Introduction to Julia

Cédric Simal

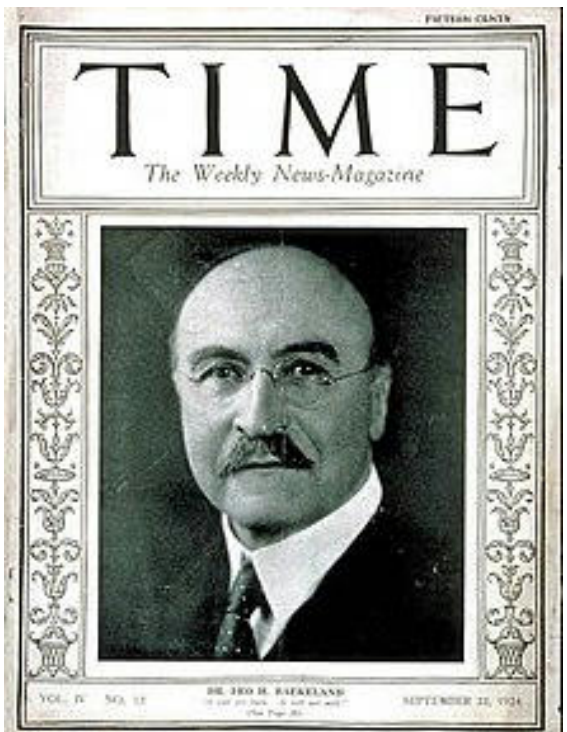
cedric.simal@unamur.be

CéCI Training Sessions

14/11/2023

Science Anniversaries of today

Apollo 12 Launch (1969)



Birth of Leo Baekeland (1863)

image source: Wikipedia

Outline

1. Installing Julia
2. Elevator pitch (while you're installing)
3. The Julia REPL
4. Basics of Julia (`basic.ipynb`)
5. Multiple Dispatch (`types.ipynb`)
6. Package Management

Installing Julia (via Juliaup)

https://github.com/JuliaLang/juliaup

Windows

```
winget install julia -s msstore
```

Mac/Linux

```
curl -fsSL https://install.julialang.
org | sh
```

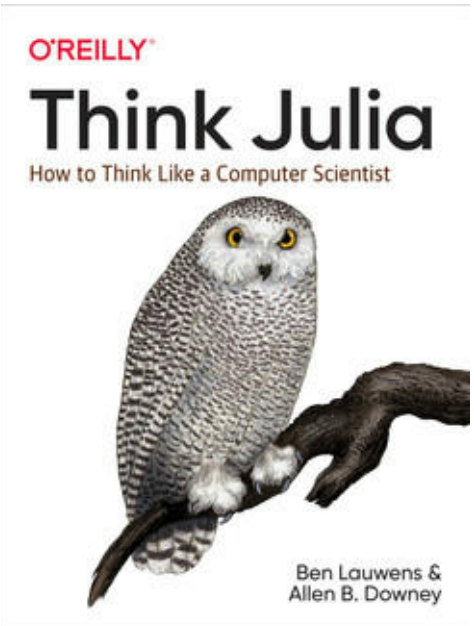
Then

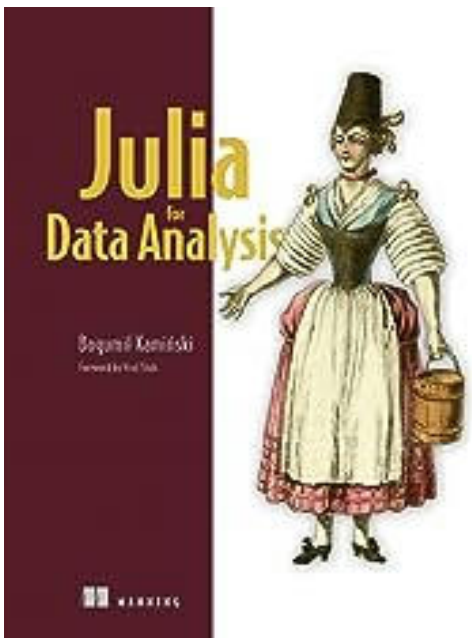
```
juliaup update
```

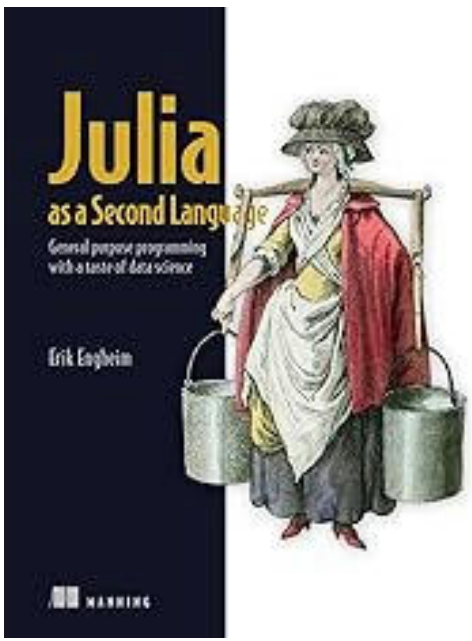
Follow along!

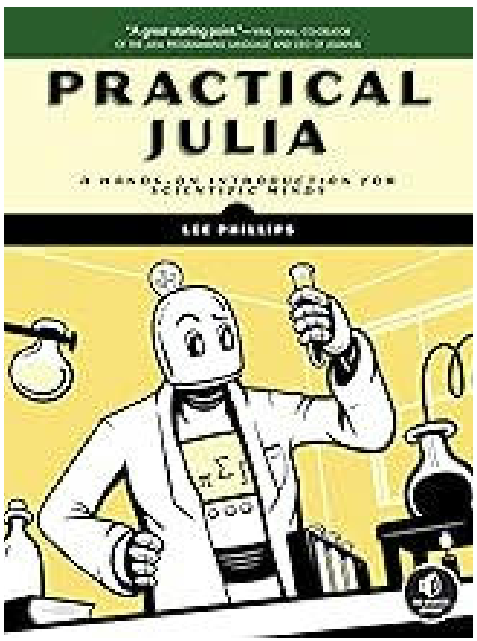
https://github.com/csimal/CeCI-Intro-Julia

Book Recommendations





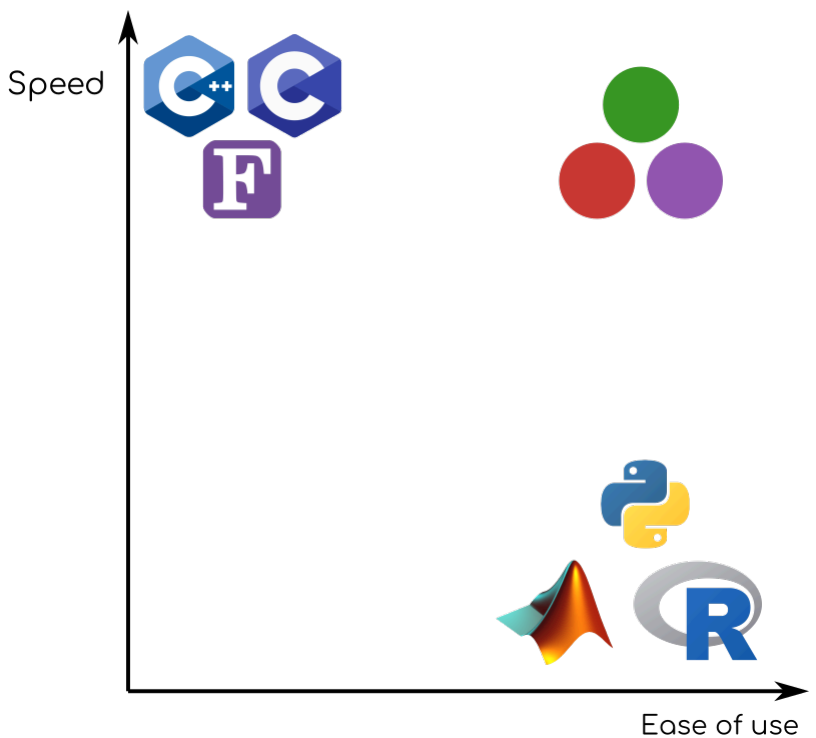




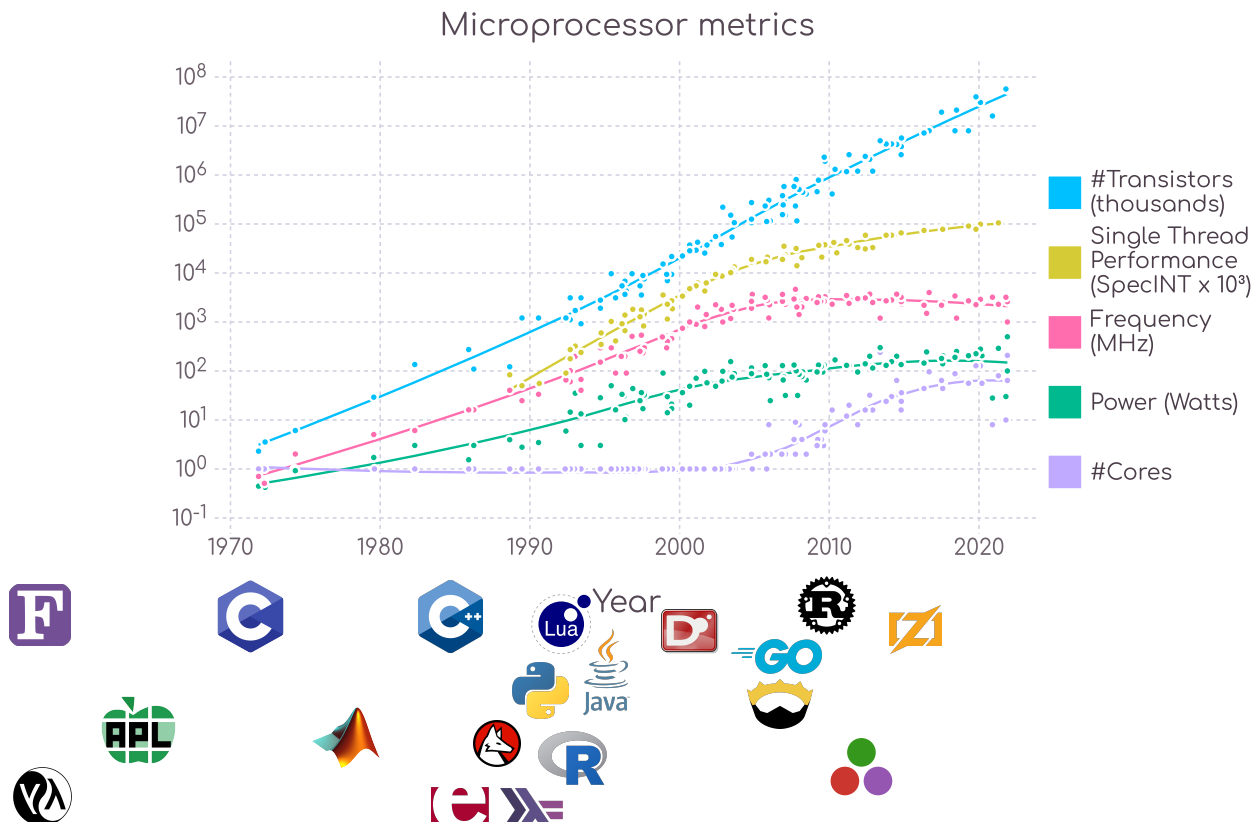
https://benlauwens.github.io/ThinkJulia.jl/latest/book

Modern Problems in Scientific Computing

The two languages problem



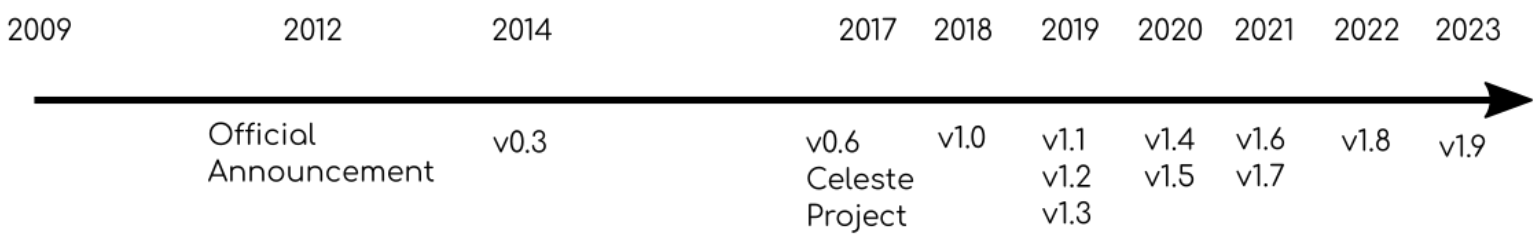
The rise of parallel computing



data: https://github.com/karlrupp/microprocessor-trend-data

Meet Julia

A short history



Why We Created Julia

14 February 2012 | **Jeff Bezanson Stefan Karpinski Viral B. Shah Alan Edelman**

Jeff Bezanson Stefan Karpinski Viral B. Shah Alan Edelman
In short, because we are greedy.

We are power Matlab users. Some of us are Lisp hackers. Some are Pythonistas, others Rubyists, still others Perl hackers. There are those of us who used Mathematica before we could grow facial hair. There are those who still can't grow facial hair. We've generated more R plots than any sane person should. C is our desert island programming language.

Julia Joins Petaflop Club

September 12, 2017

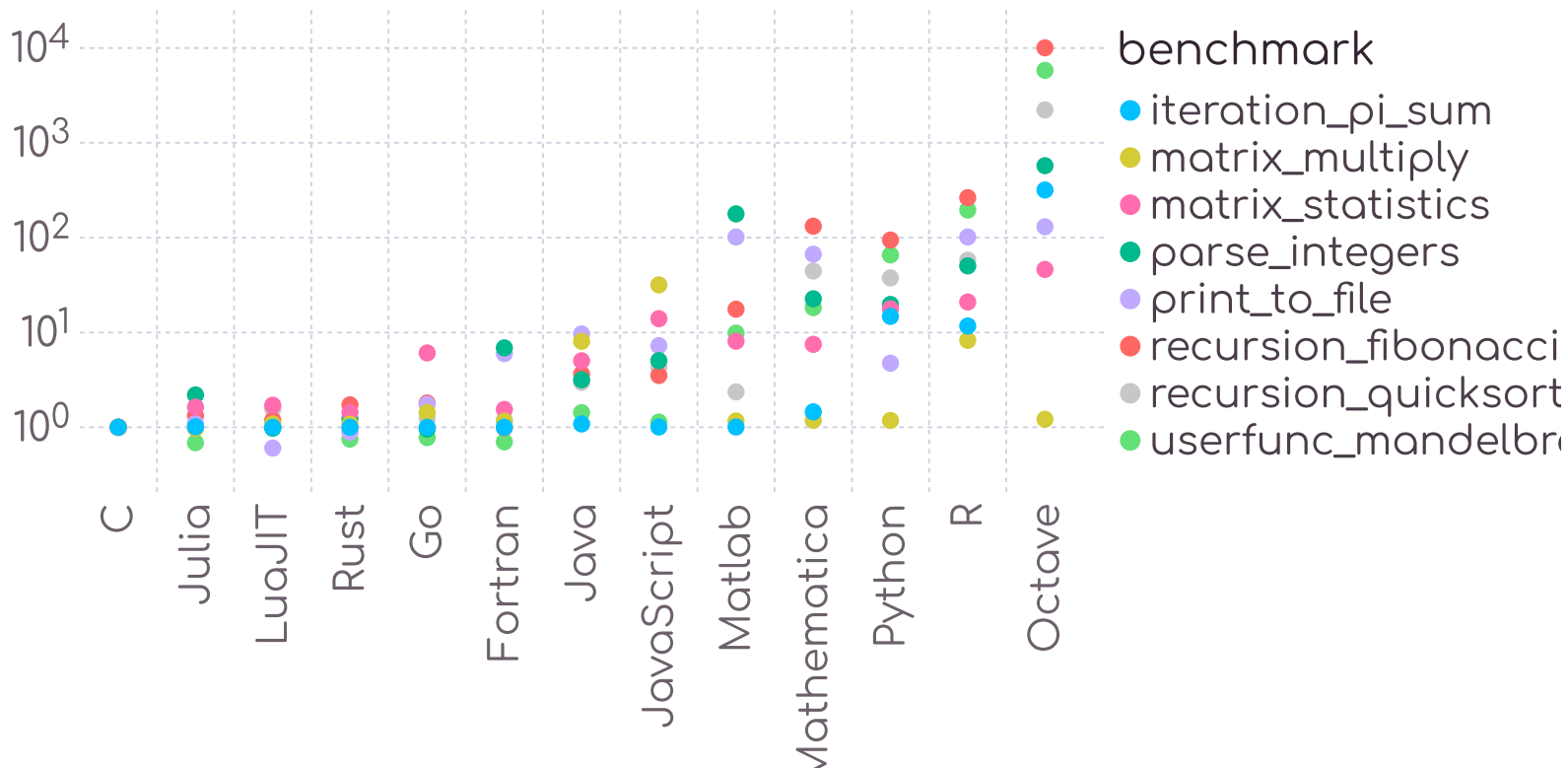
BERKELEY, Calif., Sept. 12, 2017 — Julia has joined the rarefied ranks of computing languages that have achieved peak performance exceeding one petaflop per second – the so-called 'Petaflop Club.'

The Julia application that achieved this milestone is called Celeste. It was developed by a team of astronomers, physicists, computer engineers and statisticians from UC Berkeley, Lawrence Berkeley National Laboratory, National Energy Research Scientific Computing Center (NERSC), Intel, Julia Computing and the Julia Lab at MIT.

https://julialang.org/blog/2012/02/why-we-created-julia/

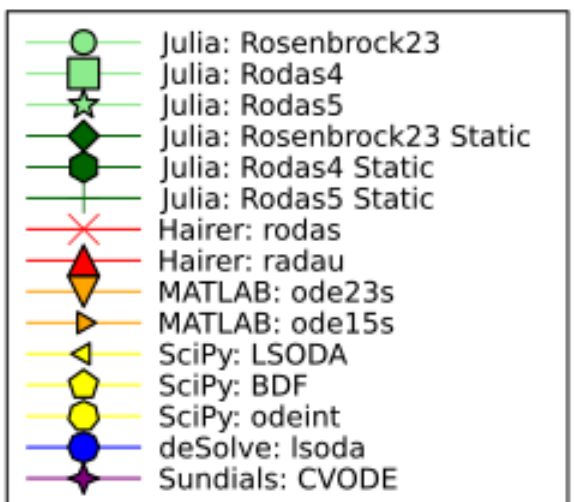
https://www.hpcwire.com/off-the-wire/julia-joins-petaflop-club/

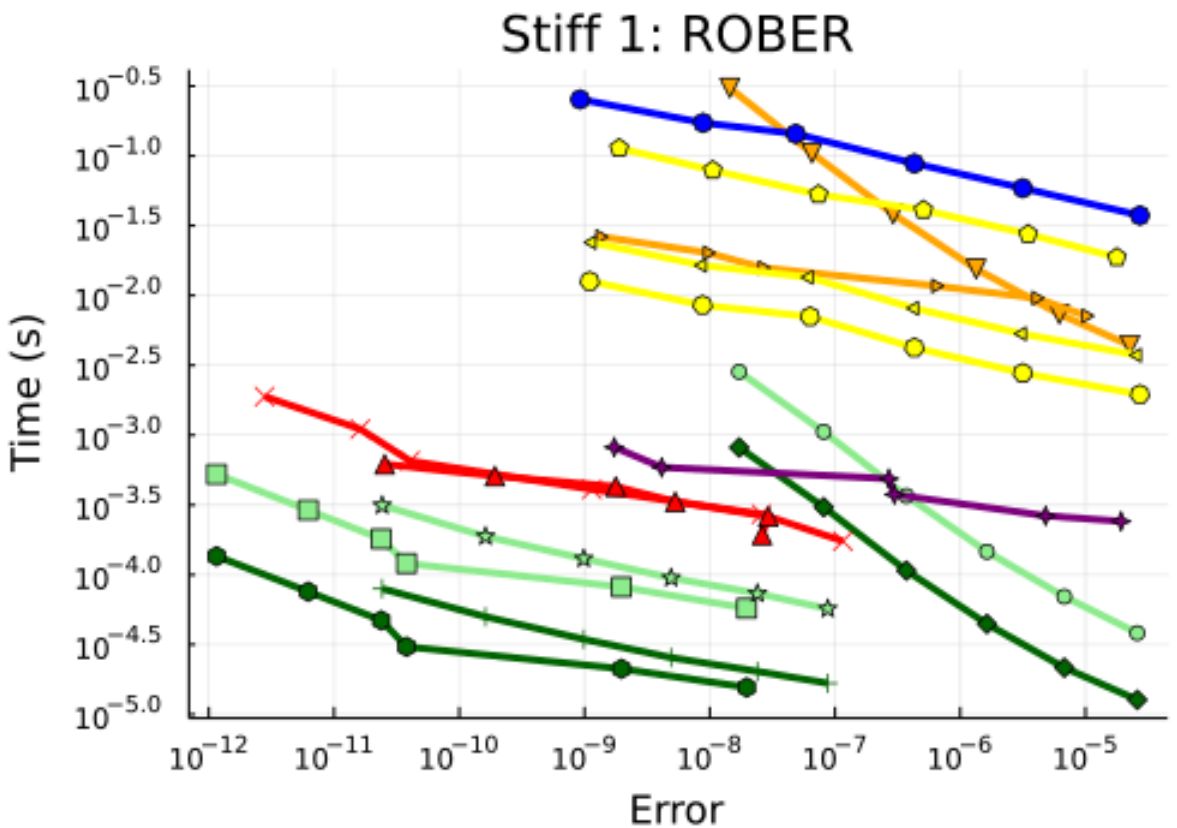
Julia is fast



https://julialang.org/benchmarks/

Julia is fast (2)





https://benchmarks.sciml.ai/stable/MultiLanguage/ode_wrapper_packages/

Why you might want to use Julia

- Open Source (MIT license)
  - Free as in Free Beer *and* Free Speech
  - Faster development than proprietary languages
- Made for numerical computing
- Amazing Interop with C/C++, Fortran, Python, Matlab...
- Unicode
  - Code looks like math
- Secretly a LISP

Why you might NOT want to use Julia

If you want to evangelize a tool or process or technology, you should be able to answer two questions:

1. What negative thing do you have to deal with?
2. What positive thing do you have to give up?

My personal takes for Julia:

1. Time to first X; Poorly documented/Undocumented features.
2. No formal Interfaces/Traits/Typeclasses. Types don't help with correctness.

https://hillelwayne.com/hate-your-tools/

Why you might NOT want to use Julia

- Open Source
  - Documentation can be lackluster
  - Projects with only one maintainer
- Time to first X
  - Parts of your program have to be recompiled on startup
- Error messages are not beginner-friendly
- Correctness bugs
- Tiny compared to Python
- Array indices start at 1

More criticism: https://yuri.is/not-julia/

The Julia REPL

Basics of Julia

Variables, comments and statements

Variable declaration

```
foo = 1
```

Multiple declarations

```
a, b, c = 1, 2, 3
# a = 1; b = 2; c = 3
```

Multiline comment

```
#= This comment
spans multiple
lines
=#
```

By default, one statement per line

```
a = 1
a += 1 # a = a + 1
a *= 2 # a = a * 2

# compact version
a = 1; a += 1; a *= 2

# block version
begin
    a = 1
    a += 1
    a *= 2
end
```

Elementary Types

Integers

```
a = 1
```

Floating point numbers (double)

```
b = 0.2
```

Booleans

```
e = true; f = false
```

Strings and characters

```
c = 'c'
d = "Hello, world\n" # NB C-style character literals work
```

Arrays, tuples and Dicts

```
x = [1, 2, 3]
y = ("Waffle", 2)
z = Dict(
    "Frites" ⇒ 2.0,
    "Fricandelle" ⇒ 1.5,
    "Sauce Andalouse" ⇒ 0.3,
)
```

Conditionals

```
# get a random number between -30 and
150
water_temperature = rand(-30:150)
if water_temperature < 0
    print("Ice")
elseif water_temperature < 100
    print("Liquid Water")
else
    print("Steam")
end
```

Note that an if block is an expression that returns a value:

```
age = rand(1:40)
status = if age < 18
    "minor"
else
    "adult"
end
```

Ternary operator

```
status = age < 18 ? "minor" : "adult"
```

Loops

Python style for-loops with iterators

```
k = 0
for i in 1:10
    k += i
end

n = 10
while n != 1
    n = iseven(n) ? div(n,2) : 3*n+1
end
```

Special Syntax for nested for loops

```
for i in 1:5
    for j in 1:3
        print(i+j)
    end
end
# This can be rewritten as
for i in 1:5, j in 1:3
    print(i+j)
end
```

Arrays

```
# a 1 dimensional vector with 10 elements of type 'double'
a = Vector{Float64}(undef, 10)
# equivalent to previous line
b = Array{Float64,1}(undef, 10)
# A 2 dimensional array of integers with 3 rows and 4 columns
c = Array{Int,2}(undef, 3,4)
d = Array{Int,2}(undef, (3,4))
```

Special constructors for arrays of zeroes/ones/arbitrary value

```
e = zeros(3,4)
f = ones(3)
g = fill("Hello", 4)
```

Array Constructors

Explicit enumeration

```
a = [1,2,3,4] # 1D array (column vector)
b = [1 2 3 4] # 1 x 4 matrix (row vector)
c = [1 2; 3 4; 5 6] # 3 x 2 matrix
d = [1; 2;; 3; 4;;; 5; 6;; 7; 8] # 2 x 2 x 2 3D array
```

Ranges

```
i = 1:10 # integers from 1 to 10
x = 0.0:0.01:1.0 # from 0.0 to 1.0 by steps of 0.1
y = LinRange(0.0,1.0,100)
```

Array Comprehensions

```
# Apply arbitrary function to a collection
a = [i^2 for i in 1:10]
# Use multiple indices to generate higher dimensional arrays
b = [i+j for i in 1:5, j in 1:5]
# Filter elements using a condition
c = [i/(i^2) for i in 1:10 if iseven(i)]
# special syntax for functions taking a collection
p = 4 * sum((-1)^k / (2k+1) for k in 0:10000)
```

Array indexing

```
# default arrays start at 1
a[1]
# use an array of indices
a[[1,3,5]]
# ranges work too
a[1:5]
a[begin] # first element
a[end] # last element
```

Iterate over an arbitrary array

```
for i in eachindex(a)
    print(a[i])
end
```

Array slicing

```
b = rand(3,4) # 3 x 4 matrix of random numbers
b[1,:] # first row of b
b[:,2] # second column of b
```

Iterate over array dimension

```
for j in axes(b,2)
    print(b[:,j])
end
```

Broadcasting

Function calls work as usual

```
cos(π/2) # NB. use "\pi<TAB>" for unicode π
```

Special syntax for "vectorizing" function calls

```
xs = LinRange(0.0,2π,100)
cos.(xs)
# equivalent to
[cos(x) for x in xs]
# It also applies to infix operators
xs .* cos.(xs)
```

Modifying arrays

```
a = zeros(5)
a .= [1,2,3,4,5] # NB. without the dot,
this would create a new array
a .+= 1.0
a .+= [6,7,8,9,10]
```

Multiple broadcasted calls

```
a .= a.^3 .- a.^2 .+ a .- 1
# equivalent to
for i in eachindex(a)
    a[i] = a[i]^3 - a[i]^2 + a[i] - 1
end
# short-hand
@. a = a^3 - a^2 + a - 1
```

Functions

```
function collatz(n)
    if iseven(n)
        return div(n,2)
    else
        return 3n+1
    end
end
# More compact version
function collatz(n)
    return iseven(n) ? div(n,2) : 3n+1
end
# implicit return
function collatz(n)
    iseven(n) ? div(n,2) : 3n+1
end
```

Inline definition

```
square(x) = x^2
```

Lambda expressions/Anonymous functions

```
square = x -> x^2
square = function(x)
    x^2
end
```

Default values and keyword arguments

```
function say_hello(x = "World"; hello_str="Hello", end_str="!")
    # NB. '*' concatenates strings
    println(hello_str * ", " * x * end_str)
end
```

```
julia> say_hello()
Hello, World!

julia> say_hello("Alice")
Hello, Alice!

julia> say_hello("Alice", hello_str="Bonjour", end_str="?")
Bonjour, Alice?
```

The Type System

Elementary Types

Type annotations

```
a::Int = 1
b::Float64 = 0.2
c::Vector{Int} = [1,2,3]
```

Numeric types

- Integers: `Int` (`Int64`), `Int32`, ...
- Unsigned Ints: `UInt64`, `UInt32`, ...
- Floats: `Float64`, `Float32`, ...
- Complex Numbers: `ComplexF64`, `Complex{Float64}`, ...

Collections with generic element types

- N-dimensional Array of eltype `T`:

  `Array{T,N}`
- Dictionary with keys of type `K` and values of type `V`:

  `Dict{K,V}`

Creating new types

```
struct Student
    name::String
    age::Int
    section::String
    year::Int
end
```

Default constructor

```
alice = Student("Alice", 22,
"Chemistry", 4)
# accessing fields
alice.section
```

Mutable structs

```
mutable struct Lightbulb
    age::Int
    wattage::Float64
    turned_on::Bool
end
```

Type parameters

```
struct Point2D{T}
    x::T
    y::T
end
```

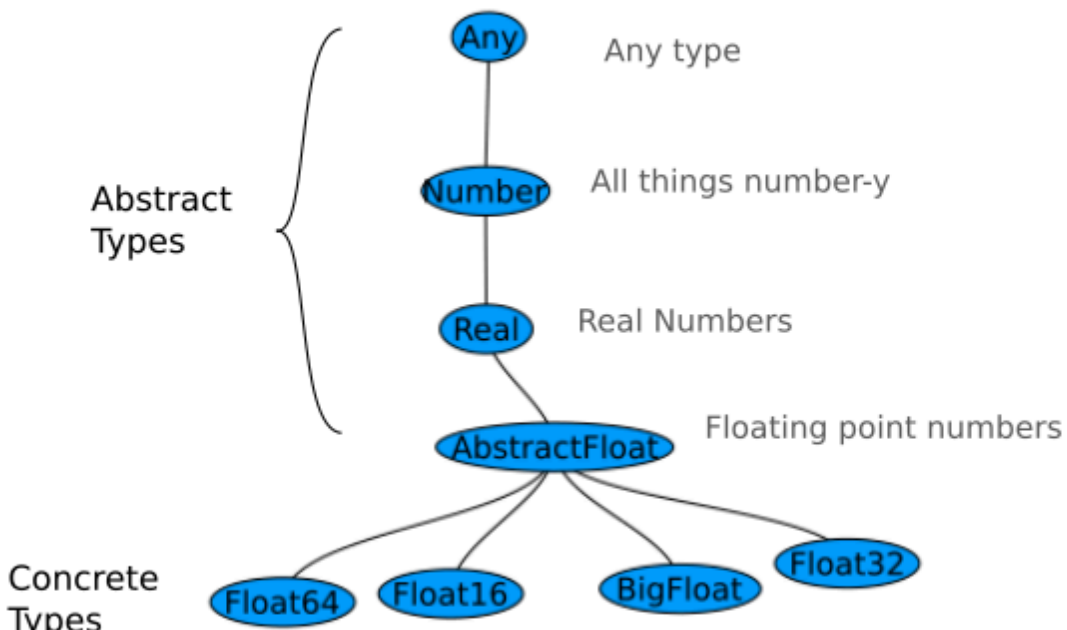
Subtyping

Subtype relation

```
Int <: Integer
```

Defining a type as a subtype

```
abstract type Animal end
abstract type Mammal <:
Animal end
struct Tiger <: Mammal end
```



First-Class Citizens

In a given programming language, first-class citizens are objects that

1. can be used as arguments of functions
2. can be returned by functions
3. can be assigned to variables
4. can be tested for equality

In Julia, the following objects are first-class citizens

- Functions (⇒ Functional Programming)
- Types (via the `Type{T}` type)
- Julia expressions (⇒ Metaprograming)

Multiple Dispatch

Functions can be defined with specific input types

```
function collatz(n::Int)
    iseven(n) ? div(n,2) : 3n+1
end
```

Optional return type

```
function collatz(n::Int)::Int
    iseven(n) ? div(n,2) : 3n+1
end
```

Multiple definitions with different types (Methods)

```
foo(n::Int) = "$n is an Int"
foo(x::Float64) = "$x is a Float"
foo(s::String) = "$s is a String"
```

NB. a function is just a name with multiple methods!

Multiple Dispatch (2)

Let's model chemical reactions

```
abstract type Chemical end
struct H2 <: Chemical
struct H2O <: Chemical end
struct O2 <: Chemical end
struct SO3 <: Chemical end
...
```

- `H2 + O2` → `H2O` (+ explosion)
- `H2O + SO3` → `H2SO4`
- otherwise nothing happens

Multiple Dispatch (3)

Don't do this!

```
function mix(x,y)
    if (x isa H2 && y isa O2)
        return "Boom!"
    elseif (x isa H2O && y isa SO3)
        return "Sulfuric Acid!"
    else
        return "Nothing happens."
    end
end
```

Use multiple dispatch

```
mix(::H2, ::O2) = "Boom!"
mix(::O2, ::H2) = "Boom!"
mix(::H2O, ::SO3) = "Sulfuric Acid!"
mix(::SO3, ::H2O) = "Sulfuric Acid!"
mix(::Chemical, ::Chemical) = "Nothing
Happens"
```

For another example: https://www.moll.dev/projects/effective-multi-dispatch/

Multiple Dispatch: Dual Numbers

```
struct Dual <: Number
    x::Float64
    y::Float64
end

convert(::Type{Dual}, x::Real) =
Dual(x, zero(x))
promote_rule(
    ::Type{Dual},
    ::Type{<:Number}) = Dual
show(io::IO, d::Dual) = print(io, d.x,
" + ", d.y," ϵ")
```

Arithmetic operations

```
+(a::Dual, b::Dual) = Dual(a.x + b.x,
a.y + b.y)
-(a::Dual, b::Dual) = Dual(a.x - b.x,
a.y - b.y)
*(a::Dual, b::Dual) = Dual(a.x * b.x,
a.y*b.x + a.x*b.y)
/(a::Dual, b::Dual) = Dual(a.x/b.x,
(a.y*b.x - a.x*b.y)/(b.x^2))
```

Package Management

Noteworthy packages (for science)

ODEs/Dynamical Systems

- `DifferentialEquations.jl`
- `ModelingToolkit.jl`
- `Catalyst.jl` (chemical equations)

Optimization

- `JuMP`
- `Optim.jl`

Misc

- `Unitful.jl`
- `Measurements.jl`

Statistics/Probability

- `Distributions.jl`
- `Statistics.jl`
- `GLM.jl` (Linear Regression)
- `Turing.jl`, `Gen.jl`, `Stan.jl` (Bayesian models)
- `Tidier.jl` (Tidyverse in Julia)

Plotting

- `Plots.jl`
- `Makie.jl`
- `Gadfly.jl`