РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖ|БЫ НАРОДОВ

Факультет физико-математических и естественных наук

Кафедра теории вероятностей и кибербезопасности

Лабораторная работа № 1

Студент: Абд эль хай Мохамад
Номер: 1032215163
Група: НПИбд-01-21

Москва 2025

In [1]:
```
typeof(5)
```

Out[1]:
```
Int64
```

In [2]:
```
typeof(π)
```

Out[2]:
```
Irrational{:π}
```

In [3]:
```
for T in
    [Int8,Int16,Int32,Int64,Int128,UInt8,UInt16,UInt32,UInt64,UInt128,Float3
    println("$(lpad(T,7)): [$(typemin(T)),$(typemax(T))]")
end
```

```
  Int8: [-128,127]
 Int16: [-32768,32767]
 Int32: [-2147483648,2147483647]
 Int64: [-9223372036854775808,9223372036854775807]
Int128: [-170141183460469231731687303715884105728,170141183460469231731687
03715884105727]
  UInt8: [0,255]
 UInt16: [0,65535]
 UInt32: [0,4294967295]
 UInt64: [0,18446744073709551615]
UInt128: [0,340282366920938463463374607431768211455]
Float32: [-Inf,Inf]
```

In [4]:
```
Int8(-128)
```

Out[4]:
```
-128
```

In [5]:
```
Int8(-129)
```

```
InexactError: trunc(Int8, -129)

Stacktrace:
 [1] throw_inexacterror(::Symbol, ::Vararg{Any})
   @ Core ./boot.jl:750
 [2] checked_trunc_sint
   @ ./boot.jl:764 [inlined]
 [3] toInt8
   @ ./boot.jl:779 [inlined]
 [4] Int8(x::Int64)
   @ Core ./boot.jl:889
 [5] top-level scope
   @ In[5]:1
```

In [6]:
```
?promote
```

```
search: promote promote_rule promote_type promote_shape permute! precompile
```

Out[6]:
```
promote(xs...)
```

Convert all arguments to a common type, and return them all (as a tuple). If no arguments can be converted, an error is raised.

See also: `promote_type`, `promote_rule`.

Examples

```
jldoctest
julia> promote(Int8(1), Float16(4.5), Float32(4.1))
(1.0f0, 4.5f0, 4.1f0)

julia> promote_type(Int8, Float16, Float32)
Float32

julia> reduce(Base.promote_typejoin, (Int8, Float16, Float32))
Real

julia> promote(1, "x")
ERROR: promotion of types Int64 and String failed to change any argu
ments
[...]

julia> promote_type(Int, String)
Any
```

In [7]:
```
typeof(promote(Int8(127),Int16(128),Int32(129)))
```

Out[7]:
```
Tuple{Int32, Int32, Int32}
```

In [8]:
```
?Char()
```

Out[8]:
```
Char(c::Union{Number,AbstractChar})
```
`Char` is a 32-bit `AbstractChar` type that is the default representation of characters in Julia. `Char` is the type used for character literals like `'x'` and it is also the element type of `String`.

In order to losslessly represent arbitrary byte streams stored in a `String`, a `Char` value may store information that cannot be converted to a Unicode codepoint — converting such a `Char` to `UInt32` will throw an error. The `isvalid(c::Char)` function can be used to query whether `c` represents a valid Unicode character.

In [9]:
```
function f(x)
    √x
end
```

Out[9]:
```
f (generic function with 1 method)
```

In [10]:
```
f(4)
```

Out[10]:
```
2.0
```

In [11]:
```
g(x)=x^2
```

Out[11]:
```
g (generic function with 1 method)
```

In [12]:
```
g(8)
```

Out[12]:
```
64
```

In [13]:
```
a=1;b=2;c=3;d=4
Am = [a b;c d]
```

Out[13]:
```
2×2 Matrix{Int64}:
 1  2
 3  4
```

In [14]:
```
Am[1,1],Am[1,2],Am[2,1],Am[2,2]
```

Out[14]:
```
(1, 2, 3, 4)
```

In [15]:
```
?read()
```

Out[15]:

`read(io::IO, T)`
Read a single value of type `T` from `io`, in canonical binary representation.

Note that Julia does not convert the endianness for you. Use `ntoh` or `ltoh` for this purpose.

`read(io::IO, String)`
Read the entirety of `io`, as a `String` (see also `readchomp`).

Examples

```
jldoctest
julia> io = IOBuffer("JuliaLang is a GitHub organization");

julia> read(io, Char)
'J': ASCII/Unicode U+004A (category Lu: Letter, uppercase)

julia> io = IOBuffer("JuliaLang is a GitHub organization");

julia> read(io, String)
"JuliaLang is a GitHub organization"
```

`read(filename::AbstractString)`
Read the entire contents of a file as a `Vector{UInt8}`.

`read(filename::AbstractString, String)`
Read the entire contents of a file as a string.

`read(filename::AbstractString, args...)`
Open a file and read its contents. `args` is passed to `read`: this is equivalent to `open(io->read(io, args...), filename)`.

`read(s::IO, nb=typemax(Int))`
Read at most `nb` bytes from `s`, returning a `Vector{UInt8}` of the bytes read.

`read(s::IOStream, nb::Integer; all=true)`
Read at most `nb` bytes from `s`, returning a `Vector{UInt8}` of the bytes read.

If `all` is `true` (the default), this function will block repeatedly trying to read all requested bytes, until an error or end-of-file occurs. If `all` is `false`, at most one `read` call is performed, and the amount of data returned is device-dependent. Note that not all stream types support the `all` option.

`read(command::Cmd)`

Run `command` and return the resulting output as an array of bytes.

`read(command::Cmd, String)`

Run `command` and return the resulting output as a `String`.

In [16]:
```
Hello = IOBuffer("This is an IO buffer.")
read(Hello, String)
```

Out[16]:
```
"This is an IO buffer."
```

In [17]:
```
?readline()
```

Out[17]:
```
readline(io::IO=stdin; keep::Bool=false)
readline(filename::AbstractString; keep::Bool=false)
```

Read a single line of text from the given I/O stream or file (defaults to `stdin`). When reading from a file, the text is assumed to be encoded in UTF-8. Lines in the input end with `'\n'` or `"\r\n"` or the end of an input stream. When `keep` is false (as it is by default), these trailing newline characters are removed from the line before it is returned. When `keep` is true, they are returned as part of the line.

Return a `String`. See also `copyline` to instead write in-place to another stream (which can be a preallocated `IOBuffer`).

See also `readuntil` for reading until more general delimiters.

Examples

```
jldoctest
julia> write("my_file.txt", "JuliaLang is a GitHub organization.\nIt
has many members.\n");

julia> readline("my_file.txt")
"JuliaLang is a GitHub organization."

julia> readline("my_file.txt", keep=true)
"JuliaLang is a GitHub organization.\n"

julia> rm("my_file.txt")
julia> print("Enter your name: ")
Enter your name:

julia> your_name = readline()
Logan
"Logan"
```

In [18]:
```
write("file.txt","This is the first line from file.txt\n")
```

Out[18]:
```
37
```

In [19]:
```
readline("file.txt")
```

Out[19]:
```
"This is the first line from file.txt"
```

In [20]:
```
readline("file.txt", keep=true)
```

Out[20]:
```
"This is the first line from file.txt\n"
```

In [21]:
```
?readlines
```

```
search: readlines readline readlink readbytes! readdir eachline leading_ones
```

Out[21]:

```
readlines(io::IO=stdin; keep::Bool=false)
readlines(filename::AbstractString; keep::Bool=false)
```

Read all lines of an I/O stream or a file as a vector of strings. Behavior is equivalent to saving the result of reading `readline` repeatedly with the same arguments and saving the resulting lines as a vector of strings. See also `eachline` to iterate over the lines without reading them all at once.

Examples

```
jldoctest
julia> write("my_file.txt", "JuliaLang is a GitHub organization.\nIt
has many members.\n");

julia> readlines("my_file.txt")
2-element Vector{String}:
 "JuliaLang is a GitHub organization."
 "It has many members."

julia> readlines("my_file.txt", keep=true)
2-element Vector{String}:
 "JuliaLang is a GitHub organization.\n"
 "It has many members.\n"

julia> rm("my_file.txt")
```

In [22]:
```
?readdlm()
```

Out[22]: No documentation found.

Binding `readdlm` does not exist.

In [23]:
```
print("print")
println("println")
print("print")
```

```
printprintln
print
```

In [24]:
```
?show
```

```
search: show @show chown throw Cshort ispow2 chop showable
```

Out[24]:

`show([io::IO = stdout], x)`
Write a text representation of a value `x` to the output stream `io`. New types `T` should overload `show(io::IO, x::T)`. The representation used by `show` generally includes Julia-specific formatting and type information, and should be parseable Julia code when possible.

`repr` returns the output of `show` as a string.

For a more verbose human-readable text output for objects of type `T`, define `show(io::IO, ::MIME"text/plain", ::T)` in addition. Checking the `:compact IOContext` key (often checked as `get(io, :compact, false)::Bool`) of `io` in such methods is recommended, since some containers show their elements by calling this method with `:compact => true`.

See also `print`, which writes un-decorated representations.

Examples

```
jldoctest
julia> show("Hello World!")
"Hello World!"
julia> print("Hello World!")
Hello World!
```

`show(io::IO, mime, x)`
The `display` functions ultimately call `show` in order to write an object `x` as a given `mime` type to a given I/O stream `io` (usually a memory buffer), if possible. In order to provide a rich multimedia representation of a user-defined type `T`, it is only necessary to define a new `show` method for `T`, via: `show(io, ::MIME"mime", x::T) = ...`, where `mime` is a MIME-type string and the function body calls `write` (or similar) to write that representation of `x` to `io`. (Note that the `MIME""` notation only supports literal strings; to construct `MIME` types in a more flexible manner use `MIME{Symbol("")}`.)

For example, if you define a `MyImage` type and know how to write it to a PNG file, you could define a function `show(io, ::MIME"image/png", x::MyImage) = ...` to allow your images to be displayed on any PNG-capable `AbstractDisplay` (such as IJulia). As usual, be sure to `import Base.show` in order to add new methods to the built-in Julia function `show`.

Technically, the `MIME"mime"` macro defines a singleton type for the given `mime` string, which allows us to exploit Julia's dispatch mechanisms in determining how to display objects of any given type.

The default MIME type is `MIME"text/plain"`. There is a fallback definition for `text/plain` output that calls `show` with 2 arguments, so it is not always necessary to add a method for that case. If a type benefits from custom human-readable output though, `show(::IO, ::MIME"text/plain", ::T)` should be defined. For example, the `Day` type uses `1 day` as the output for the `text/plain` MIME type, and `Day(1)` as the output of 2-argument `show`.

Examples

```
jldoctest
julia> struct Day
           n::Int
       end

julia> Base.show(io::IO, ::MIME"text/plain", d::Day) = print(io, d.
n, " day")

julia> Day(1)
1 day
```

Container types generally implement 3-argument `show` by calling `show(io, MIME"text/plain"(), x)` for elements `x`, with `:compact => true` set in an `IOContext` passed as the first argument.

In [25]:
```
?MIME
```

```
search: MIME MIME""
```

Out[25]:

MIME

A type representing a standard internet data format. "MIME" stands for "Multipurpose Internet Mail Extensions", since the standard was originally used to describe multimedia attachments to email messages.

A `MIME` object can be passed as the second argument to `show` to request output in that format.

Examples

```
jldoctest
julia> show(stdout, MIME("text/plain"), "hi")
"hi"
```

In [26]:
```
?parse
```

```
search: parse tryparse pairs Base parent falses
```

Out[26]:

```
parse(type, str; base)
```

Parse a string as a number. For `Integer` types, a base can be specified (the default is 10). For floating-point types, the string is parsed as a decimal floating-point number. `Complex` types are parsed from decimal strings of the form `"R±Iim"` as a `Complex(R,I)` of the requested type; `"i"` or `"j"` can also be used instead of `"im"`, and `"R"` or `"Iim"` are also permitted. If the string does not contain a valid number, an error is raised.

!!! compat "Julia 1.1" `parse(Bool, str)` requires at least Julia 1.1.

Examples

```
jldoctest
julia> parse(Int, "1234")
1234

julia> parse(Int, "1234", base = 5)
194

julia> parse(Int, "afc", base = 16)
2812

julia> parse(Float64, "1.2e-3")
0.0012

julia> parse(Complex{Float64}, "3.2e-1 + 4.5im")
0.32 + 4.5im
```

```
parse(::Type{Platform}, triplet::AbstractString)
```

Parses a string platform triplet back into a `Platform` object.

```
parse(::Type{SimpleColor}, rgb::String)
```

An analogue of `tryparse(SimpleColor, rgb::String)` (which see), that raises an error instead of returning `nothing`.

In [ ]: