

The functional toolkit

```
map(f, v); map(f, v, w); map!(f, dest, v)
filter(f, v); filter!(f, v)
reduce(+, v); reduce(vcat, vs); foldl; foldr
mapreduce(abs2, +, v)
sum, prod, maximum, minimum, extrema, count
sum(abs2, v)                        # function-first forms avoid a temporary
any(iseven, v); all(>(0), v)
findfirst(iseven, v)                # index or nothing
findlast, findall, findmax, findmin
argmax(v); argmax(f, v)             # 1.7+: argmax over f's values, returns the input
sort(v; by, lt, rev, alg); sort!; sortperm(v); partialsort(v, 1:3)
unique(v); unique(f, v); allunique
union, intersect, setdiff, symdiff
zip, enumerate, pairs
Iterators.take(it, 5); drop; partition(v, 3); flatten; product; cycle; countfrom
Iterators.filter, Iterators.map          # lazy versions
```

Dicts and sets

```
d = Dict("a" => 1, "b" => 2)
d = Dict{String,Int}()
d["c"] = 3
d["a"]                      # KeyError if missing
get(d, "z", 0)              # default
get!(d, "z", 0)             # default AND insert
haskey(d, "a"); delete!(d, "a"); pop!(d, "a", default)
keys(d); values(d); collect(pairs(d))
for (k, v) in d ... end     # order is unspecified and unstable between runs
merge(d1, d2); mergewith(+, d1, d2)

s = Set([1,2,3]); push!(s, 4); 3 in s; union(s1, s2)
```

For insertion-ordered dicts use `OrderedDict` from DataStructures.jl. `Base` also has `IdDict` (identity keys) and `WeakKeyDict`.

Missing pieces worth knowing exist

- `SparseArrays` (stdlib): `sparse(I, J, V)`, `spzeros`, and the whole LinearAlgebra stack dispatches on them.
- `StaticArrays.jl`: `SVector{3,Float64}` — stack-allocated fixed-size arrays, 10x faster for small vectors. Use for 2-D/3-D geometry.
- `OffsetArrays.jl`: arbitrary index bases, which is why you write `eachindex` instead of `1:length`.
- `StructArrays.jl`: array-of-structs interface over a struct-of-arrays layout.

09. Strings and symbols

Literals and interpolation

```
"hello"
'a'                       # Char — SINGLE quotes are a character, not a string
"a" * "b"                 # concatenation is *, not +
"ab"^3                    # repetition
"$name is $(age + 1)"     # interpolation, any expression in $()
"\$literal dollar"

"""
Triple-quoted. Leading whitespace common to all lines is stripped,
so you can indent this inside a function without polluting the string.
"""

raw"C:\no\escapes\here"   # no escape processing, no interpolation
b"bytes"                  # a byte array literal (CodeUnits)
```

The byte-index trap

Strings are UTF-8, immutable, and indexed by **byte offset**.

```
s = "héllo"
length(s)        # 5   — characters
ncodeunits(s)    # 6   — bytes
s[1]             # 'h'
s[2]             # 'é'
s[3]             # StringIndexError: index 3 is mid-character
s[nextind(s, 2)] # 'l'
s[end]           # 'o'  — `end` is a valid index by construction

firstindex(s), lastindex(s)
eachindex(s)     # the valid byte indices
prevind(s, i), nextind(s, i), thisind(s, i)
```

Safe patterns: iterate with `for c in s`, index with results from `findfirst`/`eachindex`, or `collect(s)` into a `Vector{Char}` if you truly need positional access. `SubString` is the free-slicing type — `s[2:4]` returns a `String` copy, `@view` doesn't work on strings, but `SubString(s, 2, 4)` and the outputs of `split` are views.

The everyday operations

```
length, isempty, first, last
occursin("ell", s); occursin(r"\d+", s)
startswith(s, "he"); endswith(s, "o")
findfirst("ll", s)                    # a range of byte indices, or nothing
replace(s, "l" => "L"); replace(s, "a" => "b", "c" => "d")
split("a,b,,c", ","); split(s)     # bare split(): on whitespace, drops empties
split("a,b", ","; keepempty = false)
join(["a","b","c"], ", ", " and ")
strip(s); lstrip; rstrip; strip(s, ['"'])
uppercase, lowercase, titlecase, uppercasefirst
lpad("7", 3, "0"); rpad
repeat("ab", 3)
chop(s); chomp(s)                     # remove last char / trailing newline
```

```
reverse(s)
string(1, "x", 2.0)                  # generic stringify-and-concat
```

Building strings efficiently

Repeated `s = s * x` is quadratic. Use an `IOBuffer`:

```
io = IOBuffer()
for x in items
    print(io, x, ",")
end
result = String(take!(io))

# or
sprint(io -> foreach(x -> print(io, x), items))
```

Formatting

```
using Printf
@printf("%.3f %5d %s\n", π, 42, "ok")
s = @sprintf("%08.3f", π)

# runtime format strings (1.6+)
fmt = Printf.Format("%.2f")
Printf.format(fmt, 1.234)

round(π; digits = 3)            # often all you need
string(1234; base = 16)         # "4d2"
bitstring(1)
```

For anything table-shaped, `PrettyTables.jl` beats hand-rolled padding.

Regex

```
r"(\d+)-(\d+)"                    # Regex literal
r"case"i                          # flags: i, m, s, x

m = match(r"(\d+)-(\d+)", "ab 12-34 cd")
m === nothing                     # no match
m.match                           # "12-34"
m.captures                        # ["12", "34"]  (Union{String,Nothing} elements)
m[1]                              # "12"
m.offset                          # byte index of the match

occursin(r"^\d+$", s)
replace(s, r"\s+" => " ")
replace(s, r"(\w+)@(\w+)" => s"\2 at \1")     # s"" for substitution templates
for m in eachmatch(r"\d+", s); m.match; end
split(s, r",\s*")

# named captures
m = match(r"(?<year>\d{4})-(?<month>\d{2})", "2026-08")
m[:year]
```

Parsing and conversion

```
parse(Int, "42"); parse(Float64, "1.5"); parse(Int, "ff"; base = 16)
tryparse(Int, "x")           # nothing rather than throwing
string(42); string(3.14)
Char(65); Int('A')
codeunits(s)                 # byte view
transcode(UInt16, s)
```

Symbols

A `Symbol` is an interned, immutable identifier. Same role as an Elixir atom, but used more narrowly: for property names, keyword names, option flags, and as the leaves of expression trees in metaprogramming.

```
:foo
Symbol("with spaces")
Symbol("prefix_", 42)           # Symbol("prefix_42")
String(:foo); Symbol("foo")     # round trip

nt = (host = "x", port = 80)
nt[:host]
getproperty(obj, :field)        # obj.field, computed
setproperty!(obj, :field, v)
propertynames(obj)
fieldnames(MyStruct)            # Tuple of Symbols
```

Symbol comparison is pointer comparison, so `:a == :b` is O(1) regardless of length — that's why they're used as dict keys for fixed vocabularies and as option flags:

```
sort(v; alg = :quicksort)
plot(x, y; linestyle = :dash)
```

They are *not* a general-purpose string type. Don't build them from user input in a loop — the intern table is never garbage collected.

Chars

```
c = 'a'
c + 1                       # 'b'
'a' <= c <= 'z'
isletter(c), isdigit(c), isspace(c), isuppercase(c), ispunct(c)
uppercase('a')
codepoint(c)
```

10. nothing, missing, NaN

Julia has three different "absent value" concepts and they are not interchangeable. Coming from a language with one `null` (or with `nil`/`None`), this is the distinction to internalise early.

Value	Type	Means	Arithmetic	Test
`nothing`	`Nothing`	no value; void return	`1 + nothing` is a **MethodError**	`isnothing(x)`, `x === nothing`
`missing`	`Missing`	value exists but is unknown (statistics)	`1 + missing` is `missing` — propagates	`ismissing(x)`
`NaN`	`Float64`	undefined float result	propagates within floats	`isnan(x)`
`undef`	`UndefInitializer`	"don't initialise this memory"	—	only in constructors

nothing

The absence of a value. Returned by functions that found nothing, and by functions called for their side effects.

```
findfirst(==(99), [1,2,3])      # nothing
match(r"z", "abc")              # nothing
get(dict, :missing_key, nothing)
println("x")                    # returns nothing
```

Working with it:

```
x = findfirst(==(3), v)
if isnothing(x)
    ...
end
x === nothing                # equivalent, and the fastest check

something(a, b, 0)           # first argument that isn't nothing; throws if all are
@something a b 0             # short-circuiting version (1.7+)

# the idiomatic optional-return signature
function lookup(k)::Union{Int, Nothing}
    haskey(d, k) ? d[k] : nothing
end
```

`Union{T, Nothing}` is the closest thing to `Option`/`Maybe`, and the compiler optimises small unions well, so there's no performance argument against it. What Julia does *not* have is the `{:ok, value} | {:error, reason}` convention from Elixir — the ecosystem either returns `nothing` or throws. If you want tagged results, `ResultTypes.jl` or `Try.jl` exist but are not idiomatic.

trap: `nothing` displays as nothing at the REPL. A function that silently returns `nothing` looks like it returned successfully. `@show f(x)` or `typeof(f(x))` when confused.

missing

Statistical missingness, modelled on SQL `NULL` and R's `NA`. It **propagates** through almost every operation.

```
1 + missing            # missing
missing == 1           # missing  (NOT false!)
missing == missing     # missing
isequal(missing, missing)   # true  — this is why isequal exists
missing > 1            # missing

if missing             # TypeError: non-boolean used in boolean context
end
```

Three-valued logic for the short-circuiting operators:

```
true  || missing       # true
false || missing       # missing
false && missing       # false
true  && missing       # missing
```

Handling it:

```
ismissing(x)
coalesce(x, 0)                      # first non-missing
skipmissing(v)                      # lazy iterator that drops them
sum(skipmissing(v)); mean(skipmissing(v))
collect(skipmissing(v))
replace(v, missing => 0)
filter(!ismissing, v)
disallowmissing(v)                  # (Missings.jl) narrow the element type back

eltype([1, missing, 3])             # Union{Missing, Int64}
```

Column types in DataFrames are the main place you'll meet it. A column that might have gaps is `Vector{Union{Missing,Float64}}`, and every aggregation you run against it needs `skipmissing` or it returns `missing`.

Use `missing` for data. Use `nothing` for control flow. Mixing them up produces code that either propagates when it should error or errors when it should propagate.

NaN

An IEEE-754 float, not a separate type.

```
0/0                    # NaN
NaN == NaN             # false      <- the classic
NaN === NaN            # true       (identical bits)
isequal(NaN, NaN)      # true
isnan(NaN)             # true
sort([3, NaN, 1])      # NaN sorts last, because sort uses isless
maximum([1, NaN])      # NaN — propagates
maximum(filter(!isnan, v))
NaN in [NaN]           # true (uses isequal)
```

`Inf` is separate and better behaved: `1/0 == Inf`, `isinf`, `isfinite`. `isfinite(x)` is the check that excludes both `NaN` and `Inf`.

Choosing, in practice

```
# a config value that may be unset
port::Union{Int, Nothing}

# a sensor reading that failed to arrive
temperature::Union{Float64, Missing}

# a computation that has no defined answer
result = log(-1.0)     # DomainError, actually — Julia is strict here
result = log(complex(-1.0))   # if you want the complex branch
```

Note that Julia throws `DomainError` for `sqrt(-1.0)` and `log(-1.0)` rather than returning `NaN` — it will not silently promote to complex. Pass a complex input if that's what you want.

Related: undefined and uninitialised

```
v = Vector{Float64}(undef, 3)   # garbage values, fastest allocation
v = Vector{String}(undef, 3)    # #undef references — accessing throws UndefRefError
isassigned(v, 1)

x                               # UndefVarError if never assigned
@isdefined x
```

`undef` is not a value you can store; it's a marker telling the constructor to skip initialisation. For arrays of pointers (`String`, `Any`, mutable structs), the slots really are undefined and reading them throws.

11. Errors and exceptions

Throwing

```
throw(ArgumentError("port must be positive"))
error("something went wrong")                  # shorthand for ErrorException
@assert x > 0 "x must be positive"     # can in principle be compiled out;
                                       # don't rely on it for validation
```

Idiomatic guard style, using short-circuit:

```
x > 0 || throw(ArgumentError("x must be positive"))
isfile(path) || throw(SystemError("no such file: $path"))
```

The built-in exception types

Use the specific one; it's how callers discriminate.

Type	For
`ArgumentError`	argument values that don't make sense
`BoundsError`	index out of range
`DomainError`	mathematically undefined input (`sqrt(-1.0)`)
`DimensionMismatch`	array shapes that don't line up
`KeyError`	missing dict key
`MethodError`	no method for these argument types
`TypeError`	wrong type where a specific one was required
`InexactError`	lossy conversion (`Int(2.5)`)
`OverflowError`	checked arithmetic overflow
`EOFError`, `SystemError`, `IOError`	I/O
`ErrorException`	generic; what `error()` throws
`InterruptException`	Ctrl-C
`TaskFailedException`	wraps whatever a `Task` threw

Catching

```
try
    risky()
catch e
    if e isa BoundsError
        handle_bounds(e)
    elseif e isa ArgumentError
        @warn "bad argument" exception = (e, catch_backtrace())
    else
        rethrow()          # don't swallow what you didn't plan for
    end
finally
    cleanup()
end
```

Notes:

- `catch e` binds the exception; `catch` alone is legal and discards it.
- `rethrow()` preserves the original backtrace. `throw(e)` inside a catch block resets it — use `rethrow`.
- `catch_backtrace()` gets the backtrace for logging.
- `try`/`catch` is cheap when nothing throws, expensive when it does. Don't use exceptions for control flow in hot loops.
- The value of a `try` block is its last expression, so this works:

```
value = try
    parse(Int, s)
catch
    0
end
```

but `tryparse(Int, s)` is better — where a non-throwing variant exists, prefer it (`tryparse`, `get`, `findfirst`, `isfile`).

Custom exceptions

```
struct ConfigError <: Exception
    key::String
    msg::String
end

Base.showerror(io::IO, e::ConfigError) =
```

```
    print(io, "ConfigError in key '", e.key, "': ", e.msg)

throw(ConfigError("port", "must be an integer"))
```

Subtype `Exception` directly. Define `showerror` — the default display for a custom exception is unhelpfully raw.

For a family of related errors, define an abstract exception type so callers can catch the whole family:

```
abstract type MyPkgError <: Exception end
struct ParseFailure <: MyPkgError; ...; end
struct ValidationFailure <: MyPkgError; ...; end

try ... catch e; e isa MyPkgError && recover(); end
```

Cleanup patterns

`finally` works, but the `do`-block form is more idiomatic and harder to get wrong:

```
open(path, "w") do io          # closed on normal exit AND on throw
    write(io, data)
end

lock(l) do
    ...
end

mktemp() do path, io
    ...
end
```

Write your own resource wrapper the same way:

```
function with_connection(f, url)
    conn = connect(url)
    try
        f(conn)
    finally
        close(conn)
    end
end

with_connection("db://...") do conn
    query(conn, "select 1")
end
```

Errors in tasks — the trap coming from the BEAM

There are no supervisors, no linked processes, no `:trap_exit`, and nothing restarts anything. An exception in a `Task` is **stored on the task** and rethrown when someone `wait`s or `fetch`es it. If nobody does, it disappears silently.

```
t = @async error("boom")
sleep(1)                # nothing happens, no output
fetch(t)                # NOW it throws TaskFailedException, wrapping the original

# unwrap
try
    fetch(t)
catch e
    e isa TaskFailedException && rethrow(e.task.exception)
end
```

So: always `fetch` or `wait` every task you spawn, or explicitly check `istaskfailed(t)`. Fire-and-forget tasks that fail are invisible.

Warnings, logging and debugging

```
@info "starting" host port          # key-value pairs become structured fields
@warn "retrying" attempt = i
@error "failed" exception = (e, catch_backtrace())
@debug "detail"                     # off by default; JULIA_DEBUG=Main or =MyPkg

using Logging
with_logger(SimpleLogger(stderr, Logging.Debug)) do
    ...
end
```

`@info` and friends deduplicate repeated messages from the same source line, which is occasionally confusing when you're logging inside a loop — pass `maxlog = 10` or use `println` if you genuinely want every one.

Interactive debugging: `Infiltrator.jl` (`@infiltrate` drops you into a REPL at that point with locals available — low overhead, works in normal code) is the pragmatic choice. `Debugger.jl` gives a full stepping debugger but is slow because it interprets.

```
using Infiltrator
function f(x)
    y = expensive(x)
    @infiltrate           # REPL opens here; type @continue to resume
    y + 1
end
```

Retry

```
retry(f; delays = ExponentialBackOff(n = 3))()
retry(fetch_url; check = (s, e) -> e isa IOError)(url)
```

`Base.retry` returns a wrapped function; it's easy to miss that it exists.

12. Modules and packages

Modules

```
module Geometry

export area, perimeter          # names made available by `using Geometry`

using LinearAlgebra             # dependencies of this module
import Base: show               # because we will add a method to show

abstract type Shape end
include("shapes.jl")            # textual inclusion INTO this module
include("area.jl")

end # module Geometry
```

`include` is not an import system — it splices the file's text into the current module at that point. Files are an organisational convenience with no semantic meaning. One module per package is the norm; submodules are used sparingly.

using vs import

Form	Brings in	Can you add methods to those names?
`using Foo`	exported names, unqualified, plus `Foo` itself	no (must write `Foo.bar(...) = ...`)
`using Foo: bar, baz`	only those names	no
`import Foo`	only `Foo` (use `Foo.bar`)	yes, via `Foo.bar(...) = ...`
`import Foo: bar`	`bar`, unqualified	**yes** — this is the point

```
using Statistics: mean, std     # explicit, my default: obvious where names come from
import Base: show, ==, +        # required to write `show(io, x::MyType) = ...`
Base.show(io::IO, x::MyType) = ...   # equivalent, no import needed
```

trap: after `using Base`, writing `show(io, x::MyType) = ...` defines a *new* function `show` in your module that shadows `Base.show`, and your type prints with the default. No warning. This is the single most common module-level mistake.

Other forms:

```
using ..Sibling                 # relative module paths
using .SubModule                # a submodule of the current one
public foo                      # 1.11+: mark as public API without exporting into scope
Base.@ccallable                 # unrelated, but macros do appear in module headers
```

Package layout

```
MyPkg/
├── Project.toml         # identity, deps, compat
├── Manifest.toml        # fully resolved versions (a lockfile)
├── src/
│   ├── MyPkg.jl         # module MyPkg ... end — the entry point
│   └── internals.jl
├── test/
│   ├── Project.toml     # test-only dependencies
│   └── runtests.jl
├── docs/
│   ├── Project.toml
│   ├── make.jl
│   └── src/index.md
└── ext/                 # package extensions (weak dependencies)
```

`Project.toml`:

```
name = "MyPkg"
uuid = "..."
version = "0.1.0"

[deps]
DataFrames = "a93c6f00-..."

[compat]
DataFrames = "1"
julia = "1.10"

[extras]
Test = "8dfed614-..."

[targets]
test = ["Test"]
```

`Manifest.toml`: commit it for applications and reproducible analyses; **gitignore it for libraries**, because pinning your users' whole dependency graph is antisocial.

Pkg REPL (press `]`)

```
activate .              # use this directory's Project.toml
activate --temp         # throwaway environment
add DataFrames          add DataFrames@1.6      add https://github.com/u/P.jl
rm DataFrames
up                      up DataFrames           # update
st                      st --outdated
instantiate             # install exactly what Manifest says — the CI/first-clone command
resolve
dev ../MyOtherPkg       # editable local checkout (like pip install -e)
free MyOtherPkg         # undo dev
test                    # runs test/runtests.jl in a fresh process
build
precompile
gc                      # reclaim disk from unused package versions
pin / unpin
```

Programmatic equivalent:

```
using Pkg
Pkg.activate(".")
Pkg.add("DataFrames")
Pkg.instantiate()
Pkg.status()
```

Environments

Environments are just directories containing a `Project.toml`. They stack:

```
Base.LOAD_PATH      # ["@", "@v#.#", "@stdlib"]
```

`@` is the active project, `@v1.12` is your shared default environment. So packages installed into the shared environment (Revise, BenchmarkTools, OhMyREPL) are available from every project without being dependencies of any of them — which is exactly where dev tools belong.

```
julia --project=.           # activate this directory
julia --project=@myenv      # a named shared environment in ~/.julia/environments
JULIA_PROJECT=@.            # env var: activate nearest Project.toml walking upward
```

Creating a package

```
using Pkg; Pkg.generate("MyPkg")      # minimal

using PkgTemplates                     # the real one
t = Template(;
    user = "yourhandle",
    julia = v"1.10",
    plugins = [
        Git(; ssh = true),
        GitHubActions(),
        Documenter{GitHubActions}(),
        Codecov(),
        Formatter(),
    ],
)
t("MyPkg")
```

Then `] dev ~/.julia/dev/MyPkg` from wherever you want to use it.

Package extensions (weak deps)

The mechanism for "add a method for SomeOtherPackage's type, but only if the user loaded it" — replaces the old Requires.jl hack and works with precompilation.

```
[weakdeps]
Plots = "91a5bcdd-..."

[extensions]
MyPkgPlotsExt = "Plots"
```

```
# ext/MyPkgPlotsExt.jl
module MyPkgPlotsExt
using MyPkg, Plots
Plots.plot(x::MyPkg.MyType) = ...
end
```

Loading, precompilation, and Revise

- The first `using SomePkg` after installation precompiles it (and its dependency tree) into a cache including native code. Minutes once, then fast.
- Editing a package's source invalidates the cache; the next `using` recompiles.
- **Revise.jl** watches files and applies edits to the running session, so you don't restart. Put `using Revise` at the top of `startup.jl`, before anything else. It works on `dev`'d packages and on files you `includet` (`includet("script.jl")` — "include and track").

```
using Revise
includet("myscript.jl")     # edits to this file now take effect immediately
```

What Revise cannot do: change a struct definition. Redefining a `struct` requires a restart. This is the main reason to keep types in one small file and get them right early.

Module-level odds and ends

```
@__MODULE__           # the current module
parentmodule(f)
names(MyPkg)          # exported names
names(MyPkg; all = true)
isdefined(MyPkg, :foo)
@__FILE__, @__DIR__   # this source file / its directory — use for locating data files
pkgdir(MyPkg)         # root directory of a package, the right way to find assets
```

`__init__()` in a module runs at load time, after precompilation. Anything that can't be baked into a precompile cache (opening handles, reading env vars, registering with other packages) goes there.

```
function __init__()
    ENV_SETTING[] = get(ENV, "MYPKG_MODE", "default")
end
```

13. Interfaces

There is no `interface` keyword, no `implements`, and no compile-time check. An interface in Julia is a documented set of methods; you implement it by defining those methods, and in exchange a large amount of generic code starts working on your type.

This is looser than Go's structural typing (which at least verifies at compile time) and much looser than a Java interface. The compensation is that you can retrofit an interface onto a type you don't own.

Iteration

Define `iterate` and you get `for`, `collect`, `map`, `sum`, comprehensions, `zip`, destructuring, and everything else.

```
struct Countdown
    n::Int
end

# first call: no state argument. Return (value, state) or nothing.
```

```
Base.iterate(c::Countdown) = c.n > 0 ? (c.n, c.n - 1) : nothing
# subsequent calls
Base.iterate(::Countdown, state) = state > 0 ? (state, state - 1) : nothing

# strongly recommended extras
Base.length(c::Countdown) = c.n                 # enables collect to preallocate
Base.eltype(::Type{Countdown}) = Int            # enables a typed result array

collect(Countdown(3))          # [3, 2, 1]
sum(Countdown(3))              # 6
[x^2 for x in Countdown(3)]
```

If length is unknown, say so instead of defining `length`:

```
Base.IteratorSize(::Type{MyStream}) = Base.SizeUnknown()
Base.IteratorEltype(::Type{MyStream}) = Base.EltypeUnknown()
```

Indexing

```
Base.getindex(v::MyVec, i::Int) = v.data[i]
Base.setindex!(v::MyVec, x, i::Int) = (v.data[i] = x)
Base.firstindex(v::MyVec) = 1
Base.lastindex(v::MyVec) = length(v.data)     # makes `v[end]` work
```

AbstractArray

The highest-leverage interface in the language. Implement four methods and you inherit the entire array ecosystem — broadcasting, linear algebra, printing, slicing, reductions.

```
struct Squares <: AbstractVector{Int}
    n::Int
end

Base.size(s::Squares) = (s.n,)
Base.getindex(s::Squares, i::Int) = i^2
Base.IndexStyle(::Type{<:Squares}) = IndexLinear()
# and setindex! if it's mutable

s = Squares(5)
s[3]           # 9
sum(s)         # 55
s .+ 1         # works
s[2:4]         # works
println(s)     # prints as a 5-element vector, with the right header
```

For multidimensional, `size` returns a longer tuple, `getindex(A, i::Int, j::Int)`, and `IndexCartesian()`.

Display

```
Base.show(io::IO, x::MyType) = print(io, "MyType(", x.a, ")")          # compact
Base.show(io::IO, ::MIME"text/plain", x::MyType) = ...                 # REPL, verbose
Base.show(io::IO, ::MIME"text/html", x::MyType) = ...                  # notebooks
```

Equality, hashing, ordering

```
Base.:(==)(a::T, b::T) = ...
Base.isequal(a::T, b::T) = ...
Base.hash(x::T, h::UInt) = hash(x.field, hash(:T, h))     # must match ==
Base.isless(a::T, b::T) = ...                          # gives you sort, maximum, extrema
```

Define `hash` whenever you define `==`, or dictionaries silently misbehave.

Numbers

To make your type behave arithmetically:

```
Base.:+(a::Money, b::Money) = Money(a.cents + b.cents)
Base.:-(a::Money, b::Money) = Money(a.cents - b.cents)
Base.:*(a::Money, k::Real)  = Money(round(Int, a.cents * k))
Base.zero(::Type{Money}) = Money(0)
Base.one(::Type{Money}) = Money(100)
Base.convert(::Type{Money}, x::Real) = Money(round(Int, 100x))
Base.promote_rule(::Type{Money}, ::Type{<:Integer}) = Money
```

`zero` and `one` matter because generic code (`sum`, `prod`, matrix operations) calls them to get identity elements.

Broadcasting

By default a custom type broadcasts as a container if it's iterable, or errors. To make it behave as a scalar:

```
Base.broadcastable(x::MyConfig) = Ref(x)
```

For a custom array type that should produce its own type from broadcasts, implement `BroadcastStyle` — worth reading the manual section rather than guessing.

Traits, for interfaces that cut across the type tree

You get one supertype. When a property applies to types scattered across the hierarchy, use a trait: a function returning a singleton, dispatched on in a helper.

```
abstract type Storage end
struct InMemory <: Storage end
struct OnDisk <: Storage end

storage(::Type) = InMemory()               # default
storage(::Type{<:BigTable}) = OnDisk()

process(x) = _process(storage(typeof(x)), x)
_process(::InMemory, x) = ...
_process(::OnDisk, x) = ...
```

`Base` uses this everywhere: `IndexStyle`, `IteratorSize`, `OrderStyle`, `ArithmeticStyle`.

Interface checklist for a new container type

Ask what you actually need. A rough ladder:

1. `show` — always.
2. `==` and `hash` — if instances are compared or used as keys.
3. `iterate`, `length`, `eltype` — if it's a collection.
4. `getindex`, `setindex!`, `firstindex`, `lastindex` — if it's indexable.
5. `size`, `getindex`, `IndexStyle`, `<: AbstractArray` — if it's array-like. Do this rather than 3-4 if it applies; you get far more for the same effort.
6. `isless` — if it's orderable.
7. `broadcastable` — if it should broadcast as a scalar.

Checking your work

There's no compiler check, so use tests. `Aqua.jl` catches structural problems (ambiguities, unbound type parameters, piracy). For the `AbstractArray` interface specifically, `ArrayInterfaceTests`-style packages and `JET.jl` help. Otherwise: write a testset that exercises the generic functions you claim to support.

14. Metaprogramming

Julia is homoiconic: code is data, represented as ordinary `Expr` objects you can build, inspect and transform, and macros run at parse time to rewrite that data before compilation. This is Lisp's model with an infix syntax, and it's used far more routinely than in most languages — `@test`, `@view`, `@.`, `@time`, `@kwdef`, `@enum`, `@threads`, `@printf` are all macros, and you'll read macro-heavy packages long before you write one.

The pipeline

```
source text
  -> parse   -> Expr (syntax tree)        <- macros operate HERE
```

```
  -> lower   -> lowered IR (SSA-ish)
  -> infer   -> typed IR                  <- @generated functions operate HERE
  -> codegen -> LLVM IR -> machine code
```

Inspect each stage:

```
Meta.parse("1 + x*2")
@macroexpand @time f(x)
@code_lowered f(1)
@code_typed f(1)
@code_warntype f(1)
@code_llvm f(1)
@code_native f(1)
```

Expr

```
ex = :(1 + x * 2)
typeof(ex)          # Expr
ex.head             # :call
ex.args             # Any[:+, 1, :(x * 2)]
dump(ex)            # full recursive structure — the tool you'll use most
Meta.@dump 1 + x*2  # same, as a macro
```

The four kinds of leaf you'll encounter:

Thing	Example	Notes
`Symbol`	`:x`, `:+`	an identifier
Literal	`1`, `"s"`, `3.0`	stored as itself
`Expr`	`:(f(x))`	a node: `head` plus `args`
`QuoteNode`	`:(:x)`	a *quoted symbol* as data, not an identifier

The `QuoteNode` distinction bites early: `:x` inside a generated expression means "the variable x", while `QuoteNode(:x)` means "the symbol `:x` as a value". `dump(:(a.b))` shows `a` as a Symbol and `b` wrapped in a `QuoteNode`.

Common heads:

```
:(f(x, y))      # head :call,      args [:f, :x, :y]
:(x = 1)        # head :(=),       args [:x, 1]
:(x + y)        # head :call,      args [:+, :x, :y]     — operators are calls
:(a.b)          # head :.,         args [:a, QuoteNode(:b)]
:(a[i])         # head :ref
:(if c; a; end) # head :if
:(for i=1:3;end)# head :for
:(function f(x); x; end)   # head :function
:(x::Int)       # head :(::)
:(begin; a; b; end)        # head :block
:(@m x)         # head :macrocall, args [Symbol("@m"), LineNumberNode, :x]
```

Building expressions by hand:

```
Expr(:call, :+, 1, 2)          # equivalent to :(1 + 2)
eval(Expr(:call, :+, 1, 2))    # 3
```

Quoting and interpolation

```
:(x + 1)                # short quote, one expression
quote                   # long quote, a block; inserts LineNumberNodes
    a = 1
    a + 1
end

x = 5
:( $x + 1 )             # interpolate a VALUE:  :(5 + 1)
```

```
:( $(x + 1) )            # evaluate then splice: :(6)
sym = :y
:( $sym + 1 )            # :(y + 1)

args = [:a, :b, :c]
:( f($(args...)) )       # splatting into a call: :(f(a, b, c))
```

Interpolation in a quote is the mirror image of interpolation in a string, and behaves the same way. `$` reaches out into the surrounding *runtime* scope; everything else stays as syntax.

eval

```
eval(:(x = 5))           # defines x in the current module
@eval begin
    const A = 1
end
```

`eval` always operates at **module top level**, not in the local scope where you called it. You cannot `eval` a local variable into existence inside a function. Treat `eval` as a code-generation tool used at load time, not a runtime facility.

The legitimate everyday use is generating many similar definitions:

```
for op in (:+, :-, :*)
    @eval Base.$op(a::Money, b::Money) = Money($op(a.cents, b.cents))
end

for (name, unit) in [(:meters, 1.0), (:feet, 0.3048)]
    @eval $(Symbol("to_", name))(x) = x / $unit
end
# defines to_meters, to_feet
```

This pattern — a loop over names, `@eval`, `Symbol(...)` to build identifiers, `$` to splice — is 90% of the metaprogramming in real packages.

Macros

A macro receives **unevaluated syntax** and returns syntax, at parse time.

```
macro sayhello(name)
    return :( println("hello, ", $name) )
end

@sayhello "world"
@sayhello("world")       # same thing; both forms are legal
```

Inspect what it produces — always do this while developing:

```
@macroexpand @sayhello "world"
@macroexpand1 @outer @inner x    # expand only one level
```

Arguments arrive as `Expr`/`Symbol`/literals, so you can inspect and rewrite them:

```
macro showexpr(ex)
    quote
        println($(string(ex)), " = ", $(esc(ex)))
    end
end

x = 3
@showexpr x + 1          # prints "x + 1 = 4"
```

That's essentially how `@show` and `@assert` work: they stringify the syntax *and* evaluate it, which no function can do.

Hygiene and esc

Macros are hygienic: identifiers introduced inside the macro body are renamed to unique gensyms, so they can't collide with the caller's variables. This is good, and it also means that anything you want to refer to the *caller's* scope must be escaped.

```
macro double_bad(x)
    :( $x + $x )         # $x is inserted as syntax; usually fine for expressions
end

macro setlocal_bad()
    :( y = 1 )           # defines a gensym'd variable; caller's `y` is untouched
end

macro setlocal(name)
    :( $(esc(name)) = 1 )   # escaped: assigns the caller's variable
end
```

Rules of thumb:

- `esc(ex)` means "interpret this in the caller's scope".
- Escape user-supplied expressions when they must see caller-local variables. In practice, escape almost every argument.
- Do *not* escape names you introduce yourself (temporaries), or you'll clobber the caller.
- If you need a temporary in an escaped context, make it explicitly unique with `gensym()`.

```
macro twice(ex)
    tmp = gensym(:tmp)
    quote
        $(esc(ex))
        $(esc(ex))
    end
end

macro once(ex)
    tmp = gensym()
    quote
        $tmp = $(esc(ex))     # evaluate ONCE, then use twice
        ($tmp, $tmp)
    end
end
```

The double-evaluation bug is the classic macro mistake: `@twice f()` calls `f` twice, which is fine if intended and a disaster if not. Bind to a gensym first.

A useful complete example

A macro that times an expression and labels it with its own source text:

```
macro timeit(label, ex)
    quote
        local t0 = time_ns()
        local val = $(esc(ex))
        local dt = (time_ns() - t0) / 1e9
        @info "timing" label = $(esc(label)) seconds = dt
        val
    end
end

@timeit "load" load_data(path)
```

Note `local` on the macro's own temporaries (belt and braces alongside hygiene), `esc` on both user arguments, and returning `val` so the macro is expression-like rather than statement-like.

Macros that define things

```
macro defgetter(T, field)
    fname = esc(Symbol("get_", field))
    quote
        $fname(x::$(esc(T))) = getfield(x, $(QuoteNode(field)))
```

```
    end
end

@defgetter Point x     # defines get_x(p::Point) = p.x
```

Note `QuoteNode(field)` — `field` is the symbol `:x`, and we need it to appear in the generated code as the *value* `:x`, not as the identifier `x`.

Macro hygiene escape hatches

```
macro m()
    :( $(esc(:x)) )        # explicitly refer to the caller's `x` by name
end
__module__                 # inside a macro: the module it's expanded in
__source__                 # LineNumberNode of the call site
```

`__module__` is how a macro can `eval` into the caller's module or look up bindings there.

String and command macros

```
r"\d+"                     # calls @r_str("\\d+")
raw"c:\path"               # @raw_str
big"1.1"                   # @big_str
`ls -la`                   # Cmd literal, not a macro but the same family
```

Define your own:

```
macro sql_str(s)
    :( prepare_query($s) )
end

sql"SELECT * FROM t WHERE id = 1"
```

Non-standard string literals get the raw, un-interpolated text, which is why regex literals don't need double-escaping. Flags after the closing quote (`r"x"i`) arrive as a second argument.

Generated functions

`@generated` functions run at **compile time**, after types are known but before code is generated, and return an *expression* to be compiled for that specific type signature. Inside, arguments are their **types**, not values.

```
@generated function unrolled_sum(t::NTuple{N, T}) where {N, T}
    ex = :(t[1])
    for i in 2:N
        ex = :($ex + t[$i])
    end
    return ex
end

unrolled_sum((1, 2, 3))    # compiles to literally t[1] + t[2] + t[3]
```

Uses: unrolling loops whose length is in the type (tuples, StaticArrays), generating specialised code from type parameters, avoiding runtime introspection.

Constraints, which are strict:

- You see types only, never values.
- The body must be pure — no side effects, no I/O, no observing global state that might change.
- You cannot call functions defined later, or newly defined methods (no "world age" surprises allowed).
- Debugging is unpleasant.

Almost always, a plain function plus `Val`, or a `@nexprs`-style macro from `Base.Cartesian`, does the job with less pain. Reach for `@generated` last.

Reading and rewriting expressions

Manual walking:

```
function replace_symbol(ex, from, to)
    ex isa Symbol && return ex === from ? to : ex
    ex isa Expr || return ex
    Expr(ex.head, map(a -> replace_symbol(a, from, to), ex.args)...)
end

replace_symbol(:(a + b*a), :a, :z)     # :(z + b*z)
```

`MacroTools.jl` makes this far less tedious and is what most packages use:

```
using MacroTools

@capture(ex, f_(args__))               # destructure by pattern; binds f and args
@capture(ex, function fname_(params__); body__; end)

postwalk(x -> x isa Symbol && x === :a ? :z : x, ex)
prewalk(...)
striplines(ex)                         # drop LineNumberNodes for readable dumps
rmlines(ex)
@q quote ... end                       # quote without LineNumberNodes
```

_ suffixes in `@capture` patterns are the binders; __ slurps a sequence. It's the pattern matching you'd otherwise write by hand.

Reflection at runtime

```
methods(f); which(f, Tuple{Int}); @which f(1)
fieldnames(T); fieldtypes(T); nfields(x)
getfield(x, :a); setfield!(x, :a, 1)
getproperty(x, :a)                     # what x.a lowers to; overloadable
propertynames(x)
isdefined(Main, :x); @isdefined x
names(SomeModule; all = true)
functionloc(f, Tuple{Int})
Base.return_types(f, (Int,))
```

Overloading property access is a legitimate, non-macro form of metaprogramming:

```
struct Lazy
    d::Dict{Symbol,Any}
end
Base.getproperty(l::Lazy, s::Symbol) = getfield(l, :d)[s]
Base.propertynames(l::Lazy) = keys(getfield(l, :d))
```

Note `getfield(l, :d)` inside — using `l.d` would recurse infinitely.

When not to use macros

The honest guidance, given how tempting this all is:

- If a function will do, use a function. Macros don't compose, can't be passed around, can't be `mapped`, and confuse tooling.
- Macros are justified when you need the *unevaluated syntax*: to stringify it (`@show`, `@assert`), to delay or repeat evaluation (`@time`, `@threads`), to rewrite it (`@.`, `@views`), or to generate definitions (`@kwdef`, `@enum`).
- Metaprogramming at load time (a `for` loop with `@eval`) is much safer than a macro and covers most "I have twenty near-identical methods" cases.
- Every macro you write is a small DSL your future self has to remember. The ecosystem's macro-heavy packages are macro-heavy because they're implementing genuinely new syntax (JuMP's constraint algebra, Turing's model notation), not because macros were convenient.

Debugging checklist

```
@macroexpand @mymacro x          # what did it produce?
MacroTools.striplines(@macroexpand @mymacro x)    # readable version
```

```
dump(:(the thing I'm matching))   # what does the AST actually look like?
Meta.show_sexpr(ex)               # lisp-style view, sometimes clearer than dump
```

Symptoms and causes:

Symptom	Likely cause
`UndefVarError` for a caller's variable	missing `esc`
Caller's variable mysteriously overwritten	escaped a temporary you introduced
Argument evaluated twice	interpolated `$(esc(ex))` in two places; bind to a gensym
Works at top level, fails in a function	used `eval`, which is module-scoped
"no method matching" on a Symbol	forgot `QuoteNode` where a symbol had to appear as a value

15. Concurrency and parallelism

Nothing here resembles the BEAM. Shared mutable memory, real OS threads, cooperative scheduling, no process isolation, no supervision, no mailboxes bound to addresses. Data races are possible and silent.

Four mechanisms, for four different problems:

Mechanism	For	Memory	Parallel?
`@async` / `Task`	I/O concurrency	shared	no (concurrent only)
`Threads.@spawn` / `@threads`	CPU work	shared	yes
`Distributed` (`addprocs`)	multi-process / multi-machine	separate	yes
`@simd`, vectorisation	inner loops	—	instruction-level

Tasks (coroutines)

Green threads, cooperatively scheduled. They yield at I/O, `sleep`, `wait`, channel operations — never preemptively. A tight CPU loop in a task blocks everything on that thread.

```
t = @async begin
    sleep(1)
    "done"
end

istaskdone(t); istaskstarted(t); istaskfailed(t)
wait(t)              # block until finished, rethrow on failure
fetch(t)             # wait and return the value
```

Concurrent I/O:

```
tasks = [@async download(u) for u in urls]
results = fetch.(tasks)
```

trap: an exception in a task is stored, not raised. If you never `fetch` or `wait`, the failure is invisible. Always collect your tasks.

```
@sync begin              # waits for all tasks spawned inside, and surfaces errors
    for u in urls
        @async process(u)
    end
end
```

`@sync` is the structured-concurrency primitive; use it rather than a bare list of `@async` calls whenever you can.

Threads

Start Julia with threads or you get one:

```
julia -t auto            # or JULIA_NUM_THREADS=auto
```

```
Threads.nthreads()
Threads.threadid()       # do NOT use this to index into per-thread state; see below
```

`@threads` — static partition of a loop across threads. Best for uniform work over an indexable range.

```
Threads.@threads for i in eachindex(v)
    v[i] = expensive(i)          # distinct indices per iteration: safe
end
```

`@spawn` — dynamic scheduling, composable, nests properly. Prefer it for uneven work and for anything that itself spawns.

```
tasks = map(chunks) do c
    Threads.@spawn process(c)
end
results = fetch.(tasks)

# divide and conquer
function psum(v)
    length(v) < 10_000 && return sum(v)
    mid = length(v) ÷ 2
    left = Threads.@spawn psum(@view v[1:mid])
    right = psum(@view v[mid+1:end])
    fetch(left) + right
end
```

`OhMyThreads.jl` wraps the common patterns (`tmap`, `treduce`, `tforeach`, chunking, scheduling policies) and is worth using instead of hand-rolling.

Data races

Real, silent, and yours to prevent.

```
# BROKEN: concurrent read-modify-write on a shared scalar
total = 0
Threads.@threads for i in 1:10^6
    total += i           # lost updates
end
```

Fixes, in order of preference:

```
# 1. Per-task partial results, combined at the end. Fastest, no contention.
partials = map(Iterators.partition(1:10^6, 10^4)) do chunk
    Threads.@spawn sum(chunk)
end
total = sum(fetch.(partials))

# 2. Atomics, for counters
counter = Threads.Atomic{Int}(0)
Threads.atomic_add!(counter, 1)
counter[]

# 3. Locks, for anything more complex
const L = ReentrantLock()
Threads.@threads for x in items
    y = compute(x)               # do the work OUTSIDE the lock
    lock(L) do
        push!(shared, y)
    end
end
```

Don't index per-thread buffers by `threadid()`. With dynamic scheduling a task can migrate between threads mid-execution, so `buffers[threadid()]` is not safe. Use one buffer per task (per chunk) instead.

Writing to *distinct* indices of a shared array from different threads is safe. Growing a shared `Vector` with `push!` from multiple threads is not.

Channels

Typed, bounded queues. Closest thing to a mailbox, but they are objects you pass around, not addresses attached to a process.

```
ch = Channel{Int}(32)          # buffered, capacity 32
ch = Channel{Int}(0)           # unbuffered: put! blocks until a take!
put!(ch, 1)
x = take!(ch)
isready(ch); close(ch)

for x in ch                    # iterates until the channel is closed
    process(x)
end
```

Producer bound to a task (closes automatically when the task ends):

```
producer = Channel{Int}(10) do ch
    for i in 1:100
        put!(ch, i)
    end
end
collect(producer)
```

Worker pool:

```
jobs = Channel{Int}(100)
results = Channel{Int}(100)

for _ in 1:Threads.nthreads()
    Threads.@spawn for j in jobs
        put!(results, expensive(j))
    end
end

foreach(j -> put!(jobs, j), 1:1000)
close(jobs)
```

Taking from a closed, empty channel throws `InvalidStateException`; that's how consumers know to stop, and it's what the `for x in ch` form handles for you.

Distributed

Separate processes, separate heaps, message passing over sockets. For multi-machine work or for escaping the GC on shared-memory workloads.

```
using Distributed
addprocs(4)                    # local workers
addprocs([("host", 4)])        # over SSH
nworkers(); workers()

@everywhere using MyPkg        # load on every worker
@everywhere f(x) = x^2

r = remotecall_fetch(f, 2, 10)  # run f(10) on worker 2
pmap(f, 1:100)                 # parallel map, dynamic load balancing
@distributed (+) for i in 1:1000    # parallel reduction
    f(i)
end
```

```
using SharedArrays              # shared memory on one machine
S = SharedArray{Float64}(1000)
```

`Distributed` is a stdlib but no longer the default recommendation for single-machine parallelism — threads are cheaper. Use it for clusters, or when you need process isolation.

Async I/O details

```
sleep(0.1)                # yields
yield()                   # explicitly give up the scheduler
Timer(cb, 1; interval=5)  # periodic callback
@async while true; ...; sleep(60); end     # a background loop

# blocking a task on an external event
c = Condition()
notify(c, value)
wait(c)

Base.Event()              # one-shot, thread-safe
Semaphore(n); acquire(s); release(s)
```

`ReentrantLock`, `Semaphore`, `Condition`, `Event`, `Threads.Atomic` are the whole synchronisation toolkit.

What you don't get, coming from Elixir

- No supervision trees. If a task dies, nothing restarts it; write your own retry loop.
- No process isolation. A task that corrupts shared state corrupts it for everyone.
- No preemption. A CPU-bound task without `yield` starves its thread's other tasks (though other threads keep running).
- No `receive` with pattern matching. Channels are typed queues; you match on the value yourself.
- No transparent distribution. `Distributed` requires explicit `@everywhere` code loading and explicit placement.
- GC is stop-the-world across all threads, so latency has a floor that the BEAM doesn't have.

What you do get: genuine shared-memory parallelism over large arrays with no copying, which is the trade the language is making.

Debugging concurrency

```
julia -t 4 --check-bounds=yes
```

```
@sync                     # surface exceptions you'd otherwise lose
istaskfailed(t) && fetch(t)
```

There is no race detector in the standard toolchain. `ThreadSanitizer` builds exist but are not routine. The practical defence is structure: prefer pure functions returning values over shared mutation, use `@sync` and `fetch` everywhere, and keep the mutable region small enough to reason about.

16. Performance

Julia's performance model is unusual: the same source can be C-fast or Python-slow depending on whether the compiler can infer types. Nothing warns you when it can't. This section is the checklist for finding out.

The rules, in order of impact

1. **Put code in functions.** Top-level global scope is not optimised at all. A loop in a script and the same loop in a function can differ by 50x.
2. **Type stability.** The return type must be inferable from the argument types. This is the whole game.
3. **Concrete struct fields.** `::Float64` or `::T`, never bare or abstract.
4. **Avoid non-constant globals.** If a function reads a global, its type can change at any time, so the compiler must assume `Any`. Mark it `const`, or pass it as an argument.
5. **Preallocate and mutate** in hot loops instead of allocating per iteration.
6. **`@views` on slices**, `@inbounds` when you're certain.

Everything else is noise until these six are right.

Type stability

A function is type-stable if the type of every variable, and the return type, is determined by the argument types alone.

```
function unstable(n)
    s = 0                # Int
    for i in 1:n
        s += i / 2       # Float64 — s is now Union{Int, Float64}
    end
    s                    # return type depends on whether the loop ran
end

function stable(n)
    s = 0.0
    for i in 1:n
        s += i / 2
    end
    s
end
```

Other common instability sources:

```
# returning different types from branches
f(x) = x > 0 ? 1 : "negative"          # Union{Int, String}

# accumulating into a container of the wrong type
v = []                                  # Vector{Any}
v = Int[]                               # Vector{Int}

# a global
const CONFIG = load_config()            # const, so its type is known
scale(x) = x * CONFIG.factor            # fine
# without const: every call re-checks the type of CONFIG

# hardcoded literals in generic code
function mysum(v)
    s = 0                               # breaks for Vector{Float32}, Complex, Dual, ...
    s = zero(eltype(v))                 # correct
    for x in v; s += x; end
    s
end
```

Small unions (`Union{Int, Nothing}`) are optimised and fine. Large or open unions, and `Any`, are not.

Finding it: @code_warntype

```
@code_warntype mysum([1.0, 2.0])
```

Read the output for red text and `Any` / large `Union` types. The pattern to look for:

```
Body::Union{Float64, Int64}     <- red: unstable return
  s::Union{Float64, Int64}      <- red: unstable local
```

`Cthulhu.jl` (`@descend mysum(v)`) lets you walk down into callees interactively, which is what you need once the problem is two layers deep. `JET.jl` (`@report_opt f(x)`) reports optimisation failures across a whole call tree without manual descent.

```
using JET
@report_opt mysum(v)        # runtime dispatch, allocation-forcing instability
@report_call mysum(v)       # possible errors, method errors
```

Measuring

```
using BenchmarkTools

@btime mysum($v)                    # minimum time over many samples
```

```
@benchmark mysum($v)                # full distribution
@allocated mysum(v)                 # bytes allocated — 0 is the goal in hot paths
@time mysum(v)                      # includes compilation on first call; run twice
@elapsed
```

The `$` matters. `@btime f(v)` measures a global lookup of `v` plus the call; `@btime f($v)` interpolates the value and measures the call. Without it your numbers are wrong, usually by a lot.

Profiling:

```
using Profile
@profile expensive_thing()
Profile.print()                     # text tree
Profile.print(format = :flat, sortedby = :count)

using PProf; pprof()                # flamegraph in a browser
using ProfileView; @profview f(x)   # GTK flamegraph
@profview_allocs f(x)               # allocation profiler (1.8+)
```

The allocation profiler is often more actionable than the time profiler, because in Julia most performance problems present as allocations.

Allocation

Every allocation is GC pressure. Common sources and their fixes:

```
# slicing
s = sum(A[:, 1])            # allocates a column
s = sum(@view A[:, 1])      # doesn't

# intermediate arrays in expressions
y = 2 .* x .+ 1             # one allocation (the result) — fusion handles the rest
y = 2 * x + 1               # TWO allocations — no dots, no fusion
@. y = 2x + 1               # zero allocations, writes into existing y

# growing in a loop
out = Float64[]
for x in v; push!(out, f(x)); end        # repeated reallocation
out = similar(v); for i in eachindex(v); out[i] = f(v[i]); end   # preallocated
sizehint!(out, length(v))                 # if you must push!

# closures capturing reassigned variables (boxing)
function make()
    n = 0
    () -> (n += 1)          # n is boxed
end

# splatting a runtime-length collection
f(v...)                     # slow if length(v) is not statically known

# string concatenation in a loop
s *= x                      # quadratic; use an IOBuffer
```

Small immutable structs and tuples usually do *not* allocate — they live in registers or inline in their parent. `StaticArrays.SVector{3}` is the tool for small fixed-size vectors in hot loops; it's often 10x faster than `Vector` for 3-D geometry.

Bounds checking and SIMD

```
@inbounds for i in eachindex(v)
    v[i] *= 2
end

@simd for i in eachindex(v)         # allow reordering of float reductions
```

```
    s += v[i]
end

@fastmath                           # relax IEEE semantics. Know what you're giving up.
```

`@inbounds` on an incorrect index is memory corruption, not an error. Use `eachindex`/`axes` so the bounds are provably correct, and test with `--check-bounds=yes` in CI.

Julia auto-vectorises simple loops; `LoopVectorization.jl`'s `@turbo` goes further, at the cost of a heavy dependency and stricter requirements on the loop body.

Memory layout

- Column-major: innermost loop over the first index.
- Array of structs vs struct of arrays: `StructArrays.jl` gives you the AoS interface over SoA storage, which is usually what you want for numerical work over records.
- `Vector{MyImmutableStruct}` stores the structs inline. `Vector{MyMutableStruct}` stores pointers. That difference alone is often 5x.

Compilation latency

First-call compilation ("time to first X") is a separate problem from run-time speed.

- Keep one long-lived session and use Revise. Restarting is a rebuild.
- `PrecompileTools.jl`'s `@compile_workload` bakes representative calls into the package's precompile cache, so your users don't pay the cost.
- `PackageCompiler.jl` builds a custom sysimage with your dependencies baked in — the right answer for a CLI tool or a slow-to-load analysis stack.
- Excessive specialisation (dispatching on many type combinations, heavy `@generated` use) increases compile time. `@nospecialize` on arguments where you don't need specialised code.

A workflow

1. Write the obvious version. Make it correct, with tests.
2. `@btime` it. If it's fast enough, stop.
3. `@allocated` — is it allocating in a loop it shouldn't be?
4. `@code_warntype` or `@report_opt` on the hot function. Fix instability first; it's usually the whole problem.
5. `@profview` to find where the time actually goes, rather than where you assume.
6. Only then: `@inbounds`, `@views`, StaticArrays, manual unrolling.

Steps 4 and 5 in the wrong order is the standard mistake — micro-optimising a function that's dispatching dynamically achieves nothing.

Quick reference

```
@time, @timev, @elapsed, @allocated
@btime, @benchmark, @belapsed          # BenchmarkTools, with $ interpolation
@code_warntype, @code_typed, @code_llvm, @code_native, @code_lowered
@descend                               # Cthulhu
@report_opt, @report_call              # JET
@profile, @profview, @profview_allocs
GC.gc(); GC.enable(false)              # rarely; measuring GC impact
Base.summarysize(x)                    # deep size in bytes
@inbounds, @simd, @fastmath, @views, @inline, @nospecialize
```

17. Testing

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.

18. Standard tasks

Files and paths

```
read("f.txt", String)
readlines("f.txt")
write("f.txt", "content")
readchomp(`cmd`)

open("out.csv", "w") do io
    println(io, "a,b")
    write(io, data)
end

for line in eachline("big.log")      # streaming, constant memory
    process(line)
end

readdir("."); readdir("."; join = true)
walkdir(".")                         # recursive, yields (root, dirs, files)
isfile, isdir, ispath, islink
mkdir, mkpath, rm(p; recursive = true, force = true)
cp(src, dst; force = true); mv
joinpath("a", "b", "c.txt")
splitpath, splitdir, splitext, basename, dirname
abspath, normpath, realpath, expanduser("~/x")
homedir(), tempdir(), tempname()
mktemp() do path, io ... end
mktempdir() do dir ... end
filesize(p), mtime(p), stat(p)
touch(p)
@__DIR__                             # directory of the current source file
pkgdir(MyPkg)                        # a package root — the way to find its assets
```

Binary IO:

```
data = read("f.bin")                 # Vector{UInt8}
open("f.bin") do io
    x = read(io, Int64)
    v = read(io, 100)
    seek(io, 0); position(io); eof(io)
end
```

Processes

Backticks build a `Cmd`. **There is no shell**, so no quoting bugs and no injection.

```
run(`ls -la`)                          # throws on non-zero exit
read(`git rev-parse HEAD`, String)
readchomp(`hostname`)
success(`which julia`)                 # Bool instead of throwing

file = "name with spaces.txt"
run(`cat $file`)                       # passed as ONE argument, correctly

run(pipeline(`cat f.txt`, `grep x`))
run(pipeline(`cmd`, stdout = "out.txt", stderr = "err.txt"))
open(`sort`, "w", stdout) do io
    println(io, "b"); println(io, "a")
end

p = run(`long-thing`; wait = false)
kill(p); process_running(p); p.exitcode

withenv("VAR" => "value") do
    run(`printenv VAR`)
end
addenv(`cmd`, "K" => "v")
Cmd(`cmd`; dir = "/tmp")
```

If you genuinely need shell features, invoke a shell explicitly:

```
run(`sh -c "a | b > c"`)
```

Environment and arguments

```
ENV["HOME"]
get(ENV, "PORT", "8080")
haskey(ENV, "CI")
ARGS                                   # Vector{String} of command-line args
PROGRAM_FILE
Sys.iswindows(), Sys.islinux(), Sys.isapple()
Sys.CPU_THREADS, Sys.total_memory()
VERSION                                # v"1.12.6"
@static if Sys.islinux() ... end       # compile-time branch
```

Argument parsing: `ArgParse.jl` (full-featured, argparse-shaped) or `Comonicon.jl` (turns a function signature into a CLI).

Dates and time

```
using Dates
now(); today(); now(UTC)
Date(2026, 8, 17); DateTime(2026, 8, 17, 14, 30)
Date("2026-08-17"); Date("17/08/2026", dateformat"dd/mm/yyyy")
Dates.format(now(), "yyyy-mm-dd HH:MM:SS")

now() + Day(3); now() - Hour(2)
Year, Month, Week, Day, Hour, Minute, Second, Millisecond
year(d), month(d), day(d), dayofweek(d), dayofyear(d), isleapyear(d)
d1 - d2                                # a Period
Dates.value(d1 - d2)                   # as a number
firstdayofmonth(d), lastdayofmonth(d)
Date(2026,1,1):Day(1):Date(2026,12,31)   # ranges of dates work

time()                                 # Unix seconds, Float64
```

```
time_ns()                              # monotonic nanoseconds — use for timing
unix2datetime(t), datetime2unix(d)
```

Timezones are not in the stdlib: `TimeZones.jl` (`ZonedDateTime`, `TimeZone("Europe/London")`).

Randomness

```
using Random
rand()                    # Float64 in [0,1)
rand(1:6); rand(v); rand(3, 3); rand(Bool)
randn(100)                # standard normal
randexp, randstring(8)
shuffle(v); shuffle!(v)
randperm(10); randsubseq(v, 0.1)
sample(v, 5; replace = false)         # StatsBase.jl
Random.seed!(42)
rng = MersenneTwister(42); rand(rng, 10)
rng = Xoshiro(42)                     # the current default algorithm
```

For reproducibility across Julia versions use `StableRNGs.jl` — the default RNG is explicitly allowed to change.

Data wrangling in Base

```
sort(v; by = x -> x.age, rev = true)
sort(v; lt = (a, b) -> a.n < b.n)
sortperm(v); partialsort(v, 1:10); partialsortperm
searchsorted(sorted_v, x); searchsortedfirst
filter(f, v); count(f, v); any(f, v); all(f, v)
findfirst, findlast, findall, findnext
findmax(v), argmax(v), argmax(f, v)
unique(v), unique(f, v), allunique(v)
union, intersect, setdiff, symdiff, issubset
reduce, foldl, mapreduce, accumulate, cumsum, cumprod
zip, enumerate, pairs
Iterators.partition(v, 3), flatten, product, take, drop, cycle, repeated, countfrom
group = Dict(); for x in v; push!(get!(group, key(x), []), x); end   # groupby by hand
```

`SplitApplyCombine.jl` gives you `group`, `groupreduce`, `innerjoin` for plain collections without a DataFrame.

Statistics and linear algebra (stdlib)

```
using Statistics
mean(v), median(v), std(v), var(v), quantile(v, [0.25, 0.75]), cor(x, y), cov(x, y)
mean(A; dims = 1)

using StatsBase          # not stdlib, but the standard extension
countmap(v), sample, weights, fit(Histogram, v), zscore, describe
```

Linear algebra has its own section (18).

Serialisation

```
using JSON3
obj = JSON3.read(json_string)              # lazy, indexable
obj = JSON3.read(json_string, MyStruct)    # typed, with StructTypes.jl
JSON3.write(obj)
JSON3.pretty(obj)

using StructTypes
StructTypes.StructType(::Type{MyStruct}) = StructTypes.Struct()
```

```
using CSV, DataFrames
df = CSV.read("data.csv", DataFrame)
CSV.write("out.csv", df)
CSV.File("data.csv") |> collect            # without DataFrames
```

```
using Serialization          # Julia-native binary; version-fragile, don't archive with it
serialize("f.jls", obj); deserialize("f.jls")

using JLD2                   # HDF5-based, good for arrays and long-term storage
jldsave("f.jld2"; a = 1, b = v)
load("f.jld2", "a")

using TOML                                  # stdlib
TOML.parsefile("Project.toml")
TOML.print(dict)
```

HTTP

```
using HTTP

r = HTTP.get("https://example.com")
r.status; String(r.body); Dict(r.headers)

HTTP.post(url; body = JSON3.write(payload),
              headers = ["Content-Type" => "application/json"])
HTTP.get(url; query = Dict("q" => "x"), retry = true, readtimeout = 30)

# server
router = HTTP.Router()
HTTP.register!(router, "GET", "/health", req -> HTTP.Response(200, "ok"))
HTTP.serve(router, "0.0.0.0", 8080)
```

`Oxygen.jl` wraps this with routing macros and OpenAPI generation if you want less ceremony. `Downloads.download(url, path)` (stdlib) is enough for fetching a file.

DataFrames, minimal

```
using DataFrames
df = DataFrame(a = 1:3, b = ["x","y","z"])

df.a                    # column, no copy
df[!, :a]               # column, no copy
df[:, :a]               # column, COPY
df[1, :]; df[1:2, [:a, :b]]
names(df); nrow(df); ncol(df); describe(df)

subset(df, :a => x -> x .> 1)
filter(:a => >(1), df)
transform(df, :a => (x -> 2x) => :doubled)
select(df, :a, :b => :renamed)
combine(groupby(df, :b), :a => sum => :total)
sort(df, :a; rev = true)
innerjoin(df1, df2; on = :id); leftjoin; outerjoin
stack(df); unstack(df)
```

The `source => function => destination` mini-language is the thing to learn; everything in DataFrames is expressed in it. `DataFramesMeta.jl` provides `@select`/`@subset`/`@by` macros if you prefer dplyr-shaped syntax.

Plotting

```
using Plots
plot(x, y; label = "series", xlabel = "t", lw = 2)
plot!(x, y2)             # add to the current plot (note the !)
scatter(x, y); histogram(v); heatmap(A); surface(x, y, z)
savefig("out.png")

# headless: avoid GR trying to open a window
ENV["GKSwstype"] = "100"
```

`Makie.jl` (CairoMakie for publication figures, GLMakie for interactive/GPU) is the more capable and more modern option, at the cost of heavier load time. `UnicodePlots.jl` draws in the terminal, which is genuinely useful over SSH.

19. Linear algebra

Half the reason the language exists. `LinearAlgebra` is a stdlib, so `using LinearAlgebra` needs no installation.

Basics

```
using LinearAlgebra

A = rand(3, 3); B = rand(3, 3); v = rand(3)

A * B              # matrix multiply
A .* B             # elementwise
A * v              # matrix-vector
v' * v             # inner product (scalar); v' is a lazy adjoint
v * v'             # outer product (3x3)
A'                 # adjoint (conjugate transpose), lazy
transpose(A)       # transpose without conjugation, lazy
A^2                # matrix power, not elementwise
A.^2               # elementwise
```

Solving

```
x = A \ b          # solve A x = b
x = A' \ b
X = A \ B          # multiple right-hand sides
```

`\` inspects the matrix, picks an appropriate factorisation (LU, Cholesky, QR for over-determined, least-squares for non-square), and dispatches to LAPACK. **Never write `inv(A) * b`** — slower, less accurate, and it computes something you didn't need.

For an over-determined system, `A \ b` gives the least-squares solution directly.

Reuse a factorisation across multiple solves:

```
F = lu(A)
x1 = F \ b1
x2 = F \ b2

F = cholesky(Symmetric(A))     # if A is symmetric positive definite
F = qr(A)
```

Factorisations

```
lu(A); qr(A); cholesky(A); ldlt(A)
eigen(A)                       # returns an Eigen; F.values, F.vectors
eigvals(A); eigvecs(A)
svd(A)                         # F.U, F.S, F.V, F.Vt
schur(A); hessenberg(A)
bunchkaufman(A)
pinv(A)                        # Moore-Penrose pseudoinverse
```

Factorisation objects are lazy wrappers: `F \ b` uses the factors directly, `Matrix(F)` materialises.

Scalars from matrices

```
tr(A)              # trace
det(A); logdet(A); logabsdet(A)     # use logdet for likelihoods; det underflows
rank(A)
cond(A)            # condition number
norm(v); norm(v, 1); norm(v, Inf); opnorm(A, 2)
dot(u, v); u ⋅ v   # \cdot<tab>
cross(u, v); u × v # \times<tab>
```

Constructors and wrappers

```
I                          # UniformScaling: A + 2I works at any size, allocates nothing
Matrix(I, 3, 3)            # a dense identity when you actually need one
Diagonal([1,2,3])
Bidiagonal, Tridiagonal, SymTridiagonal
Symmetric(A), Hermitian(A)
UpperTriangular(A), LowerTriangular(A), UnitUpperTriangular(A)
diagm(0 => [1,2,3], 1 => [4,5])
diag(A); diag(A, 1)
```

These wrappers are the single highest-leverage thing in the package. Wrapping is not documentation — it **dispatches to a different algorithm**. `Symmetric(A) \ b` uses Bunch-Kaufman or Cholesky rather than general LU; `Diagonal(d) * A` is $O(n^2)$ instead of $O(n^3)$; `UpperTriangular(A) \ b` is a back-substitution. Wrapping costs nothing and can change the complexity class.

In-place operations

```
mul!(C, A, B)              # C = A*B, no allocation
mul!(C, A, B, α, β)        # C = α*A*B + β*C  (the full BLAS gemm)
ldiv!(F, b)                # b = F \ b
lmul!(2, v); rmul!(v, 2)
axpy!(α, x, y)             # y = α*x + y
copyto!(dest, src)
lu!(A); cholesky!(A)       # factorise in place, destroying A
```

In an iterative algorithm, preallocating and using `mul!` is usually the difference between allocating gigabytes and allocating nothing.

Sparse

```
using SparseArrays
S = sparse(I, J, V, m, n)          # from COO triplets
S = spzeros(1000, 1000)
S = sprand(1000, 1000, 0.01)
sparse(A); Matrix(S)               # convert both ways
nnz(S); nonzeros(S); rowvals(S); findnz(S)
droptol!(S, 1e-12); dropzeros!(S)

S \ b                              # sparse LU (UMFPACK) automatically
cholesky(S)                        # CHOLMOD for SPD sparse
```

Storage is CSC (compressed sparse *column*), consistent with the column-major convention. Building a sparse matrix by assigning into `spzeros` in a loop is pathologically slow — accumulate `I`, `J`, `V` vectors and call `sparse(I, J, V)` once, which also sums duplicate entries for you (exactly what finite-element assembly wants).

Iterative solvers and beyond

For large or matrix-free problems:

- **`IterativeSolvers.jl`** / **`Krylov.jl`** — CG, GMRES, MINRES, LSQR; take any object supporting `mul!`.
- **`LinearSolve.jl`** — a uniform interface over direct and iterative solvers, with algorithm selection and caching. The modern default for library code.
- **`LinearMaps.jl`** — represent a linear operator by its action, no matrix stored.
- **`Preconditioners.jl`**, **`IncompleteLU.jl`** — preconditioning.
- **`ArnoldiMethod.jl`** / **`KrylovKit.jl`** — a few eigenvalues of a large sparse matrix.
- **`StaticArrays.jl`** — `SMatrix{3,3}`, stack-allocated, with hand-unrolled small-matrix algebra. For 3-D geometry in a hot loop this is 10x.
- **`CUDA.jl`** / **`AMDGPU.jl`** — `CuArray` implements the same interfaces, so `A \ b`, `mul!`, broadcasting and most generic code work unchanged on device arrays.

BLAS control

```
BLAS.get_num_threads(); BLAS.set_num_threads(4)
BLAS.vendor()
```

trap: BLAS runs its own thread pool, independent of `JULIA_NUM_THREADS`. Running `Threads.@threads` over matrix operations oversubscribes the CPU badly — each Julia thread launches a multithreaded BLAS call. Set `BLAS.set_num_threads(1)` when you're parallelising at the Julia level.

Numerical notes

```
logdet(A)                    # not log(det(A))
A \ b                        # not inv(A) * b
issymmetric(A), isposdef(A), ishermitian(A)
normalize(v); normalize!(v)
nullspace(A); rank(A; rtol = ...)
kron(A, B)                   # Kronecker product
```

Floating-point comparisons of matrices: `A ≈ B` with `isapprox(A, B; rtol = ...)`, never `==`.

20. The REPL and Unicode

Modes

Key at an empty prompt	Mode	Prompt
—	Julia	`julia>`
`]`	Pkg	`(env) pkg>`
`?`	Help	`help?>`
`;`	Shell	`shell>`
Backspace	back to Julia	

```
julia> ?sort             # docs, all methods, related functions
julia> ]st               # one-off Pkg command without staying in the mode
julia> ;ls -la           # one-off shell command
```

Help mode also works on operators, macros, types and unicode characters — `?∘`, `?@view`, `?Vector`, and pasting a character tells you its name and how to type it.

Keybindings

Key	Does
`Ctrl-C`	interrupt the running expression
`Ctrl-D`	exit (at an empty prompt)
`Ctrl-L`	clear screen
`Ctrl-R` / `Ctrl-S`	reverse / forward history search
`Up/Down`	history, prefix-filtered by what you've typed
`Alt-Enter`	insert a newline without evaluating
`Ctrl-Q`	edit the current line in `$EDITOR`
`Tab`	completion — of names, paths, fields, and LaTeX sequences
`Tab` twice	show all completions, including method signatures

Typing `f(` then Tab shows the applicable methods. Typing `x.` then Tab shows fields and properties.

Useful REPL functions

```
ans                      # value of the last expression
varinfo()                # what's defined in Main, with sizes
varinfo(MyPkg)
clipboard("text"); clipboard()      # copy out / paste in
edit("file.jl", 10)      # open in $JULIA_EDITOR at a line
@edit f(1)               # open the source of the method that call selects
```

```
@which f(1)
@doc f
names(Base; all = false)
methods(f); methodswith(MyType)
apropos("sort")           # full-text search of docstrings
InteractiveUtils.versioninfo()
```

startup.jl

`~/.julia/config/startup.jl` runs before every session. Keep it small and fast.

```
try
    using Revise
catch e
    @warn "Revise failed to load" exception = (e, catch_backtrace())
end

atreplinit() do repl
    try
        @eval using OhMyREPL
    catch
    end
end

ENV["JULIA_EDITOR"] = "zed --wait"
```

Install these into the shared default environment, not into your projects:

```
julia -e 'using Pkg;
    Pkg.activate("v$(VERSION.major).$(VERSION.minor)"; shared=true);
    Pkg.add(["Revise", "OhMyREPL", "BenchmarkTools"])'
```

`--startup-file=no` skips it — useful for benchmarking and required for reproducible CI.

Command-line flags worth remembering

```
julia --project                 # activate the nearest Project.toml
julia -t auto                   # all available threads
julia -p 4                      # 4 worker processes (Distributed)
julia -e 'code'                 # evaluate and exit
julia -E 'code'                 # evaluate, print the result, exit
julia -i script.jl              # run then stay interactive
julia --check-bounds=yes        # bounds-check even where @inbounds says not to
julia --startup-file=no
julia --banner=no -q            # quiet
julia --track-allocation=user   # write .mem files with per-line allocation counts
julia --code-coverage=user
```

Unicode input

Type the LaTeX name and press Tab. Works in the REPL, and in editors with the Julia extension.

Sequence	Char	Sequence	Char
`\alpha`	α	`\le`	≤
`\beta`	β	`\ge`	≥
`\gamma`	γ	`\ne`	≠
`\delta`	δ	`\approx`	≈
`\Delta`	Δ	`\equiv`	≡
`\epsilon`	ϵ	`\in`	∈
`\theta`	θ	`\notin`	∉
`\lambda`	λ	`\subseteq`	⊆
`\mu`	μ	`\cup`	∪
`\pi`	π	`\cap`	∩
`\rho`	ρ	`\cdot`	·
`\sigma`	σ	`\times`	×
`\Sigma`	Σ	`\div`	÷
`\phi`	ϕ	`\circ`	∘
`\omega`	ω	`\pm`	±
`\Omega`	Ω	`\sqrt`	√
`\infty`	∞	`\sum`	∑
`\partial`	∂	`\prod`	∏
`\nabla`	∇	`\int`	∫
`\hbar`	ℏ	`\emptyset`	∅
`\euler`	ℯ	`\im`	— (`im` is the literal)

Sub- and superscripts: `x\_1` gives x_1, `x\^2` gives x^2, `\_a` through `\_z` and `\^0` through `\^9` mostly exist. These are legal in identifiers, so σ^2, x_1, Δt are all valid variable names.

Built-in constants: π (also `pi`), *e* (also `exp(1)`), `im` (imaginary unit), `Inf`, `NaN`, `missing`, `nothing`.

Operators you can define yourself, because they're parsed as infix: ⊕ ⊗ ⊙ ⊘ ⋆ ∘ ± ∓ × · ∪ ∩ ⊆ ⊇ ≈ ≉ ≡ and many more. Precedence follows the ASCII operator each resembles.

```
⊕(a, b) = a + 2b
3 ⊕ 4       # 11
```

If you don't want Unicode in your source, every operator has an ASCII spelling: `in`, `issubset`, `isapprox`, `!=`, `<=`, `div`, `dot`, `cross`, `union`, `intersect`, `pi`, `sqrt`.

Finding out what a character is

Paste it into help mode:

```
help?> ⊗
"⊗" can be typed by \otimes<tab>
```

That works for any character, which is the fastest way to read unfamiliar numerical code.