The Fast Track to julia 1.0

A quick and dirty overview of

This page's source is located here. Pull requests are welcome!

What is...?

Julia is an open-source, multi-platform, high-level, high-performance programming language for technical computing.

Julia has an LLVM-based JIT compiler that allows it to match the performance of languages such as C and FORTRAN without the hassle of low-level code. Because the code is compiled on the fly you can run (bits of) code in a shell or REPL, which is part of the recommended workflow.

Julia is dynamically typed, provides multiple dispatch, and is designed for parallelism and distributed computation.

Julia has a built-in package manager.

Julia has many built-in mathematical functions, including special functions (e.g. Gamma), and supports complex numbers right out of the box.

Julia allows you to generate code automagically thanks to Lisp-inspired macros.

Julia was born in 2012.

Basics

Assignment	`answer = 42`
	`x, y, z = 1, [1:10; ], "A String"`
	`x, y = y, x  # swap x and y`

Constant declaration	`const DATE_OF_BIRTH = 2012`
End-of-line comment	`i = 1 # This is a comment`
Delimited comment	`#= This is another comment =#`

Chaining	`x = y = z = 1  # right-to-left`
	`0 < x < 3      # true`
	`5 < x != y < 5 # false`

Function definition	`function add_one(i)`
	`return i + 1`
	`end`

Insert LaTeX symbols	`\delta` + [Tab]

Operators

Basic arithmetic	`+`, `-`, `*`, `/`
Exponentiation	`2^3 == 8`
Division	`3/12 == 0.25`
Inverse division	`7\3 == 3/7`
Remainder	`x % y` or `rem(x,y)`
Negation	`!true == false`
Equality	`a == b`
Equality (composite types)	`is(a, b) # done on bit level`
Inequality	`a != b` or `a ≠ b`
Less and larger than	`<` and `>`
Less than or equal to	`<=` or `≤`
Greater than or equal to	`>=` or `≥`
Element-wise operation	`[1, 2, 3] .+ [1, 2, 3] == [2, 4, 6]` `[1, 2, 3] .* [1, 2, 3] == [1, 4, 9]`
Not a number	`isnan(NaN)` not(!) `NaN == NaN`
Ternary operator	`a == b ? "Equal" : "Not equal"`
Short-circuited AND and OR	`a && b` and `a || b`
Object equivalence	a === b

The shell a.k.a. REPL

Recall last result	`ans`
Interrupt execution	[Ctrl] + [C]
Clear screen	[Ctrl] + [L]
Run program	`include("filename.jl")`
Get help for `func` is defined	`?func`
See all places where `func` is defined	`apropos("func")`
Escape to OS command line	`;`
Enter interactive Package Manger	`]`
Exit	`quit()` or [Ctrl] + [D]

Package management

Packages must be registered before they are visible to the package manager. In Julia 1.0, first run `using Pkg.`

List installed packages (human-readable)	`Pkg.status()`
List installed packages (machine-readable)	`Pkg.installed()`
Update all packages	`Pkg.update()`
Install `PackageName`	`Pkg.add("PackageName")`
Checkout `Package source from Git`	`Pkg.checkout("Git URL")`
Rebuild `PackageName`	`Pkg.build("PackageName")`
Use `PackageName` (after install)	`using PackageName`
Remove `PackageName`	`Pkg.rm("PackageName")`

Characters and strings

Character	`chr = 'C'`
String	`str = "A string"`
Character code	`Int('J') == 74`
Character from code	`Char(74) == 'J'`
Any UTF character	`chr = '\uXXXX'     # 4-digit HEX` `chr = '\uXXXXXXXX' # 8-digit HEX`

Loop through characters	`for c in str`
	`println(c)`
	`end`

Concatenation	`str = "Learn" * " " * "Julia"`
String interpolation	`a = b = 2` `println("a * b = $(a*b)")`
First matching character or regular expression	`search("Julia", 'i') == 4`
Replace substring or regular expression	`replace("Julia", "a", "us") == "Julius"`
Last index (of collection)	`endof("Hello") == 5`
Number of characters	`length("Hello") == 5`
Regular expression	`pattern = r"l[aeiou]"`

Subexpressions	`str = "+1 234 567 890"`
	`pat = r"\+([0-9]) ([0-9]+)"`
	`m = match(pat, str)`
	`m.captures == ["1", "234"]`

All occurrences	`matchall(pat, str)`
All occurrences (as iterator)	`eachmatch(pat, str)`

Beware of multi-byte Unicode encodings in UTF-8:

```
10 == endof("Ångström") != length("Ångström") == 8
```

Strings are immutable.

Numbers

Integer types	`IntN` and `UIntN`, with `N` ∈ {`8`,`16`,`32`,`64`,`128`} `BigInt`
Floating-point types	`FloatN` with `N` ∈ {`16`, `32`, `64`} `BigFloat`
Minimum and maximum values by type	`typemin(Int8)` `typemax(Int64)`
Complex types	`Complex{T}`
Imaginary unit	`im`
Machine precision	`eps() # same as eps(Float64)`
Rounding	`round()  # floating-point` `iround() # integer`
Random numbers	`rand()   # uniform [0,1)` `randn()  # normal (-Inf, Inf)`

Random from Other Distribution	`using Distributions`
	`my_dist = Bernoulli(0.2) # For example`
	`rand(my_dist)`

Type conversions	`convert(TypeName, val) # attempt/error` `typename(val)          # calls convert`

Julia does *not* automatically check for numerical overflow. Use package `SaferIntegers` for ints with overflow checking

Mathematics

Global constants	`pi      # 3.1415...`
	`π       # 3.1415...`
	`e       # 2.7182...`
	`im      # real(im * im) == -1`
	`catalan # 0.9159...`
	`γ       # 0.5772...`
	`φ       # 1.6180...`

Identity matrix	`eye(n) # n×n`
Define matrix	`M = [1 0; 0 1] # eye(2)`
Matrix dimensions	`size(M)`
Select `i`th row	`M[i, :]`
Select `i`th column	`M[:, i]`
Concatenate horizontally	`M = [a b]` or `M = hcat(a, b)`
Concatenate vertically	`M = [a ; b]` or `M = vcat(a, b)`
Matrix transposition	`transpose(M)`
Conjugate matrix transposition	`M'` or `ctranspose(M)`
Matrix trace	`trace(M)`
Matrix determinant	`det(M)`
Matrix rank	`rank(M)`
Matrix eigenvalues	`eigvals(M)`
Matrix eigenvectors	`eigvecs(M)`
Matrix inverse	`inv(M)`
Solve `M*x == v`	`M\v` is better than `inv(M)*v`
Moore-Penrose pseudo-inverse	`pinv(M)`

Julia provides many mathematical (special) and statistical functions.

Julia has built-in support for matrix decompositions.

Control flow and loops

Conditional	`if-elseif-else-end`

Simple `for` loop	`for i in 1:10`
	`println(i)`
	`end`

Unnested `for` loop	`for i in 1:10, j = 1:5`
	`println(i*j)`
	`end`

Enumeration	`for (idx, val) in enumerate(arr)`
	`println("the $idx-th element is $val")`
	`end`

`while` loop	`while bool_expr`
	`# do stuff`
	`end`

Exit loop	`break`
Exit iteration	`continue`

Functions

All arguments to functions are passed by reference.

Functions with `!` appended change at least one argument, typically the first: `sort!(arr)`.

Required arguments are separated with a comma and use the positional notation.

Optional arguments need a default value in the signature, defined with `=`.

Keyword arguments use the named notation and are listed in the function's signature after the semicolon:

```
function func(req1, req2; key1=dflt1, key2=dflt2)
# do stuff
end
```

The semicolon is *not* required in the call to a function that accepts keyword arguments.

The `return` statement is optional but highly recommended.

Multiple data structures can be returned as a tuple in a single `return` statement.

Command line arguments `julia script.jl arg1 arg2...`can be processed from global constant `ARGS`:

```
for arg in ARGS
println(arg)
end
```

Anonymous functions can best be used in collection functions or list comprehensions: `x -> x^2`.

Functions can accept a variable number of arguments:

```
function func(a...)
println(a)
end

func(1, 2, [3:5]) # tuple: (1,2,[3,4,5])
```

Functions can be nested:

```
function outerfunction()
# do some outer stuff
function innerfunction()
  # do inner stuff
  # can access prior outer definitions
end
# do more outer stuff
end
```

Functions can have explicit return types

```
# take any Number subtype and return it as a String
function stringifynumber{T<:Number}(num::T)::String
return "$num"
```

Functions can be vectorized by using the Dot Syntax

```
# here we broadcast the subtraction of each mean value
# by using the dot operator
julia> A = rand(3,4);

julia> B = A .- mean(A,1)
3×4 Array{Float64,2}:
0.343976   0.427378  -0.503356  -0.00448691
 -0.210096  -0.531489   0.168928  -0.128212
 -0.13388    0.104111   0.334428   0.132699

julia> mean(B,1)
1×4 Array{Float64,2}:
 0.0  0.0  0.0  0.0
```

Julia generates specialized versions of functions based on data types. When a function is called with the same argument types again, Julia can look up the native machine code and skip the compilation process.

Since Julia 0.5 the existence of potential ambiguities is still acceptable, but actually calling an ambiguous method is an immediate error.

Stack overflow is possible when recursive functions nest many levels deep. Trampolining can be used to do tail-call optimization, as Julia does not do that automatically yet. Alternatively, you can rewrite the tail recursion as an iteration.

Arrays

Declaration	`arr = Float64[]`
Pre-allocation	`sizehint(arr, 10^4)`
Access and assignment	`arr[1] = "Some text"`

Comparison	`a = [1:10; ]`
	`b = a      # b points to a`
	`a[1] = -99`
	`a == b     # true`

Copy elements (not address)	`b = copy(a)` `b = deepcopy(a)`
Select subarray from `m` to `n`	`arr[n:m]`
`n`-element array with `0.0`s	`zeros(n)`
`n`-element array with `1.0`s	`ones(n)`
`n`-element array with `#undef`s	`cell(n)`
`n` equally spaced numbers from `start` to `stop`	`linspace(start, stop, n)`
Array with `n` random `Int8` elements	`rand(Int8, n)`
Fill array with `val`	`fill!(arr, val)`
Pop last element	`pop!(arr)`
Pop *first* element	`shift!(a)`
Push `val` as last element	`push!(arr, val)`
Push `val` as *first* element	`unshift!(arr, val)`
Remove element at index `idx`	`splice!(arr, idx)`
Sort	`sort!(arr)`
Append `a` with `b`	`append!(a,b)`
Check whether `val` is element	`in(val, arr)` or `val in arr`
Scalar product	`dot(a, b) == sum(a .* b)`
Change dimensions (if possible)	`reshape(1:6, 3, 2)' == [1 2 3; 4 5 6]`
To string (with delimiter `del` between elements)	`join(arr, del)`

Dictionaries

Dictionary	`d = Dict(key1 => val1,`
	`       key2 => val2, ...)`
	`d = Dict(:key1 => val1,`
	`       :key2 => val2, ...)`

All keys (iterator)	`keys(d)`
All values (iterator)	`values(d)`

Loop through key-value pairs	`for (k,v) in d`
	`println("key: $k, value: $v")`
	`end`

Check for key `:k`	`haskey(d, :k)`
Copy keys (or values) to array	`arr = collect(keys(d))` `arr = [k for (k,v) in d]`

Dictionaries are mutable; when symbols are used as keys, the keys are immutable.

Sets

Declaration	`s = Set([1, 2, 3, "Some text"])`
Union `s1 ∪ s2`	`union(s1, s2)`
Intersection `s1 ∩ s2`	`intersect(s1, s2)`
Difference `s1 \ s2`	`setdiff(s1, s2)`
Difference `s1 △ s2`	`symdiff(s1, s2)`
Subset `s1 ⊆ s2`	`issubset(s1, s2)`

Checking whether an element is contained in a set is done in O(1).

Collection functions

Apply `f` to all elements of collection `coll`	`map(f, coll)` or
	`map(coll) do elem`
	`# do stuff with elem`
	`# must contain return`
	`end`

Filter `coll` for true values of `f`	`filter(f, coll)`
List comprehension	`arr = [f(elem) for elem in coll]`

Types

Julia has no classes and thus no class-specific methods.

Types are like classes without methods.

Abstract types can be subtyped but not instantiated.

Concrete types cannot be subtyped.

By default, `struct`s are immutable.

Immutable types enhance performance and are thread safe, as they can be shared among threads without the need for synchronization.

Objects that may be one of a set of types are called `Union` types.

Type annotation	`var::TypeName`

Type declaration	`struct Programmer`
	`name::String`
	`birth_year::UInt16`
	`fave_language::AbstractString`
	`end`

Mutable type declaration	replace `struct` with `mutable struct`
Type alias	`const Nerd = Programmer`
Type constructors	`methods(TypeName)`
Type instantiation	`me = Programmer("Ian", 1984, "Julia")` `me = Nerd("Ian", 1984, "Julia")`

Subtype declaration	`abstract Programmer`
	`struct Hacker <: Programmer`
	`name::AbstractString`
	`birth_year::UInt16`
	`fave_language::AbstractString`
	`end`

Parametric type	`struct Point{T <: Real}`
	`x::T`
	`y::T`
	`end`
	
	`p = Point{Float64}(1,2)`

Union types	`Union{Int, String}`
Traverse type hierarchy	`super(TypeName)` and `subtypes(TypeName)`
Default supertype	`Any`
All fields	`fieldnames(TypeName)`
All field types	`TypeName.types`
Nullable type	`Nullable{T}`

When a type is defined with an *inner* constructor, the default *outer* constructors are not available and have to be defined manually if need be. An inner constructor is best used to check whether the parameters conform to certain (invariance) conditions. Obviously, these invariants can be violated by accessing and modifying the fields directly, unless the type is defined as immutable. The `new` keyword may be used to create an object of the same type.

Type parameters are invariant, which means that `Point{Float64} <: Point{Real}` is false, even though `Float64 <: Real`. Tuple types, on the other hand, are covariant: `Tuple{Float64} <: Tuple{Real}`.

The type-inferred form of Julia's internal representation can be found with `code_typed()`. This is useful to identify where `Any` rather than type-specific native code is generated.

Missing and Nothing

Progammers Null	`nothing`
Missing Data	`missing`
Not a Number in Float	`NaN`
Filter missings	`skipmissing([1, 2, missing]) == [1,2]`
Replace missings	`collect((df[:col], 1))`
Check if missing	`ismissing(x)` not `x == missing`

Exceptions

Throw SomeExcep	`throw(SomeExcep())`
Rethrow current exception	`rethrow()`

Define NewExcep	`type NewExcep <: Exception`
	`v::AbstractString`
	`end`
	
	`Base.showerror(io::IO, e::NewExcep) =`
	`print(io, "A problem with $(e.v)!")`
	
	`throw(NewExcep("x"))`

Throw error with msg text	`error(msg)`
Throw warning with msg text	`warn(msg)`
Throw information with msg text	`info(msg)`
Handler	see below

```
try
# do something potentially iffy
catch ex
if isa(ex, SomeExcep)
  # handle SomeExcep
elseif isa(ex, AnotherExcep)
  # handle AnotherExcep
else
  # handle all others
finally
  # do this in any case
end
```

Modules

Modules are separate global variable workspaces that group together similar functionality.

Definition

```
module PackageName
# add module definitions
# use export to make definitions accessible
end
```

Include `filename.jl`

```
include("filename.jl")
```

Load

```
using ModuleName          # all exported names
using ModuleName: x, y    # only x, y
using ModuleName.x,
    ModuleName.y:       # only x, y
import ModuleName         # only ModuleName
import ModuleName: x, y # only x, y
import ModuleName.x,
     ModuleName.y       # only x, y
```

Exports

```
# Get an array of names exported by Module
names(ModuleName)

# include non-exports, deprecateds
# and compiler-generated names
names(ModuleName, all::Bool)

#also show names explicitely imported from other modules
names(ModuleName, all::Bool, imported::Bool)
```

There is only one difference between `using` and `import`: with `using` you need to say `function Foo.bar(..` to extend module `Foo`'s function `bar` with a new method, but with `import Foo.bar`, you only need to say `function bar(...` and it automatically extends module `Foo`'s function `bar`.

Expressions

Julia is homoiconic: programs are represented as data structures of the language itself. In fact, everything is an expression `Expr`.

Symbols are interned strings prefixed with a colon. Symbols are more efficient and they are typically used as identifiers, keys (in dictionaries), or columns in data frames. Symbols cannot be concatenated.

Quoting `:( ... )` or `quote ... end` creates an expression, just like `parse(str)`, and `Expr(:call, ...)`.

```
x = 1
line = "1 + $x"      # some code
expr = parse(line)   # make an Expr object
typeof(expr) == Expr # true
dump(expr)           # generate abstract syntax tree
eval(expr) == 2      # evaluate Expr object: true
```

Macros

Macros allow generated code (i.e. expressions) to be included in a program.

Definition

```
macro macroname(expr)
# do stuff
end
```

Usage: `@macroname(ex1, ex2, ...)` or `@macroname ex1, ex2, ...`

Built-in macros

```
@test           # equal (exact)
@test_approx_eq # equal (modulo numerical errors)
@test x ≈ y     # isapprox(x, y)
@assert         # assert (unit test)
@which          # types used
@time           # time and memory statistics
@elapsed        # time elapsed
@allocated      # memory allocated
@profile        # profile
@spawn          # run at some worker
@spawnat        # run at specified worker
@async          # asynchronous task
@parallel       # parallel for loop
@everywhere     # make available to workers
```

Rules for creating *hygienic* macros:

- Declare variables inside macro with `local`.
- Do not call `eval` inside macro.
- Escape interpolated expressions to avoid expansion: `$(esc(expr))`.

Tasks a.k.a. coroutines

Tasks provide a lightweight threading mechanism and they can be suspended and resumed at will. Tasks follow the producer-consumer model, which means that between consumption calls the task is suspended.

Tasks are *not* executed in different threads, so they cannot run on different processors.

Tasks have low overhead because switching between them does not consume stack space unlike normal function calls.

Define: `t = Task( () -> somefunc(...) )` or `t = @task somefunc(...)`

Produce

```
function somefunc(...)
# do stuff
produce(...)
end
```

Consume: `consume(t)`

Parallel Computing

Launch REPL with N workers	`julia -p N`
Number of available workers	`nprocs()`
Add N workers	`addprocs(N)`

See all worker ids	`for pid in workers()`
	`println(pid)`
	`end`

Get id of executing worker	`myid()`
Remove worker	`rmprocs(pid)`

Run f with arguments args on pid	`r = remotecall(f, pid, args...)`
	`# or:`
	`r = @spawnat pid f(args)`
	`...`
	`fetch(r)`

Run f with arguments args on pid (more efficient)	`remotecall_fetch(f, pid, args...)`
Run f with arguments args on any worker	`r = @spawn f(args) ... fetch(r)`
Run f with arguments args on all workers	`r = [@spawnat w f(args) for w in workers()] ... fetch(r)`
Make expr available to all workers	`@everywhere expr`

Parallel for loop with reducer function red	`sum = @parallel (red) for i in 1:10^6`
	`# do parallel stuff`
	`end`

Apply f to all elements in collection coll	`pmap(f, coll)`

Workers are also known as concurrent/parallel processes.

Modules with parallel processing capabilities are best split into a functions file that contains all the functions and variables needed by all workers, and a driver file that handles the processing of data. The driver file obviously has to import the functions file.

A non-trivial (word count) example of a reducer function is provided by Adam DeConinck.

I/O

Read stream	`stream = STDIN`
	`for line in eachline(stream)`
	`# do stuff`
	`end`

Read file	`open(filename) do file`
	`for line in eachline(file)`
	`  # do stuff`
	`end`
	`end`

Read CSV file	`using CSV` `CSV.read(filename)`
Save Julia Object	`using JLD` `save(filename, "object_key", object, ...)`
Load Julia Object	`using JLD` `d = load(filename) # Returns a dict of objects`
Save HDF5	`using HDF5` `h5write(filename, "key", object)`
Load HDF5	`using HDF5` `h5read(filename, "key")`

Data frames

Read CSV	`using CSV` `CSV.read(filename)`
Read CSV	`using CSV` `CSV.write(filename)`
Read Stata, SPSS, etc.	`StatFiles` Package
Describe data frame	`describe(df)`
Make vector of column col	`v = df[:col]`
Sort by col	`sort!(df, cols = [:col])`
Pool col	`pool!(df, [:col])`
List col levels	`levels(df[:col])`
All observations with col==val	`df[df[:col] .== val, :]` `subset(df, :(col .== val))`

Reshape from wide to long format	`stack(df, [1:n; ])`
	`stack(df, [:col1, :col2, ...])`
	`melt(df, [:col1, :col2]) [`

Reshape from long to wide format	`unstack(df, :id, :val)`
Make Nullable	`allowmissing!(df)` or `allowmissing!(df, :col)`

Loop over Rows	`for r in eachrow(df)`
	`# do stuff.`
	`# r is Struct with fields of col names.`
	`end`

Loop over Columns	`for c in eachcol(df)`
	`# do stuff.`
	`# c is tuple with name, then vector`
	`end`

Apply func to groups	`by(df, :group_col, func)`
Query	see below

```
using Query
query = @from r in df begin
      @where r.col1 > 40
      @select {new_name=r.col1, r.col2}
      @collect DataFrame #Default: iterator
      end
```

Introspection and reflection

Type	`typeof(name)`
Type check	`isa(name, TypeName)`
List subtypes	`subtypes(name)`
List supertype	`super(TypeName)`
Function methods	`methods(func)`
JIT bytecode	`code_llvm(expr)`
Assembly code	`code_native(expr)`

Noteworthy packages and projects

Many core packages are managed by communities with names of the form Julia[Topic].

Statistics	JuliaStats
Automatic differentiation	JuliaDiff
Numerical optimization	JuliaOpt
Plotting	JuliaPlots
Network (Graph) Analysis	JuliaGraphs
Web	JuliaWeb
Geo-Spatial	JuliaGeo
Data frames	`DataFrames`
Statistical distributions	`Distributions`
Data visualization à la `ggplot2`	`Gadfly`

Analytics	`Regression # linear/logistic regression`
	`GLM # gen. linear models`
	`Lasso # lasso and elastic nets`
	`DecisionTree # trees and random forests`
	`Clustering # clustering`
	`TextAnalysis # text mining`
	`Mocha # deep learning`

Conventions

The main convention in Julia is to avoid underscores unless they are required for legibility.

Variable names are in lower (or snake) case: `somevariable`.

Constants are in upper case: `SOMECONSTANT`.

Functions are in lower (or snake) case: `somefunction`.

Macros are in lower (or snake) case: `@somemacro`.

Type names are in initial-capital camel case: `SomeType`.

Julia files have the `jl` extension.

John Myles White has provided a comprehensive style guide with more details and suggestions.

Performance tips

- Write type-stable code.
- In fact, use immutable types where possible.
- Use `sizehint` for large arrays.
- Free up memory for large arrays with `arr = nothing`.
- Access arrays along columns, because multi-dimensional arrays are stored in column-major order.
- Pre-allocate resultant data structures.
- Disable the garbage collector in real-time applications: `disable_gc()`.
- Avoid the splat (`...`) operator for keyword arguments.
- Use mutating APIs (i.e. functions with `!` to avoid copying data structures.
- Use array (element-wise) operations instead of list comprehensions.
- Avoid `try-catch` in (computation-intensive) loops.
- Avoid `Any` in collections.
- Avoid abstract types in collections.
- Avoid string interpolation in I/O.
- Devectorizing does not improve speed (unlike R, MATLAB or Python).
- Avoid `eval` at run-time.

IDEs, Editors and Plug-ins

- Juno (editor)
- JuliaBox (online IJulia notebook)
- Jupyter (online IJulia notebook)
- LiClipse (editor)
- gedit syntax highlighting (editor)
- Emacs Julia mode (editor)
- vim Julia mode (editor)
- VS Code extension (editor)
- Sublime Text Julia plug-in (editor)
- Sublime Text IJulia plug-in (editor)

Resources

- Official documentation.
- Learning Julia page.
- Month of Julia
- Community standards.
- Julia: A fresh approach to numerical computing (pdf)
- Julia: A Fast Dynamic Language for Technical Computing (pdf)

Videos

- The 5th annual JuliaCon 2018
- The 4th annual JuliaCon 2017 (Berkeley)
- The 3rd annual JuliaCon 2016
- Getting Started with Julia by Leah Hanson
- Intro to Julia by Huda Nassar
- Introduction to Julia for Pythonistas by John Pearson