Julia Fast scientific language

START HERE

`julia`
Open the REPL

`]`
Package mode

`julia <file.jl> <args>`
Run a script

`?`
Help mode

`julia -e '<expr>'`
Run one expression

`julia -t auto`
Use all cores

`julia --project=.`
Use this folder's env

`include("<file.jl>")`
Load a file in the REPL

Generic function name: area → `call with args` → **Dispatch** all argument types → `first call` → **Method** specialized, compiled → `cached` → **Fast native code** reused after

CONCEPTS

Julia compiles each method on first call, so the first run is slow and later calls are fast. Keep one REPL open.

Functions have many methods; the one chosen depends on the types of all arguments (multiple dispatch).

Arrays are 1-indexed, column-major, and a function ending in ! mutates its first argument.

COMMAND LINE

`julia`
Start the interactive REPL

`julia <file.jl> <args>`
Run a script; args land in ARGS

`julia -e '<expr>'`
Evaluate an expression and exit

`julia -E '<expr>'`
Evaluate and print the result

`julia --project=<dir>`
Activate the environment in a folder
--project=. or plain --project searches parent folders for Project.toml.

`julia -t <n|auto>`
Set the number of threads
Also JULIA_NUM_THREADS; check with Threads.nthreads().

`julia -p <n|auto>`
Add local worker processes

`julia -i <file.jl>`
Run a file, then stay in the REPL

`julia -L <file.jl>`
Load a file before the REPL starts

`julia --startup-file=no`
Skip ~/.julia/config/startup.jl

REPL

`?`
Help mode: docs for a name

`]`
Package mode

`;`
Shell mode: run a shell command

`Backspace`
Leave a special mode

`Ctrl+D / exit()`
Quit Julia

`Ctrl+C`
Interrupt running code

`ans`
Result of the last expression

REPL (cont.)

`Ctrl+R`
Search history

`methods(<f>)`
List methods of a function

`@time <expr>`
Time an expression and show allocations

`Revise`
Reload edited code without restarting
Install the Revise package and run using Revise first.

`include("<file.jl>")`
Evaluate a file in this session

`using <Pkg>`
Load a package's exports

`import <Pkg>`
Load a package, names stay qualified

PACKAGES

`add <pkg>`
Add a package to the environment

`rm <pkg>`
Remove a package

`status`
Show the environment (alias st)

`up`
Update packages

`activate <dir>`
Switch to the environment in a folder

`activate --temp`
Switch to a temporary environment

`instantiate`
Install what Manifest.toml lists
Run after cloning a project.

`test`
Run the project's tests

`precompile`
Precompile dependencies now

`dev <path or name>`
Work on a local copy of a package

`st -o`
Show outdated packages

`add <pkg>@<ver>`
Add a specific version

`generate <Name>`
Create a minimal package

`using Pkg; Pkg.add("<pkg>")`
Pkg functions from code

Commit Project.toml. Commit Manifest.toml too for apps, not for libraries.

BASICS

`x = 3; y = x^2`
Assign and exponentiate

`println("x = $x")`
Print with interpolation

`"a" * "b"`
Concatenate strings
+ does not work on strings.

`if a < b ... elseif ... else ... end`
Conditional

`for i in 1:10 ... end`
Loop over a range

`while <cond> ... end`
Loop while true

`function f(x, y=2; k=1) ... end`
Function with default and keyword args

`f(x) = x^2`
Short function form

`x → x^2`
Anonymous function

`(1, "a")`
Tuple, immutable

`Dict("a" ⇒ 1)`
Dictionary

`d["a"]`
Index a dictionary

`get(d, k, <default>)`
Lookup with a fallback

`1:2:9`
Range with step

`typeof(x)`
Type of a value

`try ... catch e ... finally ... end`
Handle errors

ARRAYS

`[1, 2, 3]`
Vector literal

`[1 2; 3 4]`
Matrix literal

`A[i, j]`
Element at row i, column j

`A[2:end, :]`
Slice; end is the last index

`push!(v, x)`
Append to a vector

`length(v) / size(A)`
Element count / dimensions

`zeros(n, m) / ones(n)`
Arrays of 0 or 1

`[f(x) for x in v if <cond>]`
Comprehension

`f.(v)`
Broadcast f over elements

ARRAYS (cont.)

`v .+ 1`
Elementwise operator

`map(f, v) / filter(f, v)`
Apply / keep matching

`v[v .> 0]`
Select by boolean mask

TYPES & DISPATCH

`struct P x::Float64; y::Float64 end`
Immutable composite type

`mutable struct C n::Int end`
Mutable composite type

`area(s::Circle) = pi * s.r^2`
Method for one type

`abstract type Shape end`
Abstract supertype

`struct Circle <: Shape ... end`
Declare a subtype

MACROS & TOOLING

`@time f(x)`
Time and allocation report

`@btime f($x)`
Accurate benchmark (BenchmarkTools)
Interpolate globals with $ to avoid skewed results.

RECIPES

New project

```
mkdir myproj && cd myproj
julia --project=.
# ] then: add DataFrames
# julia> using DataFrames
```

Clone and run

```
git clone <url> && cd <repo>
julia --project=. -e 'using
Pkg; Pkg.instantiate()'
julia --project=. src/main.jl
```

Script with an entry point

```
# main.jl
function (@main)(args)
    println("got
$(length(args)) args")
    return 0
end
```

Threaded sum

```
# julia -t 4 sum.jl
using Base.Threads
v = rand(10^6)
s = Atomic{Float64}(0)
@threads for x in v
    atomic_add!(s, x)
end
```

HELP

`?<name>`
Docs for a function or type