

# ÖGOR Summer-Workshop for PhD-candidates and Post-Docs

An introduction to Julia and JuMP for Operations Research

Prof. Dr. Xavier Gandibleux

Nantes Université – France  
Département Informatique – Faculté des Sciences et Techniques

Topic 2

Getting started

# Some basics



# How to assign a value to a variable (1/2)...

Integer (Int8, Int16, Int32, Int64):

```
julia> zipcode = 44000
```

44000 is an integer literal

Float (Float16, Float32, Float64):

```
julia> price = 29.95
```

29.95 is a floating point literal

Boolean:

```
julia> positive = true
```

# How to assign a value to a variable (1/2)...

Integer (Int8, Int16, Int32, Int64):

```
julia> zipcode = 44000
```

44000 is an integer literal

Float (Float16, Float32, Float64):

```
julia> price = 29.95
```

29.95 is a floating point literal

Boolean:

```
julia> positive = true
```

## How to assign a value to a variable (1/2)...

Integer (Int8, Int16, Int32, Int64):

```
julia> zipcode = 44000
```

44000 is an integer literal

Float (Float16, Float32, Float64):

```
julia> price = 29.95
```

29.95 is a floating point literal

Boolean:

```
julia> positive = true
```

# How to assign a value to a variable (1/2)...

Integer (Int8, Int16, Int32, Int64):

```
julia> zipcode = 44000
```

44000 is an integer literal

Float (Float16, Float32, Float64):

```
julia> price = 29.95
```

29.95 is a floating point literal

Boolean:

```
julia> positive = true
```

## How to assign a value to a variable (2/2)...

Character:

```
julia> dot = '.'
```

String:

```
julia> sentence = "."
```

## How to assign a value to a variable (2/2)...

Character:

```
julia> dot = '.'
```

String:

```
julia> sentence = "."
```



# Name of variables

Julia manages the unicode characters.

Exemples of valid names for a variable :

- ▶ greek letters:

```
\Delta + [TAB]
```

- ▶ an emoji:

```
\:smile: + [TAB]
```

- ▶ etc.

(see <https://docs.julialang.org/en/v1/manual/unicode-input/>)

# How to get the type of an expression...

Definition:

```
typeof(...)
```

Example:

```
julia> typeof(zipcode)
```

# How to get the type of an expression...

Definition:

```
typeof(...)
```

Example:

```
julia> typeof(zipcode)
```

# How to get the type of an expression...

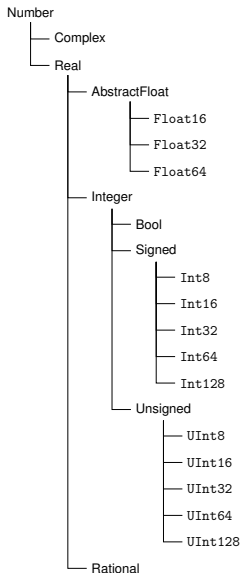
Definition:

```
typeof(...)
```

Example:

```
julia> typeof(zipcode)
```

# Partial hierarchy of types for numbers

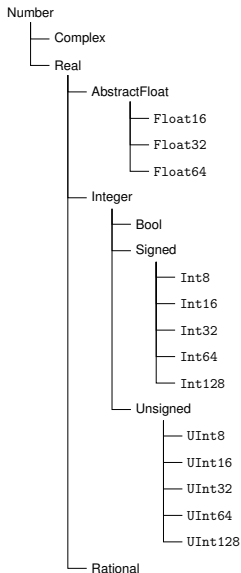


For a type  $T$ ,

*Function*  
supertype( $T$ )  
subtypes( $T$ )

*Definition*  
parent type  
child type(s)

# Partial hierarchy of types for numbers



For a type  $T$ ,

*Function*  
supertype( $T$ )  
subtypes( $T$ )

*Definition*  
parent type  
child type(s)

# Integer and floating point numbers

For a type  $T$ ,

<i>Function</i>	<i>Definition</i>
<code>sizeof(T)</code>	number of bytes used to represent a value of this type
<code>typemax(T)</code>	largest value for this type
<code>typemin(T)</code>	smallest value for this type

Example:

$T$	<i>Signed?</i>	<i>sizeof(T)</i>	<i>typemin(T)</i>	<i>typemax(T)</i>
Int16	yes	2	$-2^{15} = -32768$	$2^{15} - 1 = 32767$

# Constants

Definition:

```
const variable = value
```

Example:

```
julia> const g = 9.81
```

Some predefined symbolic constants:

pi	or	$\pi$	value of pi
Inf			positive infinity of type Float64
NaN			not a number



# How to print (1/3)...

Display without carriage return:

```
print(...)
```

Display with carriage return:

```
println(...)
```

Examples:

```
julia> print(price)
julia> println()
julia> println(zipcode)
```

```
julia> println("Zip Code:  ", zipcode)
julia> println(zipcode, " | ", price, "€ ", dot)
julia> println("Zip Code: $zipcode")
```

## How to print (1/3)...

Display without carriage return:

```
print(...)
```

Display with carriage return:

```
println(...)
```

Examples:

```
julia> print(price)
julia> println()
julia> println(zipcode)
```

```
julia> println("Zip Code:  ", zipcode)
julia> println(zipcode, " | ", price, "€ ", dot)
julia> println("Zip Code: $zipcode")
```

## How to print (1/3)...

Display without carriage return:

```
print(...)
```

Display with carriage return:

```
println(...)
```

Examples:

```
julia> print(price)
julia> println()
julia> println(zipcode)
```

```
julia> println("Zip Code: ", zipcode)
julia> println(zipcode, " | ", price, "€ ", dot)
julia> println("Zip Code: $zipcode")
```

## How to print (1/3)...

Display without carriage return:

```
print(...)
```

Display with carriage return:

```
println(...)
```

Examples:

```
julia> print(price)
julia> println()
julia> println(zipcode)
```

```
julia> println("Zip Code:  ", zipcode)
julia> println(zipcode, " | ", price, "€ ", dot)
julia> println("Zip Code: $zipcode")
```

## How to print (2/3)...

Display in c-like style:

```
@printf(...)
```

Example:

```
julia> using Printf
```

```
julia> @printf("Zip Code:  %d \n",zipcode)
```

```
julia> @printf("π = %.2f \n", pi)
```

## How to print (2/3)...

Display in c-like style:

```
@printf(...)
```

Example:

```
julia> using Printf
```

```
julia> @printf("Zip Code:  %d \n",zipcode)
```

```
julia> @printf("π = %.2f \n", pi)
```

## How to print (2/3)...

Display in c-like style:

```
@printf(...)
```

Example:

```
julia> using Printf
```

```
julia> @printf("Zip Code:  %d \n",zipcode)
```

```
julia> @printf("π = %.2f \n", pi)
```

## How to print (3/3)...

Display an expression and result:

```
@show ...
```

Example:

```
julia> @show zipcode
```



## How to print (3/3)...

Display an expression and result:

```
@show ...
```

Example:

```
julia> @show zipcode
```

## Get input from users (1/3)...

Read a line of text from the console (STDIN) until a [NEWLINE]:

```
readline()
```

Example:

```
julia> name = readline()
```

The type of `name`, obtained with `typeof(name)`, is `string`.

## Get input from users (1/3)...

Read a line of text from from the console (STDIN) until a [NEWLINE]:

```
readline()
```

Example:

```
julia> name = readline()
```

The type of `name`, obtained with `typeof(name)`, is `string`.

## Get input from users (2/3)...

Read numerical data types from console:

```
parse()
```

...convert a numeric string (Float or Int) into a numerical value.

Example:

```
julia> num = parse{Int64, readline() }
```

The type of `num`, obtained with `typeof(num)`, is `Int64`.

## Get input from users (2/3)...

Read numerical data types from console:

```
parse()
```

...convert a numeric string (Float or Int) into a numerical value.

Example:

```
julia> num = parse{Int64, readline()})
```

The type of `num`, obtained with `typeof(num)`, is `Int64`.

## Get input from users (3/3)...

Read N lines of text input from the console:

```
readlines()
```

N lines are stored as the entries of a one-dimensional String array.  
Lines must be delimited with [NEWLINE] or by pressing [ENTER].  
[Ctrl-D] to stop taking input.

Example:

```
julia> lines = readlines()
```

See also `TerminalMenus` for simple interactive menus in the terminal  
(an example is given in the project).

## Get input from users (3/3)...

Read N lines of text input from the console:

```
readlines()
```

N lines are stored as the entries of a one-dimensional String array.  
Lines must be delimited with [NEWLINE] or by pressing [ENTER].  
[Ctrl-D] to stop taking input.

Example:

```
julia> lines = readlines()
```

See also `TerminalMenus` for simple interactive menus in the terminal  
(an example is given in the project).

## Get input from users (3/3)...

Read N lines of text input from the console:

```
readlines()
```

N lines are stored as the entries of a one-dimensional String array.  
Lines must be delimited with [NEWLINE] or by pressing [ENTER].  
[Ctrl-D] to stop taking input.

Example:

```
julia> lines = readlines()
```

See also `TerminalMenus` for simple interactive menus in the terminal (an example is given in the project).



# How to write a comment...

Definition:

```
# ...
```

```
#= ... =#
```

Example:

```
julia> # this is a line comment
```

```
julia> #= And this  
        is a block of comment  
        =#
```

# How to write a comment...

Definition:

```
# ...
```

```
#= ... =#
```

Example:

```
julia> # this is a line comment
```

```
julia> #= And this  
        is a block of comment  
        =#
```

# How to write a comment...

Definition:

```
# ...
```

```
#= ... =#
```

Example:

```
julia> # this is a line comment
```

```
julia> #= And this  
        is a block of comment  
        =#
```

# How to do basic math...

## Arithmetic operators

With  $a$  and  $b$ , two variables:

<i>Operator</i>	<i>Expression</i>	<i>Name</i>
-	$-a$	unary minus
+	$a + b$	sum
-	$a - b$	difference
*	$a * b$	product
/	$a / b$	quotient
^	$a ^ b$	power
÷	$a \div b$	integer divide
%	$a \% b$	modulus

# How to do basic math...

## Arithmetic operators

With  $a$  and  $b$ , two variables:

<i>Operator</i>	<i>Expression</i>	<i>Name</i>
-	$-a$	unary minus
+	$a + b$	sum
-	$a - b$	difference
*	$a * b$	product
/	$a / b$	quotient
^	$a ^ b$	power
÷	$a \div b$	integer divide
%	$a \% b$	modulus

# How to do basic math...

## Updating operators

With  $a$  and  $b$ , two variables:

<i>Operator</i>	<i>example</i>	<i>equivalent to</i>
$+=$	$a+=b$	$a=a+b$
$-=$	$a-=b$	$a=a-b$
$*=$	$a*=b$	$a=a*b$
$/=$	$a/=b$	$a=a/b$
$\hat{=}$	$a\hat{=b}$	$a=a^b$
$\div=$	$a\div=b$	$a=a\div b$
$\%=$	$a\%=b$	$a=a\%b$

# How to do basic math...

## Updating operators

With  $a$  and  $b$ , two variables:

<i>Operator</i>	<i>example</i>	<i>equivalent to</i>
$+=$	$a+=b$	$a=a+b$
$-=$	$a-=b$	$a=a-b$
$*=$	$a*=b$	$a=a*b$
$/=$	$a/=b$	$a=a/b$
$\hat{=}$	$a\hat{=b}$	$a=a^b$
$\div=$	$a\div=b$	$a=a\div b$
$\%=$	$a\%=b$	$a=a\%b$

# How to do basic math...

## Numeric comparisons operators

With a and b, two variables:

<i>Operator</i>	<i>Expression</i>	<i>Name</i>
<code>==</code>	<code>a == b</code>	equality
<code>!=</code> or <code>≠</code>	<code>a != b</code>	inequality
<code>&lt;</code>	<code>a &lt; b</code>	less than
<code>&lt;=</code> or <code>≤</code>	<code>a &lt;= b</code>	less than or equal to
<code>&gt;</code>	<code>a &gt; b</code>	greater than
<code>&gt;=</code> or <code>≥</code>	<code>a &gt;= b</code>	greater than or equal to



# A selection of usual functions available (1/4)

## Arithmetic functions

With  $x$  and  $y$ , two variables;  $n$  and  $p$ , two integer variables:

<i>Function</i>	<i>Description</i>
<code>div(x,y)</code>	truncated division; quotient rounded towards zero
<code>mod(x,y)</code>	modulus
<code>abs(x)</code>	a positive value with the magnitude of $x$
<code>sign(x)</code>	indicates the sign of $x$ , returning -1, 0, or +1
<code>sqrt(x)</code>	square root of $x$
<code>exp(x)</code>	$e^x$
<code>exp2(x)</code>	$2^x$
<code>log(x)</code>	natural logarithm of $x$
<code>log(b,x)</code>	base $b$ logarithm of $x$
<code>factorial(n)</code>	$n!$
<code>binomial(n,p)</code>	the binomial coefficient

## A selection of usual functions available (2/4)

### Rounding functions

With  $x$ , a variable and  $T$ , a numeric type:

<i>Function</i>	<i>Description</i>	<i>Return type</i>
<code>round(x)</code>	round $x$ to the nearest integer	<code>typeof(x)</code>
<code>round(T,x)</code>	round $x$ to the nearest integer	$T$
<code>round(x,digits=n)</code>	round to $n$ digits after the decimal place	<code>typeof(x)</code>
<code>floor(x)</code>	round $x$ towards $-\text{Inf}$	<code>typeof(x)</code>
<code>floor(T,x)</code>	round $x$ towards $-\text{Inf}$	$T$
<code>ceil(x)</code>	round $x$ towards $+\text{Inf}$	<code>typeof(x)</code>
<code>ceil(T,x)</code>	round $x$ towards $+\text{Inf}$	$T$
<code>trunc(x)</code>	round $x$ towards zero	<code>typeof(x)</code>
<code>trunc(T,x)</code>	round $x$ towards zero	$T$

## A selection of usual functions available (3/4)

### Trigonometric functions

With  $x$ , a variable:

<i>Function</i>	<i>Description</i>
<code>sin(x)</code>	Compute sine of $x$ , where $x$ is in radians
<code>cos(x)</code>	Compute cosine of $x$ , where $x$ is in radians
<code>tan(x)</code>	Compute tangent of $x$ , where $x$ is in radians
<code>sind(x)</code>	Compute sine of $x$ , where $x$ is in degrees
<code>cosd(x)</code>	Compute cosine of $x$ , where $x$ is in degrees
<code>tand(x)</code>	Compute tangent of $x$ , where $x$ is in degrees
<code>deg2rad(x)</code>	Convert $x$ from degrees to radians
<code>rad2deg(x)</code>	Convert $x$ from radians to degrees

## A selection of usual functions available (4/4)

### functions on strings

With *c*, a character, *s*, *s2*, strings, :

<i>Function</i>	<i>Description</i>
<code>reverse(s)</code>	Reverse a string
<code>occursin(c,s)</code>	Determine if <i>c</i> is a substring of <i>s</i>
<code>split(s,c)</code>	Split <i>s</i> into an array of substrings on occurrences of <i>c</i>
<code>endswith(s,s2)</code>	Return true if <i>s</i> ends with <i>s2</i>
<code>uppercase(s)</code>	Return <i>s</i> with all characters converted to uppercase
<code>textwidth(s)</code>	Give the number of columns needed to print <i>s</i>
<code>isdigit(c)</code>	Test if <i>c</i> is a decimal digit (0-9)
<code>isletter(c)</code>	Test if <i>c</i> is a letter

At this stage, Julia is a super calculator



