Structs

Immutability by default, constructors, `@kwdef`

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.