Interfaces

Informal protocols: iterate, getindex, show

There is no `interface` keyword, no `implements`, and no compile-time check. An interface in Julia is a documented set of methods; you implement it by defining those methods, and in exchange a large amount of generic code starts working on your type.

This is looser than Go's structural typing (which at least verifies at compile time) and much looser than a Java interface. The compensation is that you can retrofit an interface onto a type you don't own.

Iteration

Define `iterate` and you get `for`, `collect`, `map`, `sum`, comprehensions, `zip`, destructuring, and everything else.

```
struct Countdown
    n::Int
end

# first call: no state argument. Return (value, state) or nothing.
Base.iterate(c::Countdown) = c.n > 0 ? (c.n, c.n - 1) : nothing
# subsequent calls
Base.iterate(::Countdown, state) = state > 0 ? (state, state - 1) : nothing

# strongly recommended extras
Base.length(c::Countdown) = c.n                 # enables collect to preallocate
Base.eltype(::Type{Countdown}) = Int            # enables a typed result array

collect(Countdown(3))          # [3, 2, 1]
sum(Countdown(3))              # 6
[x^2 for x in Countdown(3)]
```

If length is unknown, say so instead of defining `length`:

```
Base.IteratorSize(::Type{MyStream}) = Base.SizeUnknown()
Base.IteratorEltype(::Type{MyStream}) = Base.EltypeUnknown()
```

Indexing

```
Base.getindex(v::MyVec, i::Int) = v.data[i]
Base.setindex!(v::MyVec, x, i::Int) = (v.data[i] = x)
Base.firstindex(v::MyVec) = 1
Base.lastindex(v::MyVec) = length(v.data)     # makes `v[end]` work
```

AbstractArray

The highest-leverage interface in the language. Implement four methods and you inherit the entire array ecosystem — broadcasting, linear algebra, printing, slicing, reductions.

```
struct Squares <: AbstractVector{Int}
    n::Int
end

Base.size(s::Squares) = (s.n,)
Base.getindex(s::Squares, i::Int) = i^2
Base.IndexStyle(::Type{<:Squares}) = IndexLinear()
# and setindex! if it's mutable

s = Squares(5)
s[3]           # 9
sum(s)         # 55
s .+ 1         # works
s[2:4]         # works
println(s)     # prints as a 5-element vector, with the right header
```

For multidimensional, `size` returns a longer tuple, `getindex(A, i::Int, j::Int)`, and `IndexCartesian()`.

Display

```
Base.show(io::IO, x::MyType) = print(io, "MyType(", x.a, ")")          # compact
Base.show(io::IO, ::MIME"text/plain", x::MyType) = ...                 # REPL, verbose
Base.show(io::IO, ::MIME"text/html", x::MyType) = ...                  # notebooks
```

Equality, hashing, ordering

```
Base.:(==)(a::T, b::T) = ...
Base.isequal(a::T, b::T) = ...
Base.hash(x::T, h::UInt) = hash(x.field, hash(:T, h))     # must match ==
Base.isless(a::T, b::T) = ...                          # gives you sort, maximum, extrema
```

Define `hash` whenever you define ==, or dictionaries silently misbehave.

Numbers

To make your type behave arithmetically:

```
Base.:+(a::Money, b::Money) = Money(a.cents + b.cents)
Base.:-(a::Money, b::Money) = Money(a.cents - b.cents)
Base.:*(a::Money, k::Real)  = Money(round(Int, a.cents * k))
Base.zero(::Type{Money}) = Money(0)
Base.one(::Type{Money}) = Money(100)
Base.convert(::Type{Money}, x::Real) = Money(round(Int, 100x))
Base.promote_rule(::Type{Money}, ::Type{<:Integer}) = Money
```

`zero` and `one` matter because generic code (`sum`, `prod`, matrix operations) calls them to get identity elements.

Broadcasting

By default a custom type broadcasts as a container if it's iterable, or errors. To make it behave as a scalar:

```
Base.broadcastable(x::MyConfig) = Ref(x)
```

For a custom array type that should produce its own type from broadcasts, implement `BroadcastStyle` — worth reading the manual section rather than guessing.

Traits, for interfaces that cut across the type tree

You get one supertype. When a property applies to types scattered across the hierarchy, use a trait: a function returning a singleton, dispatched on in a helper.

```
abstract type Storage end
struct InMemory <: Storage end
struct OnDisk <: Storage end

storage(::Type) = InMemory()               # default
storage(::Type{<:BigTable}) = OnDisk()

process(x) = _process(storage(typeof(x)), x)
_process(::InMemory, x) = ...
_process(::OnDisk, x) = ...
```

`Base` uses this everywhere: `IndexStyle`, `IteratorSize`, `OrderStyle`, `ArithmeticStyle`.

Interface checklist for a new container type

Ask what you actually need. A rough ladder:

1. `show` — always.
2. == and `hash` — if instances are compared or used as keys.
3. `iterate`, `length`, `eltype` — if it's a collection.
4. `getindex`, `setindex!`, `firstindex`, `lastindex` — if it's indexable.
5. `size`, `getindex`, `IndexStyle`, `<: AbstractArray` — if it's array-like. Do this rather than 3-4 if it applies; you get far more for the same effort.
6. `isless` — if it's orderable.

7. `broadcastable` — if it should broadcast as a scalar.

Checking your work

There's no compiler check, so use tests. `Aqua.jl` catches structural problems (ambiguities, unbound type parameters, piracy). For the `AbstractArray` interface specifically, `ArrayInterfaceTests`-style packages and `JET.jl` help. Otherwise: write a testset that exercises the generic functions you claim to support.