Presentación

```
using PlutoUI
```

Programación en Julia: Primeros pasos

Variables, tipos y operaciones

Héctor Medel

Benjamín Pérez

Existen distintos tipos de datos en Julia

Por ejemplo: Char, Int64, Float64

```
y = 7
```

```
Int64
```

```
typeof(y)
```

```
x = "CADI"
```

```
x = "CADI"
```

```
String
```

```
typeof(x)
```

```
w = 2.5;   # Agregamos ; para que no imprima el resultado
```

```
Float64
```

```
typeof(w)
```

```
UndefVarError: z not defined
```

1. top-level scope @ Local: 1

```
y+z
```

```
MethodError: no method matching +(::Int64, ::String)
Closest candidates are:
+(::Any, ::Any, !Matched::Any, !Matched::Any...) at operators.jl:591
+(::T, !Matched::T) where T<:Union{Int128, Int16, Int32, Int64, Int8, UInt128, UInt16, UInt32, UInt64, UInt8} at int.jl:87
+(::Union{Int16, Int32, Int64, Int8}, !Matched::BigInt) at gmp.jl:537
...
```

1. top-level scope @ Local: 1 [inlined]

```
y+x
```

```
9.5
```

```
y+w
```

Algunas ideas para nombrar variables

- Minúsculas con múltiples palabras separadas por un guion bajo (_).
- Nombres cortos.
- Podemos usar símbolos Unicode. Por ejemplo, **\alpha[TAB]** despliega α.

```
current_time = 0.7
```

```
current_time = 0.7
```

```
α = 2
```

Los comentarios se agregan usando el símbolo **#**

```
β = 3.1 # Este es un comentario
```

Para desplegar valores usamos la funcion **print()** o **println()**

```
print(β)
```

```
3.1
```

Incluso podemos agregar texto y el valor de una variable usando el mismo comando

```
print("El valor de la variable β es $(β)")
```

```
El valor de la variable β es 3.1
```

El sistema de tipos en Julia es único

- Julia se comporta de manera dinámica. Es decir, una variable relacionada a una cantidad Int64, puede cambiar y estar relacionada a un String.

```
begin   # Noten que en esta celda usamos multiples lineas de codigo!
    r = 10
    println(typeof(r))
    r = "Hola"
    println(typeof(r))
end
```

```
Int64
String
```

- Algo que podemos hacer es agregar el tipo de dato a una variable. Esto lo hacemos con la sintaxis **variable::Tipo**

```
MethodError: Cannot `convert` an object of type String to an object of type Int64
Closest candidates are:
convert(::Type{T}, !Matched::Ptr) where T<:Integer at pointer.jl:23
convert(::Type{T}, !Matched::T) where T<:Number at number.jl:6
convert(::Type{T}, !Matched::Number) where T<:Number at number.jl:7
...
```

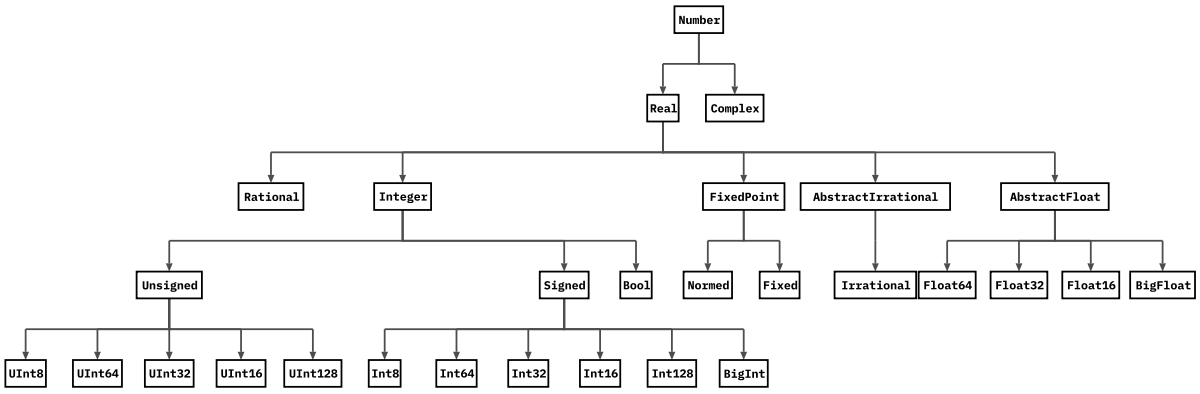
1. **top-level scope** @ *Local: 4* [inlined]

```
begin   # Noten que en esta celda usamos multiples lineas de codigo!
    a::Int64 = 10
    println(typeof(a))
    a = "Hola"
end
```

```
Int64
```

Código estable en tipo

- En general, el tipo de una variable puede cambiar en Julia, pero esto puede afectar el rendimiento en la ejecución.
- De manera automática, Julia infiere el tipo de las variables.
- Para tener buen performance, necesitamos escribir código estable en tipo. Es decir, cada variable que usamos en alguna función no cambie de tipo a lo largo de su ejecución.
- La jerarquía de tipos para datos numéricos es la siguiente.



Conversión de variables

- Para convertir el tipo de variable usamos la siguiente instrucción **Type(Var)**

```
3.0
```

```
Float64(3)
```

```
MethodError: no method matching Int64(::String)
Closest candidates are:
(::Type{T})(!Matched::AbstractChar) where T<:Union{Int32, Int64} at char.jl:51
(::Type{T})(!Matched::AbstractChar) where T<:Union{AbstractChar, Number} at char.jl:50
(::Type{T})(!Matched::BigInt) where T<:Union{Int128, Int16, Int32, Int64, Int8} at gmp.jl:359
...
```

1. **top-level scope** @ *Local: 1* [inlined]

```
Int64("Hola")
```

Las funciones matemáticas típicas se encuentran disponibles

Probemos con sqrt(), exp(), log(), sin(), cos(), rand()...

```
a1 = π = 3.1415926535897...
```

```
a1 = pi
```

```
b1 = -1.0
```

```
b1 = cos(a1)
```

```
b2 = 2.718281828459045
```

```
b2 = exp(1.0)
```

- En la documentación encontrarás más funciones.
- Algunas funciones especiales están implementadas en Pkgs que veremos más adelante (por ejemplo **GSL.jl**).

El manejo de números complejos está bien soportado

- Las operaciones elementales como abs(), real(), imag(), exp(), están definidas para cantidades complejas.

```
z1 = 4.0 + 3.0im
```

```
z1 = 4.0 + im*3.0
```

```
ComplexF64 (alias for Complex{Float64})
```
```
typeof(z1)
```

```
(5.0, 25.0)
```
```
abs(z1), abs2(z1)
```

```
(4.0, 3.0)
```
```
real(z1), imag(z1)
```

Tipo de dato Char

Un caracter (individual) corresponde a un **Char**

```
caracterA = 'α': Unicode U+03B1 (category Ll: Letter, lowercase)
```
```
caracterA = 'α'
```

```
Char
```
```
typeof(caracterA)
```

```
945
```
```
Int16(caracterA)
```

```
"0000000000000000000000000000000000000000000000000000001110110001"
```
```
bitstring(Int(caracterA))
```

- Podemos hacer algo similar en la "dirección" opuesta.

```
'α': Unicode U+03B1 (category Ll: Letter, lowercase)
```
```
Char(945)
```

Tipo de dato String

Un **String** consiste en un arreglo de caracteres.

```
palabra = "Tecnologico"
```
```
palabra = "Tecnologico"
```

```
'T': ASCII/Unicode U+0054 (category Lu: Letter, uppercase)
```
```
palabra[1]
```

Manejo de arreglos y rangos

```
rangoA = 0:10
```
```
rangoA = 0:10
```

```
UnitRange{Int64}
```
```
typeof(rangoA)
```

```
for ii = 1:5  # ii in 1:5   --> podemos iterar a lo largo de un rango
    println(ii)
end
```

```
1
2
3
4
5
```

```
crA = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```
```
crA = collect(rangoA)
```

```
Vector{Int64} (alias for Array{Int64, 1})
```
```
typeof(crA)
```

- En general, el formato de un arreglo es **Array{Tipo, n}**, donde n es el número de dimensiones.

```
arr = [1.2, 2.2, 3.5]
```
```
arr = [1.2, 2.2, 3.5]
```

En ocasiones es recomendable inicializar arreglos

- Tenemos varias opciones

```
arr1 = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```
```
arr1 = zeros(10)
```

```
Vector{Float64} (alias for Array{Float64, 1})
```
```
typeof(arr1)
```

```
arr2 = [6.90093e-310, 0.0, 6.90093e-310, 0.0, 6.901e-310, 0.0, 6.90101e-310, 0.0, 6.90093e-310, 0.0]
```
```
arr2 = Array{Float64}(undef, 10)
```

```
Vector{Float64} (alias for Array{Float64, 1})
```
```
typeof(arr2)
```

```
arr3 = []
```
```
arr3 = Float64[]
```

```
Vector{Float64} (alias for Array{Float64, 1})
```
```
typeof(arr3)
```

```
[1.0]
```
```
push!(arr3, 1.0)
```

```
arr4 = []
```
```
arr4 = []
```

```
Vector{Any} (alias for Array{Any, 1})
```
```
typeof(arr4)
```

Las principales características de un arreglo son dadas por las siguientes funciones

- Tipo de los elementos: **eltype(arr)**
- Número de elementos: **length(arr)**
- Número de dimensiones: **ndims(arr)**
- Número de elementos por dimensión: **size(arr)**

```
arr5 = 3×4 Matrix{Int64}:
 1   2   3   4
 5   6   7   8
 9  10  11  12
```
```
arr5 = [1  2  3  4;
        5  6  7  8;
        9 10 11 12]
```

```
Matrix{Int64} (alias for Array{Int64, 2})
```
```
typeof(arr5)
```

```
Int64
```
```
eltype(arr5)
```

```
12
```
```
length(arr5)
```

```
2
```

```
ndims(arr5)
```

```
(3, 4)
```

```
size(arr5)
```

Indexado de arreglos

- Usamos **[]** para acceder a elementos dentro de un arreglo.
- El operador **:** nos ayuda a generar rangos, los cuales podemos usar para accesar a un conjunto de elementos en un arreglo.

```
c1 = 10×5 Matrix{Float64}:
      0.194006  0.898124   0.124543   0.488982   0.185488
      0.689783  0.500552   0.337955   0.0480732  0.254971
      0.325633  0.7882     0.0240456  0.684892   0.571667
      0.663696  0.949033   0.23322    0.478213   0.918403
      0.364027  0.262315   0.372628   0.0927152  0.847402
      0.558809  0.892896   0.0521169  0.499027   0.585923
      0.743075  0.292765   0.808707   0.780286   0.621375
      0.334759  0.410942   0.793941   0.858249   0.46909
      0.12525   0.0396948  0.203288   0.620274   0.888549
      0.320916  0.475668   0.830862   0.852435   0.0843697
```

```
c1 = rand(10,5) # Esto genera una matriz aleatoria de 10x3 elementos
```

```
0.23321977974671715
```

```
c1[4,3] # [renglon, columna]
```

```
[0.325633, 0.7882, 0.0240456, 0.684892, 0.571667]
```

```
c1[3,:] # Que tipo de arreglo es?
```

```
[0.898124, 0.500552, 0.7882, 0.949033, 0.262315, 0.892896, 0.292765, 0.410942, 0.0396948, 0.475668]
```

```
c1[:,2] # Que tipo de arreglo es?
```

```
4×4 Matrix{Float64}:
 0.7882    0.0240456  0.684892   0.571667
 0.949033  0.23322    0.478213   0.918403
 0.262315  0.372628   0.0927152  0.847402
 0.892896  0.0521169  0.499027   0.585923
```

```
c1[3:6,2:5]
```

Otras funciones comunes para arreglos

```
aa = [1, 2, 3, 4, 5, 6, 7]
```

```
aa = collect(1:7)
```

```
bb = [100, 200, 300]
```

```
bb = [100, 200, 300]
```

```
[1, 2, 3, 4, 5, 6, 7]
```

```
aa[:]
```

```
[100, 200, 300]
```

```
bb[:]
```

```
[1, 2, 3, 4, 5, 6, 7, 100, 200, 300]
```

```
[aa; bb]  # Esto concatena ambos vectores
```

- Existe otra opción usando **append!()**

```
cc = [1, 2, 3, 4, 5, 6, 7]
```

```
cc = collect(1:7)
```

```
dd = [100, 200, 300]
```

```
dd = [100, 200, 300]
```

```
▸ [1, 2, 3, 4, 5, 6, 7, 100, 200, 300]
```

```
append!(cc,dd)   # Notemos el uso del símbolo !
```

Operaciones por elemento – Operador .

```
x1 = ▸ [0.0, 0.785398, 1.5708, 2.35619, 3.14159, 3.92699, 4.71239, 5.49779, 6.28319]
```

```
x1 = collect(0.0:pi/4:2pi)
```

```
MethodError: no method matching +(::Vector{Float64}, ::Float64)
For element-wise addition, use broadcasting with dot syntax: array .+ scalar
Closest candidates are:
+(::Any, ::Any, !Matched::Any, !Matched::Any...) at operators.jl:591
+(!Matched::T, ::T) where T<:Union{Float16, Float32, Float64} at float.jl:383
+(!Matched::Base.TwicePrecision, ::Number) at twiceprecision.jl:290
...
```

1. **top-level scope** @ *Local: 1* [inlined]

```
x1 + 5.0  # Esta operación no está bien definida
```

```
▸ [5.0, 5.7854, 6.5708, 7.35619, 8.14159, 8.92699, 9.71239, 10.4978, 11.2832]
```

```
x1 .+ 5.0
```

```
MethodError: no method matching sin(::Vector{Float64})
Closest candidates are:
sin(!Matched::T) where T<:Union{Float32, Float64} at special/trig.jl:29
sin(!Matched::LinearAlgebra.Hermitian{var"#s884", S} where {var"#s884"<:Complex, S<:(AbstractMatrix{<:var"#s884"})}) at /usr/sh
sin(!Matched::Union{LinearAlgebra.Hermitian{var"#s885", S}, LinearAlgebra.Symmetric{var"#s885", S}} where {var"#s885"<:Real, S}
...
```

1. **top-level scope** @ *Local: 1* [inlined]

```
sin(x1)  # Mismo caso para otra funciones matemáticas (más adelante retomaremos esto...)
```

```
▸ [0.0, 0.707107, 1.0, 0.707107, 1.22465e-16, -0.707107, -1.0, -0.707107, -2.44929e-16]
```

```
sin.(x1)
```