# Modules and packages

## `using` vs `import`, package layout, Pkg REPL

### 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`:

```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:

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

## Environments

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

```julia
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
```