24

Programming in Julia

Nathan Warner

Northern Illinois
University

Computer Science
Northern Illinois University
United States

Contents

Creating and running Julia scripts, and outputting

First, make sure you have Julia (juliaup) downloaded on your machine. Create a .jl file and run it with

```
julia script.jl
```

Here is a simple Hello World! script written in julia

```
# helloworld.jl
println("Hello World!")
```

1.1 Outputting data

1.1.1 println

Prints the value followed by a newline. Suitable for general-purpose printing

```
println("Hello world")
```

1.1.2 Print

Similar to `println`, but does not append a newline

1.1.3 Formatted output: @printf from the Printf module

Prints the value followed by a newline. Suitable for general-purpose printing

```
using Printf
    @printf("Pi to 3 decimal places: %.3f\n", pi)  # Output: Pi
↪  to 3 decimal places: 3.142
```

1.1.4 Error and debugging output

- **@warn**: Logs a warning message.

```
@warn "This is a warning message."
```

- **@info**: Logs an informational message.

```
@info "This is an informational message."
```

- **@error**: Logs an error message.

```
@error "This is an error message."
```

- **@show**: Prints the expression and its value. Useful for debugging.

```
x = 42
@show x  # Output: x = 42
```

1.1.5 Display

Displays a value using a richer representation (e.g., for plots or tables in Jupyter).

```
display("Hello, World!")  # Output: "Hello, World!"
```

Preliminary

2.1 Typing style

Julia has a dynamic and strong typing system with optional type annotations.

2.1.1 Dynamic Typing

Variables don't need explicit type declarations, but their types are checked at runtime.

```
x = 42       # Type inferred as Int
y = 3.14     # Type inferred as Float64
```

2.1.2 Optional Type Annotations

Types can be explicitly specified for variables, function arguments, and return values to enforce constraints or improve readability.

```
function add(a::Int, b::Int)::Int
    return a + b
end
```

2.2 Just in time

Julia is a Just-In-Time (JIT) compiled language. It uses the LLVM compiler framework to generate efficient machine code at runtime.

Julia infers types of variables and function arguments dynamically when a function is called.

For each unique combination of input types, Julia generates specialized machine code. This allows functions to be optimized for the specific types of data they operate on.

The first time a function is called with a specific set of argument types, Julia compiles it to machine code. Subsequent calls with the same types execute the precompiled code, leading to improved performance.

The JIT compilation provides the interactivity of an interpreted language (like Python) with the performance of a compiled language (like C or Fortran).

However, the initial compilation can introduce a "time-to-first-call" latency, which is a common trade-off in JIT-compiled languages

2.3 Language style

Julia's language style is explicit, concise, and block-delimited, resembling languages like Python and MATLAB

2.3.1 Block structure

Block Structure: Julia uses explicit keywords to start and end code blocks, ensuring clarity and readability.

```
function add(a, b)  # Function block starts with `function`
    return a + b
end                 # Block ends with `end`
```

Note: Julia supports optional semicolons (;) for separating statements on the same line.

2.4 Paradigms

Julia is a multi-paradigm programming language that supports a variety of programming styles, allowing flexibility and adaptability for different use cases. The key paradigms Julia offers include

1. **Procedural Programming**: Follows a structured approach with sequences of instructions (procedures or functions).

   Uses variables, loops, and conditionals to control the flow of execution.

2. **Object-Oriented Programming (OOP)**: While Julia doesn't have traditional classes and inheritance, it uses structures (struct) and multiple dispatch to model OOP-like behavior.

   Supports encapsulation and polymorphism using types and methods.

3. **Functional Programming**: Treats functions as first-class citizens (can be passed as arguments, returned as results)

   Supports immutability, higher-order functions, and function composition.

4. **Meta-programming**: Julia has powerful capabilities for generating and transforming code at runtime using macros and the Expr type.

   Enables code introspection and metaprogramming.

5. **Dynamic Programming**: Julia supports dynamic typing and runtime type flexibility, making it suitable for exploratory programming.

6. **Concurrent and Parallel Programming**: Julia provides built-in support for concurrent and parallel computing with features like coroutines (@async), threads, and distributed computing

7. **Scientific and Numerical Programming**: Julia's syntax and performance are well-suited for numerical computation, similar to MATLAB or NumPy.

Provides rich libraries for linear algebra, optimization, and differential equations.

8. **Multiple Dispatch (Core Paradigm of Julia)**: At its core, Julia leverages multiple dispatch, allowing method selection based on the types of all function arguments, not just the first one (as in single dispatch).

   This provides flexibility and is fundamental to Julia's design.

9. **Array-Oriented Programming**: ulia is designed for efficient array and matrix operations, similar to MATLAB or R.

   Supports broadcasting and element-wise operations with the . operator.

2.5 Memory

In Julia, memory management is automatic, and it uses references under the hood for many operations. While Julia doesn't expose raw pointers in the same way as C or C++, it provides mechanisms to work with references and low-level memory when needed

Julia has automatic garbage collection to manage memory, reclaiming unused objects when they are no longer referenced.

Users generally don't need to manually free memory, but they can force garbage collection using GC.gc() if necessary

2.5.1 The garbage collector

A garbage collector is a system for automatic memory management in programming languages. Its purpose is to reclaim memory occupied by objects no longer accessible or needed, thus preventing memory leaks and reducing the need for manual memory management by the programmer.

The garbage collector periodically identifies and frees unused memory by determining which objects are no longer reachable from the program. The process typically involves the following steps:

When objects or variables are created, memory is allocated from the heap (a memory area reserved for dynamic allocation). The garbage collector manages this heap and tracks references to objects.

The GC determines which objects are "reachable" and should not be collected. Reachable objects are those that can be accessed directly or indirectly by the program.

The starting points for determining reachability, such as

- Global variables
- Local variables in the current call stack
- CPU registers

Unreachable objects are those that:

- Are not referenced by any active part of the program.
- Cannot be accessed directly or indirectly from the "roots."

Once the GC identifies unreachable objects, it frees the memory they occupy, making it available for new allocations.

Garbage collectors use a technique called "*reference counting*". Reference counting tracks the number of references to each object. When an object's reference count drops to zero, it is immediately deallocated.

A limitation of garbage collectors are *circular references*. A circular reference occurs when two or more objects reference each other in a way that creates a loop. This can prevent proper memory deallocation in systems that rely solely on reference counting for garbage collection.

2.6 Importing

In Julia, imports are managed using the using and import keywords to access modules and their functionality. Modules are collections of functions, types, and constants that help organize code into namespaces.

2.6.1 Using

Makes all exported symbols of a module available. Symbols (functions, types, etc.) are accessed directly without the module prefix.

```
using Random  # Load the Random module

x = rand(1:10, 5)  # Use `rand` directly (exported by Random)
```

2.6.2 Import

Imports the module but does not bring its symbols into the current namespace.

You must qualify all functions/types with the module name unless explicitly brought into scope.

2.6.3 Selective imports with using

Allows you to bring only specific exported symbols into the current namespace.

```
using Random: rand  # Import only `rand`

x = rand(1:10, 5)  # Use `rand` directly
```

Makes the specific symbol (rand in this case) from the Random module accessible in the current scope. Allows you to use rand directly in your code without the Random. prefix.

However, you cannot extend (add methods to) the rand function using this approach.

2.6.4 Selective imports with import

Allows you to import specific symbols, even if they are not exported by the module.

```
import Base: +

# Extend the `+` operator for a custom type
struct MyType
x::Int
end

+(a::MyType, b::MyType) = MyType(a.x + b.x)
```

Imports the specific symbol (rand) from the Random module into the current scope and allows extending it.

rand is directly accessible in your code.

You can extend the rand function by adding new methods.

2.6.5 Extending functions

If you want to extend a function from a module (e.g., add a new method), you must use import.

```
x = Int8(15)  # Creates an Int8 with value 15
```

The basics

3.1 Data types

Julia has the following data types

3.1.1 Numerical Types

- **Integers:**
  - `Int8, Int16, Int32, Int64, Int128:` Signed integers with various bit sizes.
  - `UInt8, UInt16, UInt32, UInt64, UInt128:` Unsigned integers.
  - `Int:` Default signed integer type (dependent on the platform, typically Int64 or Int32).
- **Floating-point numbers:** `Float16, Float32, Float64:` IEEE 754 floating-point numbers.
- **Big numbers:**
  - `BigInt:` Arbitrary precision integers.
  - `BigFloat:` Arbitrary precision floating-point numbers.
- **Complex numbers:** `Complex{T}:` Complex numbers with real and imaginary parts of type T.
- **Rational numbers:** `Rational{T}:` Fractions represented as numerator//denominator.

In Julia, the default type for a literal integer like 15 is Int (platform-dependent, typically Int64 on 64-bit systems). However, you can explicitly create integers of specific types (Int8, Int16, etc.) using constructors. For example,

```
x = 10
println(typeof(x))
```

3.1.2 Boolean Type

- **Bool**: true or false

3.1.3 Characters and Strings

- **Char:** Single Unicode character (e.g., 'a').
- **String:** A sequence of characters (e.g., "Hello, world!").

3.1.4 Abstract Data Types

- **Number:** Abstract type for all numbers.
- **Real:** Abstract type for real numbers (Int, Float64, etc.).
- **AbstractString:** Abstract type for string-like objects.

3.1.5 Composite Types

- **Tuples:** Fixed-size collections of values, e.g., (1, "hello", true).
- **NamedTuples:** Tuples with named fields, e.g., (a=1, b=2).

3.1.6 Collection Types

- **Arrays**:
  - **1D arrays (vectors):** Vector{T} (e.g., [1, 2, 3]).
  - **2D arrays (matrices):** Matrix{T} (e.g., [1 2; 3 4]).
  - **Higher-dimensional arrays:** Array{T, N}.
- **Ranges:**
  - **1:10:** A range from 1 to 10.
  - **1:2:10:** A range with a step of 2.
- **Dictionaries:**
  - **Dict{K, V}:** A collection of key-value pairs, e.g., Dict("a" => 1, "b" => 2).
- **Sets**:
  - **SetT:** An unordered collection of unique elements, e.g., Set([1, 2, 3])

3.1.7 Nothing and Missing:

- **Nothing:** Represents the absence of a value (similar to null in other languages).
- **Missing:** Represents missing data (useful in data analysis).

3.1.8 User-defined Types

- **struct:** Immutable composite types.
- **mutable struct:** Mutable composite types.

3.1.9 typeof()

We can use the `typeof()` function to retrieve the type of a variable

```
# Convert to Integer Types
x = Int8(42)       # Converts 42 to an Int8
y = UInt16(300)    # Converts 300 to a UInt16
```

3.2 Type conversions

3.2.1 Using Constructors

Julia uses constructors to convert a value to a specific type. This is the most common way to perform type casting.

```
convert(Type, value)
```

3.2.2 Using convert Function

The convert function explicitly converts a value to the desired type.

```
# Convert to Int
x = convert(Int, 42.5)       # Converts 42.5 to 42 (truncates
 ↪  the decimal)

# Convert to Float64
y = convert(Float64, 42)     # Converts 42 to 42.0

# Convert to String
s = convert(String, 123)     # Converts 123 to "123"
```

For example

```
# Convert to Int
x = convert(Int, 42.5)       # Converts 42.5 to 42 (truncates
 ↪  the decimal)

# Convert to Float64
y = convert(Float64, 42)     # Converts 42 to 42.0

# Convert to String
s = convert(String, 123)     # Converts 123 to "123"
```

3.2.3 Parsing Strings

To convert a string to a numerical type, use the parse function.

```
# Parse to Integer
x = parse(Int, "123")        # Converts "123" to 123

# Parse to Float64
y = parse(Float64, "3.14")   # Converts "3.14" to 3.14
```

For example,

```
a = 3          # Int
b = 2.5        # Float64
c = a + b      # Automatically promotes `a` to Float64
println(c)     # Output: 5.5
println(typeof(c))  # Output: Float64
```

3.2.4 Automatic Conversion

Some operations perform automatic type promotion or conversion.

```
x = 3    # Int
y = 3.5  # Float64

z = x + y  # `x` is promoted to Float64
println(typeof(z))  # Output: Float64
```

3.3 Type coercion / Type promotion

Type coercion in Julia refers to the process of converting a value from one type to another. This can happen explicitly when you manually convert types or implicitly when Julia promotes values to a common type to ensure compatibility in operations.

Explicit type coercion is performed using the convert function or type constructors. These methods create a new value of the desired type from an existing value.

Implicit type coercion occurs automatically when Julia promotes values to a common type during operations. This is part of Julia's type promotion system.

When two values of different types are used in an operation, Julia promotes them to a common type

```
a, b = promote(2, 3.5)
println(typeof(a))  # Float64
println(typeof(b))  # Float64
```

Julia's promotion rules are designed for precision and performance:

- Integers are promoted to floats when combined with floating-point numbers.
- Smaller numeric types (e.g., Int8) are promoted to larger types (e.g., Int or Float64).
- Complex numbers are preserved when combined with real numbers.

The promote function explicitly promotes multiple values to a common type without performing operations

```
x = 1
x = string(x)
println(typeof(x)) # string

# ERROR!
y = "1"
y = Int(y) # Cannot do
```

We can easily convert numeric types to strings, but errors will occur when you try to do the opposite

```
x = 10
name = "Julia"
```

3.4 Variables

In Julia, variables are declared simply by assigning a value to a name. Julia is dynamically typed, so variables don't need an explicit type declaration, but you can optionally annotate types.

```
x = 10
name = "Julia"
```

3.4.1 Type annotations

You can explicitly specify the type of a variable using a :: type annotation:

```
const pi = 3.14159
```

If the value assigned to the variable doesn't match the specified type, Julia throws a TypeError

3.4.2 Constant Declaration (Immutable Binding / mutable content)

Use the const keyword to declare constants. The type of the constant is inferred from its initial value, and while the value itself can be mutable, the binding cannot be changed

```
const MY_CONSTANT = 42

MY_CONSTANT = 100  # ERROR: invalid redefinition of constant
↪ MY_CONSTANT
```

(**Immutable binding**): "the binding cannot be changed" means that the name of the constant is permanently bound to the initial object it was assigned to, and you cannot reassign the constant to a new value or object. However, if the value itself is a mutable object, you can modify the content of that object.

```
const MY_CONSTANT = 42

MY_CONSTANT = 100  # ERROR: invalid redefinition of constant
↪ MY_CONSTANT
```

(**Mutable content**): If the constant is a mutable object (e.g., an array or dictionary), the content of the object can still be modified, even though the binding to the name is fixed

```
function example()
    local_var = 42  # Local variable
    println(local_var)
end

example()
# println(local_var)  # ERROR: local_var is not defined outside
↪ the function
```

3.4.3 Local / global keywords

The local and global keywords control the scope of variables and determine whether a variable exists within a local scope (e.g., inside a function or loop) or the global scope (outside all functions, at the top level of a module or script).

The global scope refers to variables that are accessible throughout the entire module or script. Variables in the global scope are defined at the top level and are not limited to specific blocks or functions.

By default, variables assigned outside functions or blocks are global.

You can explicitly declare a variable as global inside a function if you want to modify a variable from the global scope.

```
x = 10  # Global variable

function modify_global()
global x  # Declare x as global to modify it
x += 5
end

modify_global()
println(x)  # Output: 15
```

Variables in the global scope can be read within functions without using the global keyword. To modify a global variable inside a function, you must explicitly declare it as global.

By default, variables declared inside loops are local to the loop. To modify a global variable inside a loop, you must use global.

```
x = 0  # Global variable

for i = 1:5
    global x += i  # Explicitly declare x as global to modify it
end

println(x)  # Output: 15
```

The local scope refers to variables that are confined to a specific block, such as a function, loop, or let block. These variables cannot be accessed outside the block where they are defined.

Variables declared inside a function, loop, or block are local by default. You can explicitly use the local keyword to restrict a variable to the current block.

```
function example()
    local_var = 42  # Local variable
    println(local_var)
end

example()
# println(local_var)  # ERROR: local_var is not defined outside
 ↪  the function
```

```
x = 100  # Global variable

function nested_scope()
local x = 50  # Create a local variable named x
println(x)    # Output: 50 (local variable)
end

nested_scope()
println(x)        # Output: 100 (global variable remains
 ↪  unchanged)
```

Use the local keyword for clarity.

3.5 Operators

Arithmetic:

- **Addition:** `+` (e.g., `a + b`)
- **Subtraction:** `-` (e.g., `a - b`)
- **Multiplication:** `*` (e.g., `a * b`)

- **Division:** `/` (e.g., `a / b`)
- **Integer Division:** `div` (e.g., `div(a, b)`)
- **Modulo (Remainder):** `%` (e.g., `a % b`)
- **Floor Division:** `//` (e.g., `a // b`)
- **Power:** ^ (e.g., a^b)
- **Negation:** `-` (e.g., `-a`)

Comparison

- **Equality:** ==
- **Inequality:** != or $\neq$
- **Less than:** $<$
- **Greater than:** $>$
- **Less than or equal to:** <= or $\leqslant$
- **Greater than or equal to:** >= or $\geqslant$

Logical Operators

- **Logical AND:** &&
- **Logical OR:** ||
- **Logical NOT:** !

Bitwise Operators

- **Bitwise AND:** &
- **Bitwise OR:** |
- **Bitwise XOR:** $\veebar$ (veebar in REPL with LaTeX)
- **Bitwise NOT:** $\sim$ (Keyboard tilde)
- **Left Shift:** «
- **Right Shift:** »

Assignment Operators

- **Basic assignment:** =
- **Compound assignments:** +=, -=, *=, /=, ÷=, %=, ^=, &=, |=, $\veebar$=, «=, »=

Inclusive range:

- **Inclusive range:** :
- **Exclusive range:** start:step:end

Element-wise Operators

- **Dot syntax for element-wise operations:** .+, .-, .*, ./, .^, .// (add a . before an operator to make it element-wise)

String Operators

- **String concatenation:** *
- **String interpolation:** $

Ternary

- **Ternary conditional:** $a?b:c$ (c-style)

Concatenation and Splitting Operators

- **Horizontal concatenation:** hcat
- **Vertical concatenation:** vcat
- **Range concatenation:** ... (splatting)

Broadcasting Operator

- **Dot syntax for broadcasting:** .

3.6 Operator precedence

Julia follows PEMDAS

3.7 Ranges

Ranges are compact representations of sequences with a defined step size. They are memory-efficient as they do not explicitly store each element.

```
# Default step of 1
r1 = 1:5
println(collect(r1))  # Output: [1, 2, 3, 4, 5]

# Custom step size
r2 = 1:2:10
println(collect(r2))  # Output: [1, 3, 5, 7, 9]
```

3.8 Concatenation and Splitting Operators

Julia provides functions and operators to concatenate or split arrays, matrices, and strings.

Consider the two arrays

```
a = [1, 2]
b = [3, 4]
```

- **Horizontal Concatenation (hcat):** Combines arrays horizontally (columns).

```
result = hcat(a, b)
println(result)  # Output: [1 3; 2 4]
```

- **Vertical Concatenation (vcat): Combines arrays vertically (rows).**

```
result = vcat(a, b)
println(result)  # Output: [1; 2; 3; 4]
```

- **Concatenation Operator ([...]): Combines arrays inline.**

```
result = [a; b]  # Vertical concatenation
println(result)  # Output: [1; 2; 3; 4]

result = [a b]  # Horizontal concatenation
println(result)  # Output: [1 3; 2 4]
```

- **Strings can be split using split:**

```
x = 42       # Type inferred as Int
y = 3.14     # Type inferred as Float64
```

- **Arrays can be split manually with slicing:**

```
arr = [1, 2, 3, 4, 5]
part1 = arr[1:3]
part2 = arr[4:end]
println(part1)  # Output: [1, 2, 3]
println(part2)  # Output: [4, 5]
```

3.9 Broadcasting

Broadcasting allows you to apply a function or operator element-wise to arrays or other collections. This is achieved using the dot operator (.).

```
x = 3    # Int
y = 3.5  # Float64

z = x + y  # `x` is promoted to Float64
println(typeof(z))  # Output: Float64
```

You can also broadcast custom functions:

```
f(x, y) = x + y^2
result = f.(a, b)
println(result)  # Output: [17, 27, 39]
```