Multiple dispatch

The organising principle. How it differs from OO and from Elixir clauses

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

```
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:

```
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:

```
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.

```
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}`:

```
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:

```
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.