

nothing, missing, NaN

Three different holes, three different behaviours

Julia has three different “absent value” concepts and they are not interchangeable. Coming from a language with one null (or with nil/None), this is the distinction to internalise early.

Value	Type	Means	Arithmetic	Test
nothing	Nothing	no value; void return	1 + nothing is a MethodError	isnothing(x), x === nothing
missing	Missing	value exists but is unknown (statistics)	1 + missing is missing — propagates	ismissing(x)
NaN	Float64	undefined float result	propagates within floats	isnan(x)
undef	UndefInitializer	“don’t initialise this memory”	—	only in constructors

nothing

The absence of a value. Returned by functions that found nothing, and by functions called for their side effects.

```
findfirst(==(99), [1,2,3])    # nothing
match(r"z", "abc")            # nothing
get(dict, :missing_key, nothing)
println("x")                  # returns nothing
```

Working with it:

```
x = findfirst(==(3), v)
if isnothing(x)
    ...
end
x === nothing                # equivalent, and the fastest check

something(a, b, 0)           # first argument that isn't nothing; throws if all are
@something a b 0              # short-circuiting version (1.7+)

# the idiomatic optional-return signature
function lookup(k)::Union{Int, Nothing}
    haskey(d, k) ? d[k] : nothing
end
```

Union{T, Nothing} is the closest thing to Option/Maybe, and the compiler optimises small unions well, so there’s no performance argument against it. What Julia does *not* have is the { :ok, value } | { :error, reason } convention from Elixir — the ecosystem either returns nothing or throws. If you want tagged results, ResultTypes.jl or Try.jl exist but are not idiomatic.

trap: nothing displays as nothing at the REPL. A function that silently returns nothing looks like it returned successfully. @show f(x) or typeof(f(x)) when confused.

missing

Statistical missingness, modelled on SQL NULL and R’s NA. It **propagates** through almost every operation.

```
1 + missing                  # missing
missing == 1                  # missing (NOT false!)
missing == missing            # missing
isequal(missing, missing)    # true — this is why isequal exists
missing > 1                   # missing
```

```
if missing           # TypeError: non-boolean used in boolean context
end
```

Three-valued logic for the short-circuiting operators:

```
true  || missing    # true
false || missing    # missing
false && missing     # false
true  && missing     # missing
```

Handling it:

```
ismissing(x)
coalesce(x, 0)           # first non-missing
skipmissing(v)          # lazy iterator that drops them
sum(skipmissing(v)); mean(skipmissing(v))
collect(skipmissing(v))
replace(v, missing => 0)
filter(!ismissing, v)
disallowmissing(v)       # (Missings.jl) narrow the element type back

eltype([1, missing, 3]) # Union{Missing, Int64}
```

Column types in DataFrames are the main place you'll meet it. A column that might have gaps is `Vector{Union{Missing, Float64}}`, and every aggregation you run against it needs `skipmissing` or it returns missing.

Use missing for data. Use nothing for control flow. Mixing them up produces code that either propagates when it should error or errors when it should propagate.

NaN

An IEEE-754 float, not a separate type.

```
0/0           # NaN
NaN == NaN    # false      <- the classic
NaN === NaN   # true       (identical bits)
isequal(NaN, NaN) # true
isnan(NaN)      # true
sort([3, NaN, 1]) # NaN sorts last, because sort uses isless
maximum([1, NaN]) # NaN – propagates
maximum(filter(!isnan, v))
NaN in [NaN]     # true (uses isequal)
```

Inf is separate and better behaved: `1/0 == Inf`, `isinf`, `isfinite`. `isfinite(x)` is the check that excludes both NaN and Inf.

Choosing, in practice

```
# a config value that may be unset
port::Union{Int, Nothing}

# a sensor reading that failed to arrive
temperature::Union{Float64, Missing}

# a computation that has no defined answer
result = log(-1.0) # DomainError, actually – Julia is strict here
result = log(complex(-1.0)) # if you want the complex branch
```

Note that Julia throws `DomainError` for `sqrt(-1.0)` and `log(-1.0)` rather than returning NaN — it will not silently promote to complex. Pass a complex input if that's what you want.

Related: undefined and uninitialised

```
v = Vector{Float64}(undef, 3) # garbage values, fastest allocation
v = Vector{String}(undef, 3)  # #undef references – accessing throws UndefRefError
```

```
isassigned(v, 1)
```

```
x                                #.UndefVarError if never assigned  
@isdefined x
```

undef is not a value you can store; it's a marker telling the constructor to skip initialisation. For arrays of pointers (String, Any, mutable structs), the slots really are undefined and reading them throws.