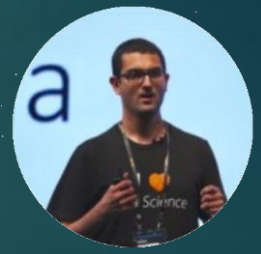
Programming Language

AHMET BİROL ÇAVDAR
21920365

Very Brief History[1]

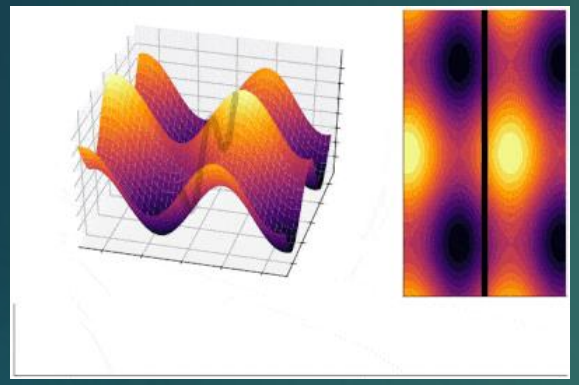
- Work on Julia was started in 2009, by Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman, who set out to create a free language that was both high-level and fast.
- In an interview with InfoWorld in April 2012, Karpinski said of the name "Julia": "There's no good reason, really. It just seemed like a pretty name". Bezanson said he chose the name on the recommendation of a friend.
- Since the 2012 launch, the Julia community has grown, and "Julia has been downloaded by users at more than 10,000 companies", with over 20,000,000 downloads as of September 2020, up from 9 million a year prior (and is used at more than 1,500 universities).
- The Official Julia Docker images, at Docker Hub, have seen over 4,000,000 downloads as of January 2019.
- The JuliaCon academic conference for Julia users and developers has been held annually since 2014.
- Version 0.3 was released in August 2014, version 0.4 in October 2015, version 0.5 in October 2016, and version 0.6 in June 2017.
- Both Julia 0.7 and version 1.0 were released on 8 August 2018.
- Julia 1.5 has been released in August 2020 and latest stable release v1.5.3 has been released November 2020.



[1] Julia (programming language), https://en.wikipedia.org/wiki/Julia_(programming_language)

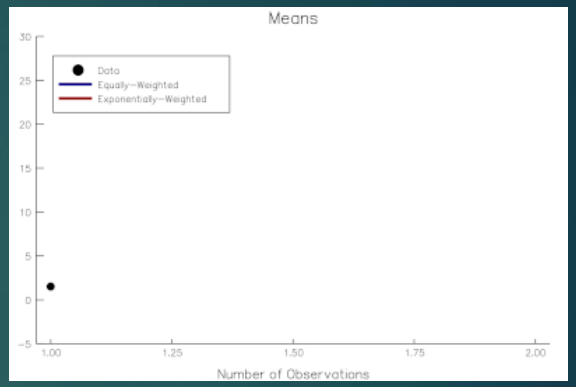
Application Domain [1]

Visualization

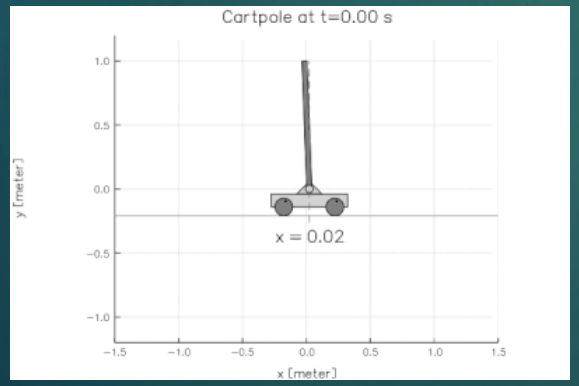


General Purpose

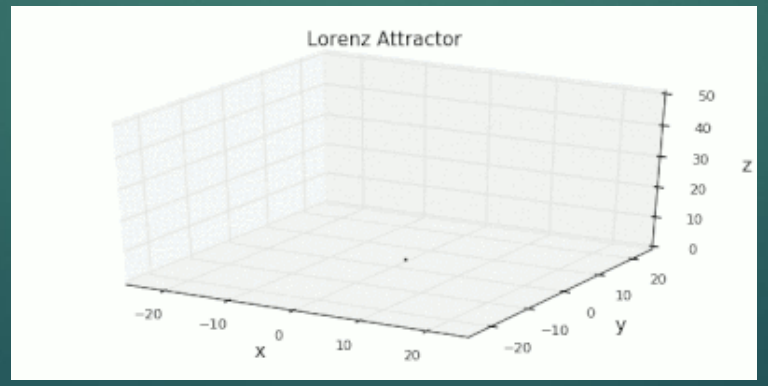
Data Science



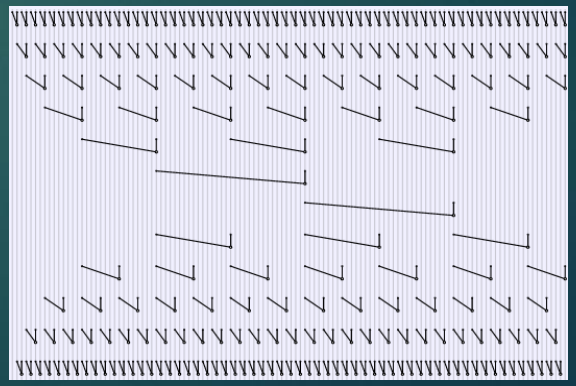
Machine Learning



Scientific Domains



Parallel Computing



[2] The Julia Programming Language, https://julialang.org/

Language Evaluation - Readability

- One of the basic design philosophies of Julia is keep the core (Base) ultra-lean, develop extra functionalities as packages [3]. This principle makes the language simpler and more readable. For example, Julia does not have **switch** statement.
- Having use of self-descriptive constructs and meaningful keywords increases readability. begin, end, for, while, global, return, break, continue, etc.

```
6   dict = Dict("Key" => "Value", "Key2" => "Val2")
7   if haskey(dict, "Foo") == false
8       dict["Foo"] = "Bar"
9       println("""Added: dict["Foo"] = $(dict["Foo"])""")
10  else
11      println("""dict["Foo"] = $(dict["Foo"])""")
12  end
```

```
14  sum = 3 + 7
15  difference = 10 - 3
16  product = 20 * 5
17  quotient = 100 / 10
18  power = 5 ^ 2 ^ 3
19  modulus = 101 % 2
20  s = "hi" * "hi"
21  s = "hi" ^ 2
```

- Having use of natural operators for arithmetical operations in infix notation increases readability: +, -, *, /, ^, % (**Different example: String concatenation**)
- Having not use any special characters in variable definition decreases readability
- All types can be returned from can be returned from the functions support orthogonality

[3] Stack Overflow, https://stackoverflow.com/questions/45616604/why-doesnt-julia-support-something-like-switch-case/45620246

Readability: Type System [4]

- Describing Julia in the lingo of type systems, it is:
  - **Dynamic:** Nothing is known about types until run time
  - **Nominative:** Two variables are type-compatible if and only if their declarations name the same type.
  - **Parametric:** Generic types can be parameterized.
- No meaningful concept of "**compile-time type**", all types are "**run-time type**".
- Only values, not variables, have types – variables are simply names bound to values.
- The default behavior in Julia when types are omitted is to allow values to be of any type (Any: A very readable name !).
- Annotations can be added to types by using :: operator in order to:
  - Take advantage of Julia's powerful multiple-dispatch mechanism
  - Improve human readability
  - Catch programming errors
  - Improve performance in some cases
- Concrete types **may not** subtype each other: all concrete types are final and may only have abstract types as their supertypes.

[4] Types – The Julia Language, https://docs.julialang.org/en/v1/manual/types/

Readability: Type System [4]

Abstract Types

```
abstract type Number end
abstract type Real     <: Number end
abstract type AbstractFloat <: Real end
abstract type Integer  <: Real end
abstract type Signed   <: Integer end
abstract type Unsigned <: Integer end
```

Primitive Types

```
primitive type Float16 <: AbstractFloat 16 end
primitive type Float32 <: AbstractFloat 32 end
primitive type Float64 <: AbstractFloat 64 end

primitive type Bool <: Integer 8 end
primitive type Char <: AbstractChar 32 end

primitive type Int8    <: Signed   8 end
primitive type UInt8   <: Unsigned 8 end
primitive type Int16   <: Signed   16 end
primitive type UInt16  <: Unsigned 16 end
primitive type Int32   <: Signed   32 end
primitive type UInt32  <: Unsigned 32 end
primitive type Int64   <: Signed   64 end
primitive type UInt64  <: Unsigned 64 end
primitive type Int128  <: Signed   128 end
primitive type UInt128 <: Unsigned 128 end
```

Parametric Types

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

```
julia> abstract type Pointy{T} end
```

```
struct Tuple2{A,B}
    a::A
    b::B
end
```

Type Unions

```
julia> IntOrString = Union{Int,AbstractString}
Union{Int64, AbstractString}

julia> 1 :: IntOrString
1

julia> "Hello!" :: IntOrString
"Hello!"
```

Composite Types

```
struct Foo
    bar
    baz::Int
    qux::Float64
end
```

```
mutable struct Bar
    baz
    qux::Float64
end

bar = Bar("Hello", 1.5);

bar.qux = 2.0
```

[4] Types – The Julia Language, https://docs.julialang.org/en/v1/manual/types/

Language Evaluation - Writability

- Uses dynamic typing, i.e., you do not need to care about the data types of the variables. Julia understands it from the context. Two separate string notations: Quotes and triple-quotes.

```
a = 3                    # Int64
b = 100 / 10             # Float64
c = "String example !"   # String
```

```
s1 = "A string example."
s2 = """Triple-quotes" string example for "strings that contain quotes"."""
```

- Comprehensive packages (**4712** registered packages in package registry [4]) that make writing the code fast and easily in the application domains mentioned earlier.
- It is very easy to access built-in help system.
  - ?convert # This command explains the convert function with examples
- Lots of syntactic sugars increases writability:
  - For loops example
  - Function declaration example including Duck-typing feature
  - Array definition and initialization example
  - If conditionals, ternary operator

[4] Julia Registries/General - GitHub, https://github.com/JuliaRegistries/General/blob/master/Registry.toml

Language Evaluation - Reliability

- Dynamic type coercion at run-time **seams** to reduce reliability.
- Different functions to do type conversion and string type to numeric type conversion: convert and parse functions

```
convert(Int64, "1")             # Fires exception at run-time:
                                # "MethodError: Cannot `convert` an object of type String to an object of type Int64"
result = parse(Int64, "1")      # The value of result will be 1 and the type of the result will be Int64
result = parse(Int64, "1.0")    # ArgumentError: invalid base 10 digit '.' in "1.0"
```

- The division operator always returns real number result even though all the operands are integer types.

```
quotient = 100 / 10
typeof(quotient) # The result will be Float64
```

- Type annotations improves reliability
  - Type checking of variables and even expressions at run-time
  - When appended to a variable on the left-hand side of an assignment, or as part of a local declaration, the :: operator declares the variable to always have the specified type, like a type declaration in a statically-typed language such as C.
- The use of @assert keyword improves reliability

```
days = 365
@assert days == 365
```

```
(1+2)::AbstractFloat  # ERROR: TypeError: in typeassert,
                      # expected AbstractFloat, got a value of type Int64
local x::Int8  # in a local declaration
x::Int8 = 10   # as the left-hand side of an assignment
```

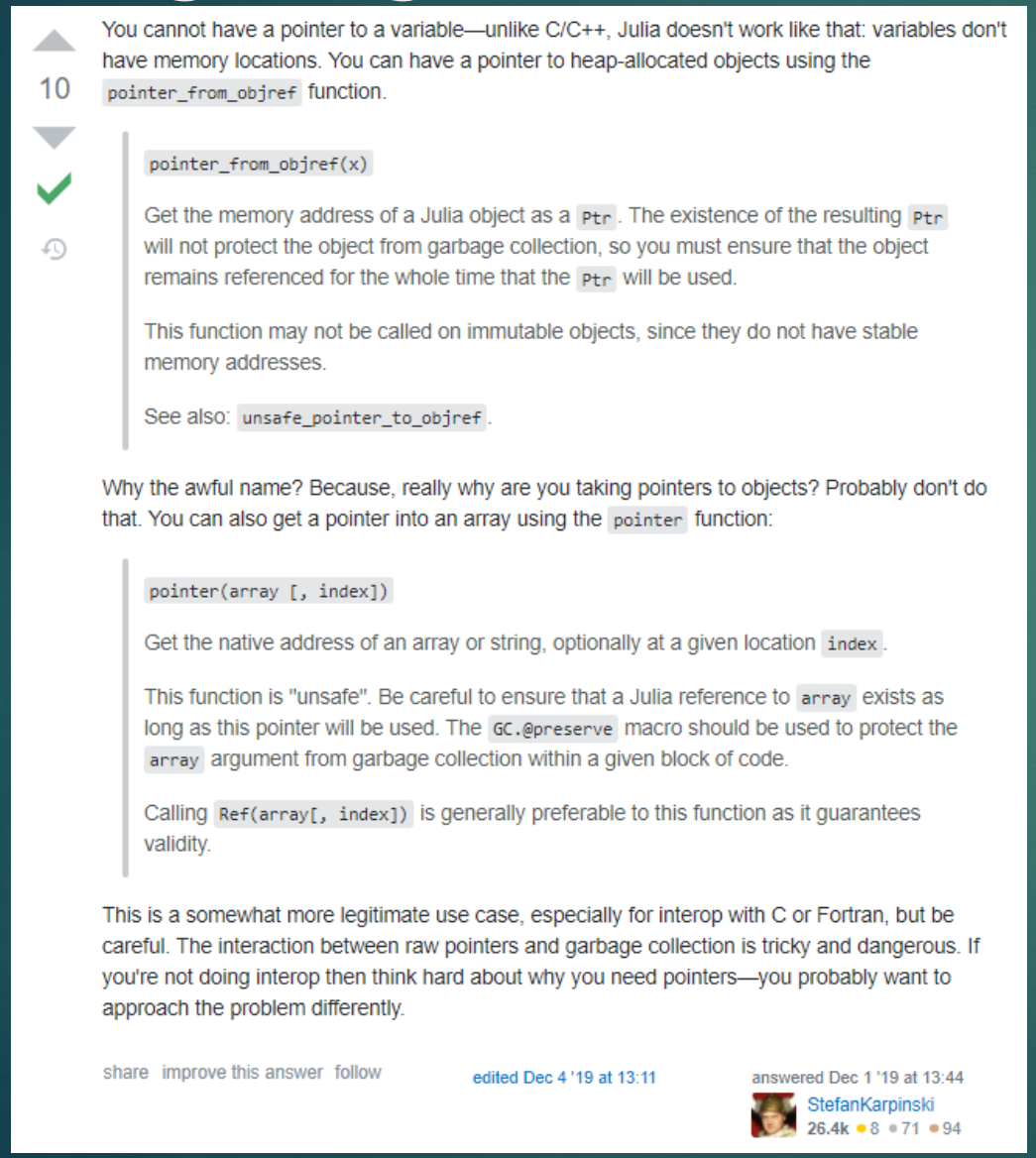
- Power operator is evaluated right-associative as expected 😎

```
power = 5 ^ 2 ^ 3   # The result will be 5 ^ 8 = 390625
```

Language Evaluation - Reliability

- Julia has a very powerful type system
- Multiple dispatch makes the Julia code **generic** and **fast**. It also improves reliability.
  - **Multiple dispatch:** Whenever a **generic function** is called on a particular set of arguments, Julia will infer the types of the inputs and dispatch the appropriate **method**.
  - The code can be **generic** and **flexible** because the code can be written in terms of **abstract operations** such as addition and multiplication, rather than in terms of specific implementations.
  - The code also runs quickly because Julia is able to call efficient methods for the relevant types.
  - @which macro shows which method is being dispatched.
- Supports exception handling: built-in exceptions, throw function, error function, the try/catch statement, finally clause
- More advanced error handling functions: rethrow, backtrace, catch_backtrace, Base.catch_stack

Language Evaluation - Reliability



10

You cannot have a pointer to a variable—unlike C/C++, Julia doesn't work like that: variables don't have memory locations. You can have a pointer to heap-allocated objects using the `pointer_from_objref` function.

> `pointer_from_objref(x)`
>
> Get the memory address of a Julia object as a `Ptr`. The existence of the resulting `Ptr` will not protect the object from garbage collection, so you must ensure that the object remains referenced for the whole time that the `Ptr` will be used.
>
> This function may not be called on immutable objects, since they do not have stable memory addresses.
>
> See also: `unsafe_pointer_to_objref`.

Why the awful name? Because, really why are you taking pointers to objects? Probably don't do that. You can also get a pointer into an array using the `pointer` function:

> `pointer(array [, index])`
>
> Get the native address of an array or string, optionally at a given location `index`.
>
> This function is "unsafe". Be careful to ensure that a Julia reference to `array` exists as long as this pointer will be used. The `GC.@preserve` macro should be used to protect the `array` argument from garbage collection within a given block of code.
>
> Calling `Ref(array[, index])` is generally preferable to this function as it guarantees validity.

This is a somewhat more legitimate use case, especially for interop with C or Fortran, but be careful. The interaction between raw pointers and garbage collection is tricky and dangerous. If you're not doing interop then think hard about why you need pointers—you probably want to approach the problem differently.

share improve this answer follow

edited Dec 4 '19 at 13:11

answered Dec 1 '19 at 13:44

StefanKarpinski

26.4k ● 8 ● 71 ● 94

- Can I define a pointer in Julia?
- The answer of Stephan Karpinski from stackoverflow.com
- The short answer: **Yes, you can, but do you really need it !!**

Language Evaluation - Cost

- Julia is a free and open-source under the MIT license
- Source code is available on GitHub
- Julia is easy to learn
- Good online documentation (https://docs.julialang.org/)
- Julia Academy: Source of free online video courses
- It compiles fast with its Just-In-Time (JIT) compiler
- Fast execution of programs
- Very fast development of applications in various domains

Language Evalutation - Portability

Linux

Windows

MacOS

Portable: Prebuilt binaries for Linux, Windows, MacOS as well as the source code can be downloaded from: https://julialang.org/downloads/

Language Characteristics

- Julia is an imperative and quasi-object oriented language that supports metaprogramming
- Julia is compiled, like C or Fortran, so it's fast. Julia is compiled at runtime ('Just In Time' – JIT – for execution)[5].
- Integrated Development Environments
  - Juno
  - Visual Studio Code
  - Atom
  - Sublime
  - Vim
  - Emacs
- Popularity: #26 in TIOBE index

[5] Julia in a Nutshell -- Agile, https://agilescientific.com/blog/2014/9/4/julia-in-a-nutshell.html

Julia in a Nutshell[1]

Fast

Julia was designed from the beginning for high performance. Julia programs compile to efficient native code for multiple platforms via LLVM.

Dynamic

Julia is dynamically typed, feels like a scripting language, and has good support for interactive use.

Reproducible

Reproducible environments make it possible to recreate the same Julia environment every time, across platforms, with pre-built binaries.

Open

Julia is an open source project with over 1,000 contributors. It is made available under the MIT license. The source code is available on GitHub.

General

Julia provides asynchronous I/O, metaprogramming, debugging, logging, profiling, a package manager, and more. One can build entire Applications and Microservices in Julia.

Composable

Julia uses multiple dispatch as a paradigm, making it easy to express many object-oriented and functional programming patterns. The source code is available on GitHub.

[2] The Julia Programming Language, https://julialang.org/

Personal Evaluation

- Easy to learn, easy to setup
- Very strong type system that improves readability and reliability
- Powerful, fast, portable and reliable
- Comprehensive: Meets the needs of the popular application domains
- Supports multi-treaded programming: Threads, Tasks (i.e., coroutine)
- Making use of modern hardware, such as GPUs very easily
- Seamless integration with C, Fortran, C ++, Python, R, Java and many others

I am a Julia enthusiast now !

References

- The Julia Programming Language, https://julialang.org/
- Julia Documentation, https://docs.julialang.org/en/v1/
- Stack Overflow, https://stackoverflow.com/
- Julia in a Nutshell -- Agile, https://agilescientific.com/blog/2014/9/4/julia-in-a-nutshell.html
- Best coding IDE for Julia, https://medium.com/dev-genius/what-is-the-best-ide-for-developing-in-the-programming-language-julia-484c913f07bc
- Julia Academy, https://juliaacademy.com/
- TIOBE – The Software Quality, https://www.tiobe.com/tiobe-index/
- Julia Registries/General - GitHub, https://github.com/JuliaRegistries/General/blob/master/Registry.toml
- Wikipedia: Julia (programming language), https://en.wikipedia.org/wiki/Julia_(programming_language)