Programming in Julia

Nathan Warner

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

```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

```julia
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

```julia
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.

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

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

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

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

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

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

```julia
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).

```julia
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.

```julia
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.

```julia
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.

```julia
import Random: rand, shuffle!  # Import `rand` and `shuffle!`

x = rand(1:10, 5)
```

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.

```julia
import Base: +

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

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

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,

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

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

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

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

3.2.2 Using convert Function

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

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

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(Type, string)
```

For example,

```
# 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
```

3.2.4 Automatic Conversion

Some operations perform automatic type promotion or conversion.

```julia
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
```

3.3 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

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

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

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

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

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

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

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:

```
y::Int = 42    # y must always hold an Int
z::Float64 = 3.14
```

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 pi = 3.14159
```

(**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

```
const MY_ARRAY = [1, 2, 3]

# Modify the content of the array (allowed)
push!(MY_ARRAY, 4)
println(MY_ARRAY)  # Output: [1, 2, 3, 4]

# Reassign the constant to a new object (not allowed)
MY_ARRAY = [5, 6, 7]  # ERROR: invalid redefinition of constant
    ↪  MY_ARRAY
```

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.

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

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

```julia
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:**

```
str = "Hello,World,Julia"
result = split(str, ",")
println(result)  # Output: ["Hello", "World", "Julia"]
```

- **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 (.).

```
a = [1, 2, 3]
b = [4, 5, 6]

# Element-wise addition
c = a .+ b
println(c)  # Output: [5, 7, 9]

# Broadcasting with scalar
d = a .* 2
println(d)  # Output: [2, 4, 6]
```

You can also broadcast custom functions:

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

3.10 Comments

Julia uses the pound sign for comments

```julia
# Comment
```

3.11 The in operator

The in operator works similar to Python

```julia
a = [1,2,3]
println(1 in a) # True
println(5 in a) # False

if 2 in a
    println("2 is in a")
end

# Error: Use occursin
b = "Julia"
println("J" in b)
println("x" in b)
```

Functions

Functions in Julia are first-class, and can therefore be higher order. In programming, first-class functions refer to functions that are treated as first-class citizens. This means they are treated like any other value or variable in the language

- **Assignment:** Functions can be assigned to variables.
- **Passing as Arguments:** Functions can be passed as arguments to other functions.
- **Returning from Other Functions:** Functions can be returned as the result of other functions.
- **Storage in Data Structures:** Functions can be stored in lists, dictionaries, or other data structures.

If functions in a language lack any of these properties, they are not first-class. Such functions are sometimes referred to as second-class functions or non-first-class functions

A higher-order function is a function that satisfies at least one of the following criteria:

- Takes another function as an argument
- Returns a function as its result

A function that neither takes a function as an argument nor returns a function is not a higher-order function. These are sometimes referred to as first-order functions or regular functions

Functions in Julia are created with the keyword *function* and ended with the keyword *end*

```julia
function f()
    ...
end
```

Functions can also take arguments and access globals

```julia
x = 5

function f(arg1)
    println(x)
    println(arg1)
end
```

4.1 Returning

We can specify a return value with the keyword *return*, but if it is omitted, Julia will return the last evaluated expression. Observe the following function

```julia
function f()
    5 + 10
end

x = f()
println(x) # 15
```

4.2 Short form (single-line) functions

For simple functions, you can define them in a single line.

```julia
fn(x,y) = x + y
```

4.2.1 Lambdas

Anonymous functions are unnamed functions and are often used as arguments to other functions.

```julia
(x,y) -> x+y
```

4.3 Multiple Dispatch

Multiple dispatch is a core feature of Julia that allows the selection of a method to execute based on the types of all arguments passed to a function, not just the first argument (as in single dispatch). This makes Julia highly flexible and efficient for polymorphism.

```julia
# Define methods for different type combinations
function calculate(a::Int, b::Int)
    a + b
end

function calculate(a::Float64, b::Float64)
    a * b
end

function calculate(a::Int, b::Float64)
    a - b
end

# Call the function
println(calculate(3, 4))       # Output: 7 (Int + Int)
println(calculate(3.0, 4.0))   # Output: 12.0 (Float64 * Float64)
println(calculate(3, 4.0))     # Output: -1.0 (Int - Float64)
```

Control flow

5.1 The let block

In Julia, the let block is used to create a new local scope. This allows you to define variables whose lifetime is limited to the block, even if there are variables with the same name in the surrounding scope.

```
let [variables = initial_values]
    ...
end
```

For example,

```
let a=5,b=10
    println(a,"\n", b)
end

println(a, "\n", b) # Error
```

If no variable is declared in the let block, it still serves as a local scope:

```
let
    y = 5  # Only exists within this block
    println("y inside let: $y")
end

# println(y)  # This would cause an error since y is not defined outside
```

5.2 Conditional blocks (if-elseif-else)

Used for conditional execution of code.

```
x = 10
if x > 0
    println("Positive")
    elseif x == 0
    println("Zero")
else
    println("Negative")
end
```

5.3 For loops

Used for iterating over collections, ranges, or custom iterables.

```
for i in 1:5
    println(i)  # Output: 1, 2, 3, 4, 5
end
```

Iterating Over Collections

```
arr = ["apple", "banana", "cherry"]
for fruit in arr
    println(fruit)
end
```

5.4 While loops

Repeats a block of code as long as a condition is true.

```
x = 5
while x > 0
    println(x)
    x -= 1
end
```

5.5 Short-Circuit Control

```
x = 5
x > 0 && println("Positive")  # Executes if x > 0
x < 0  println("Not Negative")  # Executes if x >= 0
```

5.6 Begin blocks

Groups multiple statements into a single block, useful for one-liners or compound expressions.

```
begin
    x = 1
    y = 2
    println(x + y)  # Output: 3
end
```

5.7 Anonymous Blocks

You can use anonymous blocks for scoping purposes without naming them explicitly.

```
x = 10
println("Before block: $x")
# New block
x = begin
    y = 20
    z = y + 10
    z  # Return value
end
println("After block: $x")  # Output: 30
```

5.8 Functions as Control Blocks (do blocks)

```
map(x -> x^2, [1, 2, 3])  # Single-line lambda
# Equivalent with a do block
map([1, 2, 3]) do x
    x^2
end
```

Strings

In Julia, strings are sequences of characters, and they are used to represent text. Strings are immutable, meaning once a string is created, it cannot be altered directly

6.1 String Operations

In general, you can't perform mathematical operations on strings, even if the strings look like numbers. But there are two exceptions, * and ^.

The * operator performs string concatenation, which means it joins the strings by linking them end-to-end. For example:

The ^ operator also works on strings; it performs repetition. For example, "Spam"^3 is "SpamSpamSpam". If one of the values is a string, the other has to be an integer.

Use $ to insert variables or expressions into strings.

```
name = "Julia"
println("Welcome to $name programming!")  # Output: "Welcome to Julia programming!"
```

6.2 Indexing and Slicing

Julia strings are 1-indexed, and you can access individual characters using square brackets.

```
str = "Julia"
first_char = str[1]  # 'J'
substring = str[2:4]  # "uli"
```

6.3 Into to string Functions

Julia provides a variety of built-in functions to work with strings:

- **Length**:

  ```
  length("hello")
  ```

- **Find and replace**:

  ```
  str = "apple pie"
  new_str = replace(str, "apple" => "cherry")  # "cherry pie"
  ```

- **Splitting and joining**:

```
split_str = split("a,b,c", ",")  # ["a", "b", "c"]
joined_str = join(["a", "b", "c"], "-")  # "a-b-c"
```

- **Searching**:

```
occursin("Julia", "Hello, Julia!")  # true
findfirst("Julia", "Hello, Julia!")  # 8 (start index of the
↪  match)
```

6.4 Unicode Support

Julia strings are fully Unicode-compliant, meaning they support characters from virtually all languages.

6.5 Mutability

Strings in Julia are immutable, so modifying an existing string directly is not allowed. However, you can create new strings based on the original.

```
str = "Hello"
str[1] = 'J'  # Error: Strings are immutable
```

Instead, create a new string:

```
new_str = "J" * str[2:end]  # "Jello"
```

6.6 Multiline Strings

For strings spanning multiple lines, use triple double quotes (""").

```
multiline_str = """
This is a
multiline string.
"""
```

6.7 Raw Strings

Use raw"..." to create raw strings where escape sequences like \n are not processed.

```
raw_str = raw"Line 1\nLine 2"  # "Line 1\\nLine 2"
```

6.8 String Types

- **String:** The default string type in Julia.
- **SubString:** A view of a part of a string, created using slicing. More efficient for substring operations.

```
s = "Hello, Julia!"
sub_s = SubString(s, 8, 12)  # "Julia"
```

Use String for general-purpose text and SubString for efficient substring manipulation

6.9 String functions

Julia provides a rich set of functions for working with strings. Below is a comprehensive list categorized for easier reference.

6.9.1 Basic String Information

- length(s) - Returns the number of characters in the string s.
- sizeof(s) - Returns the number of bytes in the string s.
- isequal(s1, s2) - Checks if two strings are exactly the same.
- isempty(s) - Returns true if the string s is empty

6.9.2 String Manipulation

- join(strings, delim) - Joins strings in an array with delim as the separator.
- replace(s, pattern => replacement) - Replaces all occurrences of pattern with replacement.
- strip(s) - Removes leading and trailing whitespace.
- lstrip(s) - Removes leading whitespace.
- rstrip(s) - Removes trailing whitespace.
- chomp(s) - Removes the trailing newline character.
- pad(s, n; lpad=n1, rpad=n2) - Pads a string to a specified length.

6.9.3 Searching and Matching

- occursin(substr, s) - Checks if substr exists in s.
- findfirst(substr, s) - Finds the first occurrence of substr in s and returns its index.
- findlast(substr, s) - Finds the last occurrence of substr in s.
- findall(substr, s) - Returns all indices where substr occurs in s.
- startswith(s, prefix) - Checks if s starts with prefix.
- endswith(s, suffix) - Checks if s ends with suffix.

6.9.4 Splitting and Joining

- split(s, delim) - Splits s into an array of substrings using delim.
- rsplit(s, delim) - Splits s into substrings from the right.
- split(s) - Splits s by whitespace.
- join(strings, delim) - Joins an array of strings into one string using delim.

6.9.5 String Transformation

- uppercase(s) - Converts all characters in s to uppercase.
- lowercase(s) - Converts all characters in s to lowercase.
- titlecase(s) - Converts the first character of each word to uppercase.
- capitalize(s) - Converts the first character of s to uppercase.
- replace(s, pattern => replacement) - Replaces all occurrences of pattern with replacement.

6.9.6 Character Inspection

- isuppercase(c) - Checks if a character c is uppercase.
- islowercase(c) - Checks if a character c is lowercase.
- isletter(c) - Checks if a character c is a letter.
- isdigit(c) - Checks if a character c is a digit.
- iswhitespace(c) - Checks if a character c is a whitespace.
- isascii(c) - Checks if a character c is an ASCII character.
- isprint(c) - Checks if a character c is printable.
- isalnum(c) - Checks if a character c is alphanumeric.

Splatting and slurping

7.1 Splatting

The ellipsis (...) can unpack elements from collections like arrays or tuples when calling a function or constructing another collection.

```
function sum_numbers(a, b, c)
    return a + b + c
end

nums = [1, 2, 3]
result = sum_numbers(nums...)  # Equivalent to sum_numbers(1, 2, 3)
println(result)  # 6
```

7.2 Slurping

In function definitions, ... allows a function to accept a variable number of arguments, known as varargs.

```
function greet(names...)
    for name in names
        println("Hello, $name!")
    end
end

greet("Alice", "Bob", "Charlie")
# Output:
# Hello, Alice!
# Hello, Bob!
# Hello, Charlie!
```

Arrays

Arrays in Julia are a core data structure used to store collections of elements in an ordered and indexed manner. They can have multiple dimensions and are highly efficient, making them suitable for numerical computations and general-purpose programming.

8.1 Creating Arrays

- **1D Array (Vector):**

```julia
arr = [1, 2, 3, 4]  # Creates a 1D array with elements 1, 2, 3, and 4
```

- **2D Array (Matrix):**

```julia
mat = [1 2; 3 4]  # Creates a 2x2 matrix
```

- **Using Array constructor**:

```julia
arr = Array{Int64}(undef, 5)  # Creates a 1D array of integers with 5 elements, uninitialized
mat = Array{Float64}(undef, 3, 3)  # Creates a 3x3 matrix of uninitialized floats
```

- **Zeros and Ones:**

```julia
zeros(3, 3)  # 3x3 matrix filled with zeros
ones(4)      # 1D array with four elements, all ones
```

8.2 Accessing elements

Julia uses 1-based indexing

```julia
arr = [10, 20, 30]
println(arr[1])  # Access the first element (output: 10)

subarr = arr[1:2]  # Creates a subarray with elements 10 and 20

mat = [1 2; 3 4]
println(mat[1, 2])  # Accesses the element at row 1, column 2 (output: 2)
```

8.3 Modifying arrays

```
arr[2] = 25  # Modifies the second element of arr to 25

push!(arr, 40)  # Appends 40 to the end of the array
pop!(arr)  # Removes the last element from the array
```

8.4 Properties of arrays

```
println(size(mat))  # Returns the dimensions of the array
println(length(arr))  # Returns the total number of elements
println(eltype(arr))  # Returns the type of elements stored in the array
```

8.5 Broadcasting

```
arr = [1, 2, 3]
result = arr .+ 10  # Adds 10 to each element (output: [11, 12, 13])
```

8.6 Key features

- Arrays in Julia are mutable, meaning you can change their contents.
- Arrays are column-major (like Fortran and MATLAB), which impacts performance for linear algebra operations.
- Built-in functions like map, reduce, and comprehensions make array manipulation concise and powerful.

8.7 Array functions

8.7.1 Array Creation

- Array - Constructs an uninitialized array.
- zeros - Creates an array filled with zeros.
- ones - Creates an array filled with ones.
- fill - Creates an array filled with a specific value.
- rand - Creates an array with random values.

- randn - Creates an array with random values from the normal distribution.
- reshape - Changes the shape of an array without changing its data.
- hcat - Horizontally concatenates arrays.
- vcat - Vertically concatenates arrays.
- hvcat - Concatenates arrays in multiple dimensions.
- repeat - Repeats an array in specified dimensions.
- range - Creates an array of evenly spaced numbers.
- collect - Converts an iterable (like a range) into an array.

8.7.2 Array Properties

- size - Returns the dimensions of the array.
- length - Returns the total number of elements in the array.
- ndims - Returns the number of dimensions.
- eltype - Returns the type of elements stored in the array.
- axes - Returns the valid indices for each dimension.
- eachindex - Iterates efficiently over all indices.
- stride - Returns the memory stride for a given dimension.
- strides - Returns the memory strides for all dimensions.
- isempty - Checks if the array is empty.

8.7.3 Array Indexing and Slicing

- getindex - Accesses array elements (arr[i]).
- setindex! - Sets array elements (arr[i] = value).
- view - Creates a view (a lightweight subset) of an array.
- @view - A macro for creating views.
- slice - Extracts a slice of an array.

8.7.4 Array Modification

- push! - Appends an element to the end of a 1D array.
- pop! - Removes the last element of a 1D array.
- append! - Appends all elements of one array to another.
- insert! - Inserts an element at a specific position.
- deleteat! - Deletes elements at specified indices.
- splice! - Removes elements and optionally replaces them.
- empty! - Removes all elements from an array.
- resize! - Resizes an array to a specified size.

8.7.5 Combining and Splitting Arrays

- cat - Concatenates arrays along a specified dimension.
- hcat, vcat, hvcat - Horizontally, vertically, or multi-dimensionally concatenate arrays.
- splitdims - Splits a multi-dimensional array into its slices along a specified dimension.
- reshape - Reshapes an array into a different dimensionality.
- permutedims - Permutes the dimensions of an array.

8.7.6 Mapping and Broadcasting

- map - Applies a function to each element of an array.
- broadcast - Applies a function element-wise to arrays.
- @. - Macro for broadcasting every operation in an expression.
- map! - In-place version of map.
- broadcast! - In-place version of broadcast.

8.7.7 Statistical Functions

- sum - Computes the sum of all elements.
- prod - Computes the product of all elements.
- mean - Computes the mean of all elements.
- std - Computes the standard deviation.
- var - Computes the variance.
- maximum - Finds the maximum value.
- minimum - Finds the minimum value.
- argmax - Returns the index of the maximum value.
- argmin - Returns the index of the minimum value.
- count - Counts the number of elements satisfying a condition.

8.7.8 Array Manipulation

- reverse - Reverses the order of elements.
- sort - Sorts the elements of an array.
- sort! - Sorts an array in-place.
- shuffle - Randomly shuffles the elements.
- permute! - Rearranges elements in a specific order.
- unique - Returns unique elements of an array.
- filter - Filters elements based on a condition.
- findall - Finds all indices where a condition is true.
- findfirst - Finds the first index where a condition is true.
- findlast - Finds the last index where a condition is true

8.7.9 Matrix-Specific Functions

- transpose - Computes the transpose of a matrix.
- adjoint - Computes the conjugate transpose.
- det - Computes the determinant of a square matrix.
- inv - Computes the inverse of a square matrix.
- rank - Computes the rank of a matrix.
- trace - Computes the trace of a square matrix.
- diag - Extracts or creates a diagonal matrix.
- tril - Extracts the lower triangular part of a matrix.
- triu - Extracts the upper triangular part of a matrix.

8.7.10 Linear Algebra Functions

- eigen - Computes eigenvalues and eigenvectors.
- svd - Computes the singular value decomposition.
- qr - Computes the QR factorization.
- lu - Computes the LU decomposition.
- chol - Computes the Cholesky decomposition.
- pinv - Computes the pseudo-inverse.

Other containers

9.1 Tuples

Immutable ordered collections of elements

```
t = (1, "hello", 3.14)
println(t[2])  # Access elements by index (1-based)
```

9.2 Named Tuples

Tuples with named fields for each element.

```
nt = (a = 1, b = "hello", c = 3.14)
println(nt.b)  # Access elements by name
```

9.3 Dictionaries

Key-value pairs, like hash maps in other languages.

```
dict = Dict("a" => 1, "b" => 2)
dict["c"] = 3  # Add a new key-value pair
println(dict["b"])  # Access value by key
```

9.4 Sets

Unordered collections of unique elements.

```
s = Set([1, 2, 2, 3])
push!(s, 4)  # Add an element
println(3 in s)  # Check membership
```

9.5 Ranges

Compact representations of sequences of numbers.

```
 r = 1:2:10  # Start at 1, increment by 2, stop at 10
println(collect(r))  # Convert to an array
```

9.6 Sparse Arrays

Efficient storage for arrays with mostly zero elements.

```
using SparseArrays
sp = sparse([1, 0, 0; 0, 2, 0; 0, 0, 3])
println(sp[2, 2])  # Access elements
```

9.7 Deques (from the DataStructures package)

Double-ended queues, allowing efficient insertion and deletion from both ends.

```
using DataStructures
dq = Deque([1, 2, 3])
push!(dq, 4)  # Add to the end
pushfirst!(dq, 0)  # Add to the front
println(dq)
```

9.8 Stores elements with associated priorities.

```
using DataStructures
pq = PriorityQueue()
enqueue!(pq, "task1", 1)  # Add element with priority
enqueue!(pq, "task2", 0)
println(dequeue!(pq))  # Returns "task2" (lowest priority first)
```

Iterators

iterators provide a way to traverse through elements of a collection (or any iterable object) in a systematic way

For an object to be iterable in Julia, it must implement the iterate function. The iterate function acts as the foundation for traversing elements of a collection.

```
iterate(iterable, state)
```

- **iterable:** The object to iterate over.
- **state:** Tracks the current position in the iteration.
- **Returns:** A tuple (value, new_state) if there are elements remaining, or nothing when iteration is complete

Iteration is automatically handled by Julia's built-in constructs, like loops:

```
for x in collection
    println(x)
end
```

You can make any type iterable by defining the iterate function.

```
 struct Counter
    max::Int
end

function Base.iterate(counter::Counter, state=1)
    if state > counter.max
        return nothing
    end
    return (state, state + 1)
end

counter = Counter(5)
for x in counter
    println(x)
end
```

Structs

In Julia, a struct (short for ”structure”) is a composite data type that groups together related variables (called fields) under one entity. It is a way to create custom types in Julia, allowing you to organize data more effectively and work with it in a structured manner.

11.1 Key features

- **Immutable by Default:** Fields in a struct cannot be modified once the struct is created. This immutability ensures safety and efficiency in operations.
- **Custom Types:** Structs allow you to define your own types, which can be used in functions, arrays, or as fields in other structs.
- **Can Be Made Mutable:** By using mutable struct, the fields can be modified after the struct is instantiated.

11.2 Creating structs

```julia
struct StructName
    field1::DataType1
    field2::DataType2
    # Add as many fields as needed
end
```

For example,

```julia
struct Point
    x::Float64
    y::Float64
end
```

To create an instance of a struct:

```julia
p = Point(3.0, 4.0)

# Access fields with the dot notation
println(p.x)  # Output: 3.0
println(p.y)  # Output: 4.0
```

11.3 Mutable Structs

Use mutable struct when you want to modify fields after the struct is created

```julia
mutable struct PointMutable
    x::Float64
    y::Float64
end

p = PointMutable(3.0, 4.0)
p.x = 5.0  # Now p.x is updated to 5.0
```

11.4 Structs Without Field Type Annotations

Fields can omit type annotations, making them more flexible but less type-specific

```julia
struct FlexiblePoint
    x
    y
end

p = FlexiblePoint(3.0, "hello")
```

11.5 Default Values

Julia does not directly support default field values in struct definitions. However, you can achieve this using constructors

```julia
struct PointDefault
    x::Float64
    y::Float64
    PointDefault(x::Float64, y::Float64 = 0.0) = new(x, y)
end

p1 = PointDefault(3.0)  # y defaults to 0.0
p2 = PointDefault(3.0, 4.0)  # y is explicitly set to 4.0
```

11.6 Parametric Structs

Parametric structs allow you to define types that work with different data types

```julia
struct PointParametric{T}
    x::T
    y::T
end

p = PointParametric{Int}(3, 4)
q = PointParametric{Float64}(3.0, 4.0)
```

Here, {T} is a type parameter that makes the struct generic.

11.7 Abstract types and Subtypes

Julia does not have traditional object-oriented inheritance like some other languages (e.g., Python, Java, C++). However, Julia supports hierarchical relationships using its type system. Specifically, Julia enables inheritance via abstract types and parametric types.

11.7.1 Abstract types

In Julia, abstract types are part of the type hierarchy and are used to define a blueprint for other types. They are placeholders in the type system, allowing you to group related types without specifying implementation details. Abstract types cannot be instantiated directly but serve as parent types for concrete types.

- **Abstract:** You cannot create instances of an abstract type.
- **Purpose:** Abstract types define a common interface or grouping for a set of related types.
- **Inheritance:** Concrete types or other abstract types can inherit (subtype) from an abstract type using the <: symbol.

```julia
abstract type AbstractName end

# For example
abstract type Shape end
```

Here, Shape is an abstract type that can act as a parent for all kinds of shapes like circles, squares, and triangles.

11.7.2 Concrete types

Concrete types inherit from abstract types by using the <: symbol.

```julia
abstract type Shape end  # Abstract type

# Concrete subtypes
struct Circle <: Shape
    radius::Float64
end

struct Rectangle <: Shape
    width::Float64
    height::Float64
end
```

11.8 Usage of Abstract Types

Abstract types are often used with functions and multiple dispatch to define methods that can operate on a group of related types.

```julia
 abstract type Shape end

struct Circle <: Shape
    radius::Float64
end

struct Rectangle <: Shape
    width::Float64
    height::Float64
end

# Define a generic method for `Shape`
area(::Shape) = throw(NotImplementedError("area method not implemented for this shape"))

# Specialize the method for each subtype
area(c::Circle) =   * c.radius^2
area(r::Rectangle) = r.width * r.height

# Usage
circle = Circle(5.0)
rect = Rectangle(4.0, 3.0)

println(area(circle))  # Output: 78.53981633974483
println(area(rect))    # Output: 12.0
```

Julia's type hierarchy has a top-level abstract type called Any, from which all other types derive.

```julia
abstract type Animal end  # Abstract type

struct Dog <: Animal
    name::String
end

struct Cat <: Animal
    name::String
end

# Animal is a subtype of Any
println(Animal <: Any)  # Output: true
```

11.9 Abstract Types and Parametric Types

Abstract types can also be parametric, which means they accept type parameters. This allows for more generic and flexible type definitions.

```julia
abstract type Container{T} end  # Abstract type with a type parameter T

struct Box{T} <: Container{T}
    value::T
end

# Usage
int_box = Box{Int}(42)
string_box = Box{String}("Hello")

println(typeof(int_box))  # Output: Box{Int64}
println(typeof(string_box))  # Output: Box{String}
```

Memory

12.1 Stack Allocation

Used for small, fixed-size objects like immutable values and local variables. Allocation and deallocation are very fast because they operate on the call stack

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

Here, a and b are allocated on the stack.

12.2 Heap Allocation

Used for larger or dynamically sized objects, such as arrays and mutable structs Allocation is slower than stack allocation because it uses the heap memory, which requires garbage collection to free unused memory

```
arr = [1, 2, 3]  # Array is allocated on the heap
```

12.3 Garbage collection

Julia uses automatic garbage collection to free memory that is no longer in use. The garbage collector periodically identifies objects in heap memory that are no longer referenced and reclaims that memory.

You can trigger garbage collection manually using

```
GC.gc()
```

12.4 Memory Views and Sharing

Julia avoids unnecessary copying by using views for slicing arrays. Instead of creating a new array, a view provides a lightweight reference to the original array's data.

```
arr = [1, 2, 3, 4]
v = @view arr[1:2]  # Does not allocate new memory
```

12.5 Static arrays

For small, fixed-size arrays, use StaticArrays.jl to avoid heap allocations.

```
using StaticArrays
v = SVector(1.0, 2.0, 3.0)
```

12.6 Advanced: Memory Buffers and Pointers

Julia allows low-level memory management through buffers and pointers. For example

```
buf = Base.Libc.malloc(1024)  # Allocate 1024 bytes manually
Base.Libc.free(buf)           # Free the allocated memory
```

12.7 References

In Julia, references play a key role in determining how objects are handled in memory, particularly for mutable and immutable objects. Understanding references is crucial for working effectively with Julia's memory model, especially for mutable objects like arrays and mutable struct

A reference is essentially a pointer to an object's location in memory. When you assign or pass objects in Julia, whether by value or by reference depends on the type of object.

12.7.1 Immutable types

Immutable types are passed by value. When you assign or pass an immutable object (e.g., Int, Float64, or a struct without mutable), Julia creates a copy of the object rather than a reference.

```
x = 5  # `x` holds the value directly
y = x  # `y` is a copy of `x`
y += 1
println(x)  # 5 (unchanged)
println(y)  # 6 (modified copy)
```

12.7.2 Mutable Types

Mutable types are passed by reference. When you assign or pass a mutable object (e.g., arrays or mutable struct), Julia creates a reference to the same memory location. Any modifications affect the original object.

```
arr = [1, 2, 3]  # Array is mutable
arr_copy = arr    # `arr_copy` references the same array
arr_copy[1] = 10  # Modify through `arr_copy`
println(arr)      # [10, 2, 3] (original is modified)
```

12.7.3 Copying Mutable Objects

If you need to modify a mutable object without affecting the original, you must create a deep or shallow copy.

- **Shallow Copy (copy):** Creates a new container but references the same inner objects.

```
a = [[1, 2], [3, 4]]
b = copy(a)
b[1][1] = 99
println(a)  # [[99, 2], [3, 4]] (inner objects are shared)
```

- **Deep Copy (deepcopy):** Recursively copies all objects, creating completely independent copies.

```
a = [[1, 2], [3, 4]]
c = deepcopy(a)
c[1][1] = 99
println(a)  # [[1, 2], [3, 4]] (completely independent)
```

12.7.4 References in Functions

Mutable objects are passed by reference to functions, so any modifications made within the function affect the original object

```
function modify_array(arr)
    arr[1] = 100
end

x = [1, 2, 3]
modify_array(x)
println(x)  # [100, 2, 3] (modified in-place)
```

If you want to avoid modifying the original, you can pass a copy

```
modify_array(copy(x))
```

12.7.5 Ref for Explicit References

The Ref type in Julia provides a way to create explicit references to values, even for immutable types. This is useful for interfacing with low-level code or when you want mutable behavior for a normally immutable value.

```
r = Ref(5)  # Create a reference to an immutable value
println(r[])  # Access the value (5)
r[] = 10     # Modify the value
println(r[])  # Access the updated value (10)
```

The syntax r[] in Julia is used to access or modify the value stored in a Ref object. The Ref type in Julia is essentially a container for a single value, allowing you to treat the contained value as mutable, even if it's normally immutable (like numbers).

Working with files

13.1 The basics

Files are accessed using their file paths, which can be absolute or relative. You specify how the file will be opened with the following modes

- **"r":** Read-only.
- **"w":** Write-only (truncates if the file exists).
- **"a":** Append mode (writes at the end of the file).
- **"r+":** Read and write.
- **"w+":** Read and write (truncates if the file exists).
- **"a+":** Read and write (writes at the end of the file).

13.2 Reading Files

Reading Entire File

```julia
content = read("example.txt", String)
println(content)
```

Line-by-Line Reading

```julia
open("example.txt", "r") do file
    for line in eachline(file)
        println(line)
    end
end
```

13.3 Writing Files

Writing Strings

```julia
write("output.txt", "Hello, Julia!")
```

Appending to a File

```julia
open("output.txt", "a") do file
    write(file, "\nAdding a new line.")
end
```

13.4 Checking File Properties

- Does the file exist?

```
if isfile("example.txt")
    println("File exists!")
else
    println("File does not exist.")
end
```

- Is it a directory?

```
if isdir("my_directory")
    println("It's a directory.")
end
```

- Get file size

```
size = filesize("example.txt")
println("File size: $size bytes")
```

13.5 Directory Operations

- List Files in Directory

```
for entry in readdir(".")
    println(entry)
end
```

- Create Directory

```
mkdir("new_directory")
```

- Remove File or Directory

```
rm("example.txt")  # Removes a file
rm("empty_directory", recursive=true)  # Removes a directory and its contents
```

13.6 Example: Copying a File

```
open("source.txt", "r") do src
    open("destination.txt", "w") do dest
        write(dest, read(src, String))
    end
end
```

Error handling

14.1 Throwing errors

To signal an error, you use the throw function. Julia provides several built-in error types, and you can define custom error types if needed.

- **ErrorException:** A general-purpose error.
- **ArgumentError:** Raised when a function argument is invalid.
- **DomainError:** Raised when a value is outside the domain of a function.
- **MethodError:** Raised when a method is not found for the given arguments.
- **KeyError:** Raised when a key is not found in a dictionary.
- **BoundsError:** Raised when an index is out of bounds.
- **DivideError**: Throw to prevent division by zero

Note: Division by zero yields `Inf (infinity)` in Julia, no exception will be thrown automatically

```
function divide(a, b)
    if b == 0
        throw(DivideError())  # Raise a DivideError
    end
        return a / b
end
```

14.2 Handling Errors

You use the try-catch block to handle errors. If an error occurs in the try block, control is transferred to the appropriate catch block.

```
try
    # Code that might throw an error
catch e
    # Handle the error
end
```

14.3 Finally Block

The finally block runs after the try and catch blocks, regardless of whether an error occurred. It is typically used for cleanup operations.

```julia
function file_operations()
    file = open("example.txt", "w")
    try
        write(file, "Hello, world!")
    catch e
        println("Caught an error: $e")
    finally
        close(file)  # Ensures the file is closed
    end
end
```

14.4 Custom Errors

You can define custom error types by creating a struct that inherits from Exception

```julia
struct CustomError <: Exception
    msg::String
end

function risky_function()
    throw(CustomError("This is a custom error"))
end

try
    risky_function()
catch e
    println("Caught error: $e")
end
```

Regular expressions

Regular expressions (regex) in Julia are a powerful tool for matching and manipulating strings based on patterns. Julia's regular expressions are based on the Perl-compatible Regular Expressions (PCRE) library, which provides rich functionality for pattern matching.

15.1 Basic Syntax

In Julia, regular expressions are written as strings prefixed with the *r* macro

```
r"pattern"
```

The *r* before the quotes signifies that the string contains a regular expression.

```
r"\d+"  # Matches one or more digits
r"\s"   # Matches any whitespace character
r"[A-Za-z]+"  # Matches one or more alphabetic characters
```

15.2 Key Functions for Regex in Julia

- **occursin:** Checks if a pattern exists in a string.

```
occursin(r"\d+", "There are 123 apples.")  # true
```

- **match:** Returns the first match of the pattern in a string.

```
m = match(r"\d+", "There are 123 apples.")
println(m.match)  # "123"
```

- **eachmatch:** Returns an iterator of all matches in a string.

```
for m in eachmatch(r"[a-z]+", "Julia is fun")
    println(m.match)
end
# Output:
# "ulia"
# "is"
# "fun"
```

- **replace:** Replaces parts of a string that match the pattern.

```
replace("123-456", r"\d", "X")  # "XXX-XXX"
```

- **split:** Splits a string based on a pattern.

```
split("one, two, three", r",\s*")  # ["one", "two", "three"]
```

- **ismatch:** Checks if the entire string matches the pattern.

```
ismatch(r"^\d+$", "12345")  # true
ismatch(r"^\d+$", "123abc")  # false
```

15.3 Flags

Flags modify the behavior of the regex.

- **i:** Case-insensitive matching.
- **m:** Multi-line mode (allows ^ and $ to match the start and end of lines).
- **s:** Dot-all mode (makes . match newline characters).

```
r"hello"i  # Case-insensitive match for "hello"
```

15.4 RegexMatch Object

The match and eachmatch functions return RegexMatch objects. These contain details about the match:

- **m.match:** The full match.
- **m.captures:** Captured groups.

```
m = match(r"(\d+)-(\d+)", "123-456")
println(m.match)    # "123-456"
println(m.captures) # ["123", "456"]
```