Types

Abstract/concrete, parametric, unions, invariance

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).