در چند دقیقه زبان برنامه نویسی جولیا julia را فراگیرید!

جولیا یک زبان جدید کامپیوتری با تمرکز بر محاسبات عددی میباشد . این زبان:

** مانند زبان لیسپ "هوموِیکُنیک" (homoiconic) است، به این معنا که کدهای برنامه نیز داده هایی از برنامه به شمار میروند، که امکان تولید کدهای برنامه نویسی پویا (meta-programming) را به برنامه نویس میدهد.

** در تعریف توابع بسیار قوی و انعطاف پذیر میباشد، که امکان تعریف رُویه های (methods) و عملگرهای محاسباتی جامع (generic) برای ساختارهای داده متفاوت را فراهم می آورد.

** دارای قابلیت های سطح پایین کنترلی و محاسباتی است، که سرعت اجرا را تا حد زبانهای سی یا فرترن افزایش میدهد و

** ساده برای یادگیری و استفاده میباشد. که آنرا به سرعت به جایگزینی برای زبانهای محاسباتی مانند متلب، متمتیکا و ... یا گزینه ای جایگزین با سرعت بالاتر برای پِیتون و زبانهای اسکریپتی مشابه تبدیل می نماید

این آموزش کوتاه بر اساس نسخه 18 اکتبر 2013 تهیه شده است.

```
# Ref:-> http://learnxinyminutes.com/docs/julia/
```

توضیحات یک خطی با کاراکتر # آغاز می شوند:

```
# Single line comments start with a hash.
```

توضیحات چند خطی با "#=" آغاز می گردند و با "=#" خاتمه می یابند:

```
#= Multiline comments can be written
   by putting '#=' before the text  and '=#'
   after the text. They can also be nested.
=#
```

```
####################################################
## 1 – داده های پایه
####################################################
```

```
####################################################
## 1. Primitive Datatypes and Operators
####################################################
```

کدهای جولیا داده هایی از نوع عبارت میباشند، بنابراین همه چیز در این زبان عباراتی قابل تفسیر توسط جولیا است.

```
# Everything in Julia is a expression.
```

انواع مختلفی از اعداد وجود دارد:

```
# There are several basic types of numbers.
```

اعداد صحیح:

```
3 # => 3 (Int64)
```

اعداد اعشاری:

```
3.2 # => 3.2 (Float64)
```

اعداد مختلط:

```
2 + 1im # => 2 + 1im (Complex{Int64})
```

اعداد کسری:

```
2//3 # => 2//3 (Rational{Int64})
```

کلیه عملگرهای محاسباتی در دسترسند:

```
# All of the normal infix operators are available.
1 + 1 # => 2
8 - 1 # => 7
10 * 2 # => 20
35 / 5 # => 7.0
```

حاصل تقسیم دو عدد صحیح همیشه از نوع اعشاری است:

```
5 / 2 # => 2.5 # dividing an Int by an Int always results in a Float
```

جهت محاسبه بخش صحیح یک تقسیم تابع `div` در دسترس است:

```
div(5, 2) # => 2 # for a truncated result, use div
5 \ 35 # => 7.0
```

^ عملگر توان است نه`xor` دودویی!:

```
2 ^ 2 # => 4 # power, not bitwise xor
12 % 10 # => 2
```

اولویت بندی محاسبات با استفاده از پرانتز:

```
# Enforce precedence with parentheses
(1 + 3) * 2 # => 8
```

عملگرهای دودویی:

```
# Bitwise Operators
```

نه دودویی:

```
~2 # => -3   # bitwise not
```

و دودویی:

```
3 & 5 # => 1 # bitwise and
```

یا دودویی:

```
2 | 4 # => 6 # bitwise or
```

xor دودویی:

```
2 $ 4 # => 6 # bitwise xor
```

حرکت دودویی به راست بدون علامت:

```
2 >>> 1 # => 1 # logical shift right
```

حرکت دودویی به راست:

```
2 >> 1  # => 1 # arithmetic shift right
```

حرکت دودویی به چپ:

```
2 << 1  # => 4 # logical/arithmetic shift left
```

میتوانید از تابع bits جهت چاپ نمای دودویی یک عدد استفاده نمایید:

```
# You can use the bits function to see the binary representation of a number.
bits(12345)
# => "00000000000000000000000000000000000000000000000011000000111001"
bits(12345.0)
# => "0100000011001000000111001000000000000000000000000000000000000000"
```

داده های منطقی از انواع اصلی میباشند:

```
# Boolean values are primitives
true
false
```

عملگرهای منطقی و رابطه ای:

```
# Boolean operators
!true # => false
!false # => true
1 == 1 # => true
2 == 1 # => false
1 != 1 # => false
2 != 1 # => true
1 < 10 # => true
1 > 10 # => false
2 <= 2 # => true
2 >= 2 # => true
```

مقایسه ها میتوانند زنجیره ای باشند:

```
# Comparisons can be chained
1 < 2 < 3 # => true
2 < 3 < 2 # => false
```

رشته ها با " تعریف میشوند:

```
"Strings are created with #
".This is a string"
```

کاراکترها ها با ' مشخص میشوند:

```
# Character literals are written with '
'a'
```

رشته ها مانند آرایه ای از کاراکترها قابل اندیس گذاری میباشند:

```
# A string can be indexed like an array of characters
"This is a string"[1] # => 'T' # Julia indexes from 1
```

البته این روش برای رشته های با کد گذاری UTF8 مناسب نمی باشد.

```
# However, this is will not work well for UTF8 strings,
# so iterating over strings is recommended (map, for loops, etc).
```

کاراکتر $ عبارات julia را در داخل یک رشته بیان میکند:

```
# $ can be used for string interpolation:
"2 + 2 = $(2 + 2)" # => "2 + 2 = 4"
# You can put any Julia expression inside the parenthesis.
```

روش جایگزین جهت تعریف قالب های چاپی برای رشته ها استفاده از ماکرو @printf میباشد:

```
# Another way to format strings is the printf macro.
@printf "%d is less than %f" 4.5 5.3 # 5 is less than 5.300000
```

تولید خروجی قابل چاپ ساده می باشد:

```
# Printing is easy
println("I'm Julia. Nice to meet you!")
```

```
####################################################
## 2 – متغییرها و انواع مجموعه ها
####################################################
```

```
####################################################
## 2. Variables and Collections
####################################################
```

پیش از اعلان متغییر میتوانید به آن مقدار بدهید:

```
# You don't declare variables before assigning to them.
some_var = 5 # => 5
some_var # => 5
```

حاصل ارجاع به متغییر قبل از مقدار دهی، تولید خطی می باشد:

```
# Accessing a previously unassigned variable is an error
try
    some_other_var # => ERROR: some_other_var not defined
catch e
    println(e)
end
```

اسامی متغییرها با یک حرف آغاز میشوند و بعد از آن میتوان ترکیبی از حروف، اعداد، زیرخط و علامت تعجب داشت:

```
# Variable names start with a letter.
# After that, you can use letters, digits, underscores, and exclamation
points.
```

```
SomeOtherVar123! = 6 # => 6
```

بعلاوه در نامگذاری متغییر ها، میتوان از کاراکترهای با کدینگ UTF8 نیز استفاده کرد:

```
# You can also use unicode characters
ρ = 8 # => 8
```

این امکان در انجام محاسبات ریاضی میتواند راهگشا باشد:

```
# These are especially handy for mathematical notation
2 * π # => 6.283185307179586
```

قراردادهای نامگذاری در زبان جولیا:

اسامی متغییرها با حروف کوچک نوشته شود و کلمات با زیر خط '_' تفکیک گردند.

اسامی انواع داده ها، و کلمات مندرج در اسم ها بدون زیرخط با حروف بزرگ آغاز میشوند.

اسامی توابع و ماکروها با حروف کوچک و بدون زیرخط انتخاب گردند.

در صورتی که تابع امکان تغییر پارامتر خود را داشته باشد، نام تابع به علامت تعجب '!' خاتمه یابد.

```
# A note on naming conventions in Julia:
#
# * Names of variables are in lower case, with word separation indicated by
#   underscores ('\_').
#
# * Names of Types begin with a capital letter and word separation is shown
#   with CamelCase instead of underscores.
#
# * Names of functions and macros are in lower case, without underscores.
#
# * Functions that modify their inputs have names that end in !. These
#   functions are sometimes called mutating functions or in-place functions.
```

آرایه ها مقادیر مرتب شده ای از داده ها را نگه میدارند، اندیس گذاری در جولیا از 1 آغاز میگردد:

```
# Arrays store a sequence of values indexed by integers 1 through n:
a = Int64[] # => 0-element Int64 Array
```

مقادیر آرایه های یک بعدی می توانند با داده های جدا شده با کما ',' مشخص گردند:

```
# 1-dimensional array literals can be written with comma-separated values.
b = [4, 5, 6] # => 3-element Int64 Array: [4, 5, 6]
b[1] # => 4
b[end] # => 6
```

مقادیر آرایه های دوبعدی در ردیف های جدا شده با نقطه کما ';' و مقادیر فاصله دار درج میگردند:

```
# 2-dimentional arrays use space-separated values and semicolon-separated
rows.
matrix = [1 2; 3 4] # => 2x2 Int64 Array: [1 2; 3 4]
```

توابع push! و append! به انتهای یک لیست داده می افزایند:

```
# Add stuff to the end of a list with push! and append!
push!(a,1)      # => [1]
push!(a,2)      # => [1,2]
push!(a,4)      # => [1,2,4]
push!(a,3)      # => [1,2,4,3]
append!(a,b) # => [1,2,4,3,4,5,6]
```

pop! از آخرین عضو لیست را حذف میکند:

```
# Remove from the end with pop
pop!(b)         # => 6 and b is now [4,5]
# Let's put it back
push!(b,6)    # b is now [4,5,6] again.
```

به یاد داشته باشید که اندیسها در جولیا از 1 شروع میشوند، نه 0:

```
a[1] # => 1 # remember that Julia indexes from 1, not 0!
```

یک راه سریع برای استخراج آخرین عضو، استفاده از اندیس end میباشد:

```
# end is a shorthand for the last index. It can be used in any
# indexing expression
a[end] # => 6
```

میتوانیم اعضاء لیست را به سمتی شیفت نماییم:

```
# we also have shift and unshift
shift!(a) # => 1 and a is now [2,4,3,4,5,6]
unshift!(a,7) # => [7,2,4,3,4,5,6]
```

توابعی که اسامی آنها به علامت تعجب ختم میگردد، مقدار پارامترهای خود را تغییر میدهند:

```
# Function names that end in exclamations points indicate that they modify
# their argument.
arr = [5,4,6] # => 3-element Int64 Array: [5,4,6]
sort(arr) # => [4,5,6]; arr is still [5,4,6]
sort!(arr) # => [4,5,6]; arr is now [4,5,6]
```

حاصل ارجاع به اندیسهای خارج از بازه آرایه، تولیدِ خطا میباشد:

```
# Looking out of bounds is a BoundsError
try
    a[0] # => ERROR: BoundsError() in getindex at array.jl:270
    a[end+1] # => ERROR: BoundsError() in getindex at array.jl:270
catch e
    println(e)
end
```

خطا های تولید شده در جولیا، نام فایل و شماره خط منبع خود را به همراه دارند، لذا همیشه امکان رجوع به فایل کد و بررسی آن وجود دارد.

```
# Errors list the line and file they came from, even if it's in the standard
# library. If you built Julia from source, you can look in the folder base
# inside the julia folder to find these files.
```

میتوانید یک توالی از اعداد را تبدیل به آرایه نمایید:

```
# You can initialize arrays from ranges
a = [1:5] # => 5-element Int64 Array: [1,2,3,4,5]
```

میتوان از توالی عددی به عنوان اندیس برای استخراج قسمتی از یک آرایه استفاده کرد:

```
# You can look at ranges with slice syntax.
a[1:3] # => [1, 2, 3]
a[2:end] # => [2, 3, 4, 5]
```

تابع splice! اندیس دلخواهی از یک آرایه را حذف میکند:

```
# Remove elements from an array by index with splice!
arr = [3,4,5]
splice!(arr,2) # => 4 ; arr is now [3,5]
```

تابع append! اعضاء دو لیست را ملحق مینماید:

```
# Concatenate lists with append!
b = [1,2,3]
append!(a,b) # Now a is [1, 2, 3, 4, 5, 1, 2, 3]
```

تابع in وجود یک عضو را در لیست بررسی میکند:

```
# Check for existence in a list with in
in(1, a) # => true
```

length تعداد عضوهای لیست را محاسبه میکند:

```
# Examine the length with length
length(a)  # => 8
```

چندتایی ها تغییر ناپذیرند:

```
# Tuples are immutable.
tup = (1, 2, 3) # => (1,2,3) # an (Int64,Int64,Int64) tuple.
tup[1] # => 1
try:
    tup[1] = 3 # => ERROR: no method
setindex!((Int64,Int64,Int64),Int64,Int64)
catch e
    println(e)
end
```

تعداد زیادی از توابع لیستی را میتوان بر چندتایی ها نیز فراخواهند:

```
# Many list functions also work on tuples
length(tup) # => 3
tup[1:2] # => (1,2)
in(2, tup) # => true
```

میتوانید چندتایی را باز کنید و عضوهای آنرا متغییرها نسبت دهید:

```
# You can unpack tuples into variables
a, b, c = (1, 2, 3) # => (1,2,3)  # a is now 1, b is now 2 and c is now 3
```

چندتایی حتی بدون پرانتز ساخته میشود:

```
# Tuples are created even if you leave out the parentheses
d, e, f = 4, 5, 6 # => (4,5,6)
```

مقدار یک چندتایی با تنها یک عضو، با مقدار آن عضو برابر نیست:

```
# A 1-element tuple is distinct from the value it contains
(1,) == 1 # => false
(1) == 1 # => true
```

ببینید مبادله مقادیر چقدر آسان است:

```
# Look how easy it is to swap two values
e, d = d, e  # => (5,4) # d is now 5 and e is now 4
```

واژه نامه ها (کلکسیون، دیکشنری یا آرایه انجمنی) مجموعه از از، نگاشت ها میباشند:

```
# Dictionaries store mappings
empty_dict = Dict() # => Dict{Any,Any}()
```

در واژه نامه ها کلید هر مقدار یک واژه است:

```
# You can create a dictionary using a literal
filled_dict = ["one"=> 1, "two"=> 2, "three"=> 3]
# => Dict{ASCIIString,Int64}
```

ابزار رجوع به واژه ها در واژه نامه، آوردن کلیدواژه در کروشه است:

```
# Look up values with []
filled_dict["one"] # => 1
```

تابع keys یک KeyIterator برای استخراج کلیه مقادیر واژه نامه تولید میکند:

```
# Get all keys
keys(filled_dict)
# => KeyIterator{Dict{ASCIIString,Int64}}(["three"=>3,"one"=>1,"two"=>2])
```

توجه داشته باشید که کلید واژه ها در واژه نامه، مرتب نیستند.

```
# Note - dictionary keys are not sorted or in the order you inserted them.
```

تابع values یک ValueIterator برای استخراج کلیه مقادیر واژه نامه تولید میکند:

```
# Get all values
values(filled_dict)
# => ValueIterator{Dict{ASCIIString,Int64}}(["three"=>3,"one"=>1,"two"=>2])
# Note - Same as above regarding key ordering.
```

بررس وجود یک مقدار در واژه نامه با استفاده از توابع in وhaskey

```
# Check for existence of keys in a dictionary with in, haskey
in(("one", 1), filled_dict) # => true
in(("two", 3), filled_dict) # => false
haskey(filled_dict, "one") # => true
haskey(filled_dict, 1) # => false
```

سعی در استخراج مقدار با کلیدواژه نامعتبر از واژه نامه تولید خطا مینماید:

```
# Trying to look up a non-existant key will raise an error
```

```
try
    filled_dict["four"] # => ERROR: key not found: four in getindex at
dict.jl:489
catch e
    println(e)
end
```

جهت ممانعت از تولید خطا در استخراج واژه ها میتوان از تابع get با تعیین یک مقدار پیشفرض استفاده کرد:

```
# Use the get method to avoid that error by providing a default value
# get(dictionary,key,default_value)
get(filled_dict,"one",4) # => 1
get(filled_dict,"four",4) # => 4
```

برای تعریف مجموعه ای از مقادیر نامرتب و غیر تکراری از ساختار Set استفاده نمایید:

```
# Use Sets to represent collections of unordered, unique values
empty_set = Set() # => Set{Any}()
```

تولید Set از مجموعه از از مقادیر:

```
# Initialize a set with values
filled_set = Set(1,2,2,3,4) # => Set{Int64}(1,2,3,4)
```

افزودن مقدار به Set:

```
# Add more values to a set
push!(filled_set,5) # => Set{Int64}(5,4,2,3,1)
```

بررسی وجود مقدار در Set:

```
# Check if the values are in the set
in(2, filled_set) # => true
in(10, filled_set) # => false
```

توابعی برای محاسبه اشتراکات، محاسبه اختلافات و ملحق نمودن Set ها وجود دارد:

```
# There are functions for set intersection, union, and difference.
other_set = Set(3, 4, 5, 6) # => Set{Int64}(6,4,5,3)
intersect(filled_set, other_set) # => Set{Int64}(3,4,5)
union(filled_set, other_set) # => Set{Int64}(1,2,3,4,5,6)
setdiff(Set(1,2,3,4),Set(2,3,5)) # => Set{Int64}(1,4)
```

```
####################################################
## 3 – ساختارهای کنترلی
####################################################
```

```
####################################################
## 3. Control Flow
####################################################
```

یک متغییر تعریف میکنیم:

```
# Let's make a variable
some_var = 5
```

یک ساختار تصمیم در جولیا به صورت زیر است، فاصله گذاری های ابتدای خطوط تنها جنبه تزیینی دارند:

```
# Here is an if statement. Indentation is not meaningful in Julia.
if some_var > 10
    println("some_var is totally bigger than 10.")
elseif some_var < 10    # This elseif clause is optional.
    println("some_var is smaller than 10.")
else                    # The else clause is optional too.
    println("some_var is indeed 10.")
end
# => prints "some var is smaller than 10"
```

ساختار حلقه ای `for` داده های قابل شمارش (آرایه، توالی عددی، `Set`، دیکشنری و رشته) را تکرار میکند:

```
# For loops iterate over iterables.
# Iterable types include Range, Array, Set, Dict, and String.
for animal=["dog", "cat", "mouse"]
    println("$animal is a mammal")
    # You can use $ to interpolate variables or expression into strings
end
# prints:
#     dog is a mammal
#     cat is a mammal
#     mouse is a mammal

# You can use 'in' instead of '='.
for animal in ["dog", "cat", "mouse"]
    println("$animal is a mammal")
end
# prints:
#     dog is a mammal
#     cat is a mammal
#     mouse is a mammal
```

```
for a in ["dog"=>"mammal","cat"=>"mammal","mouse"=>"mammal"]
    println("$(a[1]) is a $(a[2])")
end
# prints:
#    dog is a mammal
#    cat is a mammal
#    mouse is a mammal

for (k,v) in ["dog"=>"mammal","cat"=>"mammal","mouse"=>"mammal"]
    println("$k is a $v")
end
# prints:
#    dog is a mammal
#    cat is a mammal
#    mouse is a mammal
```

ساختار تکرار while، تا زمانی که شرط آن درست باشد، ادامه می یابد.

```
# While loops loop while a condition is true
x = 0
while x < 4
    println(x)
    x += 1  # Shorthand for x = x + 1
end
# prints:
#   0
#   1
#   2
#   3
```

اداره کردن استثنا:

```
# Handle exceptions with a try/catch block
try
   error("help")
catch e
   println("caught it $e")
end
# => caught it ErrorException("help")
```

```
####################################################
## 4- توابع
####################################################
```

```
####################################################
## 4. Functions
####################################################
```

عبارت function یک تابع جدید تعریف میکند:

توابع مقدار حاصل از اجرا آخرین خط دستورات خود را باز میگردانند:

```
# The keyword 'function' creates new functions
#function name(arglist)
#  body...
#end
function add(x, y)
    println("x is $x and y is $y")

    # Functions return the value of their last statement
    x + y
end

add(5, 6) # => 11 after printing out "x is 5 and y is 6"
```

میتوانید تابعی تعریف کنید که تعداد متغییری از پارامترهای ورودی را دریافت نماید:

از دستور `return` در هر جا از تابع میتوان جهت برگشت استفاده کرد:

```
# You can define functions that take a variable number of
# positional arguments
function varargs(args...)
    return args
    # use the keyword return to return anywhere in the function
end
# => varargs (generic function with 1 method)

varargs(1,2,3) # => (1,2,3)
```

... اسپلت نام دارد، از اسپلت برای تعریف توابع استفاده نمودیم، از اسپلت میتوان در فراخوانی توابع هم استفاده نمود، در چنین شرایطی، اسپلت آرایه یا چندتایی را به لیستی از مقادیر ورودی مبدل میکند:

```
# The ... is called a splat.
# We just used it in a function definition.
# It can also be used in a fuction call,
# where it will splat an Array or Tuple's contents into the argument list.
Set([1,2,3])    # => Set{Array{Int64,1}}([1,2,3]) # produces a Set of Arrays
Set([1,2,3]...) # => Set{Int64}(1,2,3) # this is equivalent to Set(1,2,3)

x = (1,2,3)     # => (1,2,3)
Set(x)          # => Set{(Int64,Int64,Int64)}((1,2,3)) # a Set of Tuples
Set(x...)       # => Set{Int64}(2,3,1)
```

توابع را میتوان با پارامترهای اختیاری دارای مقدار پیش فرض تعریف نمود:

```
# You can define functions with optional positional arguments
function defaults(a,b,x=5,y=6)
    return "$a $b and $x $y"
end
```

```
defaults('h','g') # => "h g and 5 6"
defaults('h','g','j') # => "h g and j 6"
defaults('h','g','j','k') # => "h g and j k"
try
    defaults('h') # => ERROR: no method defaults(Char,)
    defaults() # => ERROR: no methods defaults()
catch e
    println(e)
end
```

میتوان به پارامترهای تابع را با واژه مشخص تعریف نمود، به این ترتیب در هنگام فراخوانی تابع پارامترها فاقد ترتیب بلکه دارای واژه میباشند (به کاربرد ; در تعریف تابع توجه شود) :

```
# You can define functions that take keyword arguments
function keyword_args(;k1=4,name2="hello") # note the ;
    return ["k1"=>k1,"name2"=>name2]
end

keyword_args(name2="ness") # => ["name2"=>"ness","k1"=>4]
keyword_args(k1="mine") # => ["k1"=>"mine","name2"=>"hello"]
keyword_args() # => ["name2"=>"hello","k1"=>4]
```

میتوانید از پارامترهای ترتیبی و واژه گذاری شده در کنار هم برای تریف تابع بهره برید:

```
# You can combine all kinds of arguments in the same function
function all_the_args(normal_arg, optional_positional_arg=2;
keyword_arg="foo")
    println("normal arg: $normal_arg")
    println("optional arg: $optional_positional_arg")
    println("keyword arg: $keyword_arg")
end

all_the_args(1, 3, keyword_arg=4)
# prints:
#   normal arg: 1
#   optional arg: 3
#   keyword arg: 4
```

توابع در جولیا داده هایی از توع تابع میباشند، میتوان آنها را ساخت، به متغییر نسبت دارد و به عنوان پارامتر به توابع ارجاع داد:

```
# Julia has first class functions
function create_adder(x)
    adder = function (y)
        return x + y
    end
    return adder
end
```

تعریف تابع بدون نام با نحوه لامبدا:

```
# This is "stabby lambda syntax" for creating anonymous functions
(x -> x > 2)(3) # => true
```

پیاده سازهی مثال `create_adder` (فوق) به نحو لامبدا:

```
# This function is identical to create_adder implementation above.
function create_adder(x)
    y -> x + y
end
```

میتوان تابع داخلی را نام گذاری نمود:

```
# You can also name the internal function, if you want
function create_adder(x)
    function adder(y)
        x + y
    end
    adder
end

add_10 = create_adder(10)
add_10(3) # => 13
```

توابع داخلی سطح بالایی وجود دارند که یک تابع را به عنوان پارامتر ورودی دریافت مینمایند:

```
# There are built-in higher order functions
map(add_10, [1,2,3]) # => [11, 12, 13]
filter(x -> x > 5, [3, 4, 5, 6, 7]) # => [6, 7]
```

میتوان از مفهوم لیست، برای تبدیل داده ها طبق نقشه استفاده کرد:

```
# We can use list comprehensions for nicer maps
[add_10(i) for i=[1, 2, 3]] # => [11, 12, 13]
[add_10(i) for i in [1, 2, 3]] # => [11, 12, 13]
```

```
####################################################
## 5- انواع داده ها
####################################################
```

```
####################################################
## 5. Types
####################################################
```

انواع داده ها در جولیا از نظام واحدی پیروی میکنند، هر مقداری دارای نوع میباشد و متغییرها به خودی خود، فاقد نوع میباشند.
میتوانید از تابع `typeof` جهت دریافت نوع یک مقدار استفاده نمایید:

```
# Julia has a type system.
# Every value has a type; variables do not have types themselves.
# You can use the `typeof` function to get the type of a value.
typeof(5) # => Int64
```

انواع داده ها مقادیری از نوع DataType میباشند و DataType نوعی داده است که انواع داده ها را از جمله خودش را در بر میگیرد:

```
# Types are first-class values
typeof(Int64) # => DataType
typeof(DataType) # => DataType
# DataType is the type that represents types, including itself.
```

انواع داده ها به منظور تولید اسناد، بهینه سازی کد و در هنگام احضاز استفاده میشوند و بطور کلی توسط جولیا کنترل نمیشوند.

برنامه نویسان جولیا میتوانند انواع داده دلخواه را تعریف کنند، نوع های داده در جولیا مشابه نوع داده ی ساختمان (struct) در c میباشند.

عبارت type به منظور تعریف ساختمان های داده در جولیا استفاده میشود:

```
# Types are used for documentation, optimizations, and dispatch.
# They are not statically checked.

# Users can define types
# They are like records or structs in other languages.
# New types are defined used the `type` keyword.

# type Name
#   field::OptionalType
#   ...
# end
type Tiger
  taillength::Float64
  coatcolor # not including a type annotation is the same as `::Any`
end
```

ساختمانها یه سازنده پیش فرض دارند که مقادیر عناصر ساختمان را به ترتیب معرفی در ساختمان به عنوان ورودی می پذیرد و مقداری از نوع ساختمان با خواص مورد نظر تولید میکند.

```
# The default constructor's arguments are the properties
# of the type, in the order they are listed in the definition
tigger = Tiger(3.5,"orange") # => Tiger(3.5,"orange")
```

نوع مقادیر، نقش سازنده برای نوع مشابه خود را ایفا میکنند.

```
# The type doubles as the constructor function for values of that type
sherekhan = typeof(tigger)(5.6,"fire") # => Tiger(5.6,"fire")
```

ساختمانهای داده در جولیا صلب میباشند به این معنا که مشتق پذیر نیستند و نمیتوانند زیر نوع داشته باشند.

```
# These struct-style types are called concrete types
# They can be instantiated, but cannot have subtypes.
```

انواع انتزاعی داده ها نیز وجود دارد:

```
# The other kind of types is abstract types.

# abstract Name
abstract Cat # just a name and point in the type hierarchy
```

داده های انتزاعی را نمیتوان تولید نمود اما مشتق پذیرند و میتوانند مجموعه ای از انواع ساختمانهای داده یا داده های انتزاعی دیگر را در بر گیرند:

```
# Abstract types cannot be instantiated, but can have subtypes.
# For example, Number is an abstract type
subtypes(Number) # => 6-element Array{Any,1}:
                 #     Complex{Float16}
                 #     Complex{Float32}
                 #     Complex{Float64}
                 #     Complex{T<:Real}
                 #     ImaginaryUnit
                 #     Real
subtypes(Cat) # => 0-element Array{Any,1}
```

همه انواع داده ها نوع مافوق دارند، تابع `super` جهت محاسبه نوع مافوق استفاده میشود:

```
# Every type has a super type; use the `super` function to get it.
typeof(5) # => Int64
super(Int64) # => Signed
super(Signed) # => Real
super(Real) # => Number
super(Number) # => Any
super(super(Signed)) # => Number
super(Any) # => Any
# All of these type, except for Int64, are abstract.
```

یک عملگر جهت تعریف زیر نوع (مشتق) یک نوع داده وجود دارد:

```
# <: is the subtyping operator
type Lion <: Cat # Lion is a subtype of Cat
  mane_color
  roar::String
end
```

همچنین میتوان سازندگان (مولدهای) متفاوتی برای یک ساختمان داده تعریف کرد، برای این کار کافی است تابعی به نام آن ساختمان داده تعریف کنید و یک تابع مولد موجود را فراخوانی نمایید:

```
# You can define more constructors for your type
# Just define a function of the same name as the type
# and call an existing constructor to get a value of the correct type
Lion(roar::String) = Lion("green",roar)
```

به این ترتیب یک مولد جدید برای ساختمان مورد نظر تعریف میگردد، این نوع مولد به دلیل اینکه در بحش تعریفی ساختمان داده گنجانده نشده، مولد خارجی میباشد.

```
# This is an outer constructor because it's outside the type definition

type Panther <: Cat # Panther is also a subtype of Cat
  eye_color
  Panther() = new("green")
  # Panthers will only have this constructor, and no default constructor.
end
```

مولدهای داخلی امکان کنترل چگونگی تولید مقادیر عناصر ساختمان در هنگام تولید یک مقدار از نوع آن ساختمان را فراهم می آورند.

```
# Using inner constructors, like Panther does, gives you control
# over how values of the type can be created.
# When possible, you should use outer constructors rather than inner ones.
```

```
####################################################
## 6 – توزیع (مخابره) پویای توابع
####################################################
```

```
####################################################
## 6. Multiple-Dispatch
####################################################
```

در جولیا تمام توابع دارای نام، جامع میباشند. به این معنا که مجموعه ای از رویه ها را شامل میشوند، در زمان فراخوان تابع، رویه مناسب بصورت پویا مخابره میگردد. این مطلب تنها اختصاص به مولد ها ندارد.

```
# In Julia, all named functions are generic functions
# This means that they are built up from many small methods
# Each constructor for Lion is a method of the generic function Lion.
```

یک تابع `meow` را در نظر بگیرید که وظیفه انتخاب صدای حیوان را عهده دار است:

```
# For a non-constructor example, let's make a function meow:
```

این تابع رویه های متفاوتی برای حیوانات مختلف را شامل میباشد:

```
# Definitions for Lion, Panther, Tiger
function meow(animal::Lion)
  animal.roar # access type properties using dot notation
end

function meow(animal::Panther)
  "grrr"
end

function meow(animal::Tiger)
  "rawwwr"
end
```

تابع را آزمایش نماییم:

```
# Testing the meow function
meow(tigger) # => "rawwr"
meow(Lion("brown","ROAAR")) # => "ROAAR"
meow(Panther()) # => "grrr"
```

بررسی روابط بین انواع داده ها:

```
# Review the local type hierarchy
issubtype(Tiger,Cat) # => false
issubtype(Lion,Cat) # => true
issubtype(Panther,Cat) # => true
```

یک تابع بر گربه ها تعریف می شود:

```
# Defining a function that takes Cats
function pet_cat(cat::Cat)
  println("The cat says $(meow(cat))")
end

pet_cat(Lion("42")) # => prints "The cat says 42"
try
    pet_cat(tigger) # => ERROR: no method pet_cat(Tiger,)
catch e
    println(e)
end
```

در زبانهای شیئ گرا انتخاب رویه مناسب در هنگام فراخوان یک تابع، عموما با توجه به نوع اولین پارامتر (کلاس شیئ) صورت میپذیرد، اما در زبان جولیا کلیه پارامترها در انتخاب بهترین رویه مشارکت دارند.

```
# In OO languages, single dispatch is common;
```

```
# this means that the method is picked based on the type of the first
argument.
# In Julia, all of the argument types contribute to selecting the best
method.
```

برای بررسی تفاوت روشهای فوق، یک تابع با دوپارامتر تعریف میکنیم:

```
# Let's define a function with more arguments, so we can see the difference
function fight(t::Tiger,c::Cat)
  println("The $(t.coatcolor) tiger wins!")
end
# => fight (generic function with 1 method)

fight(tigger,Panther()) # => prints The orange tiger wins!
fight(tigger,Lion("ROAR")) # => prints The orange tiger wins!
```

این رفتار را میتوان در صورتی که به طور خاص گربه، شیر باشد، تغییر داد:

```
# Let's change the behavior when the Cat is specifically a Lion
fight(t::Tiger,l::Lion) = println("The $(l.mane_color)-maned lion wins!")
# => fight (generic function with 2 methods)

fight(tigger,Panther()) # => prints The orange tiger wins!
fight(tigger,Lion("ROAR")) # => prints The green-maned lion wins!
```

و میتوان برای جنگ نیاز به ببر نداشت:

```
# We don't need a Tiger in order to fight
fight(l::Lion,c::Cat) = println("The victorious cat says $(meow(c))")
# => fight (generic function with 3 methods)

fight(Lion("balooga!"),Panther()) # => prints The victorious cat says grrr
try
  fight(Panther(),Lion("RAWR")) # => ERROR: no method fight(Panther,Lion)
catch
end
```

و میتوان وضعیت جدیدی اگر ابتدا گربه و سپس شیر وارد گردند، تعریف نمود:

```
# Also let the cat go first
fight(c::Cat,l::Lion) = println("The cat beats the Lion")
# => Warning: New definition
#    fight(Cat,Lion) at none:1
# is ambiguous with
#    fight(Lion,Cat) at none:2.
# Make sure
#    fight(Lion,Lion)
# is defined first.
#fight (generic function with 4 methods)
```

در چنین شرایطی به دلیل اینکه روشن نمیباشد کدام رویه در هنگام جنگ بین دو شیر باید استفاده گردد، یک پیام اخطار تولید میشود. برای رفع این عیب میتوان جنگ بین دو شیر را به رویه های مختلف جنگیدن اضافه نمود:

```
# This warning is because it's unclear which fight will be called in:
fight(Lion("RAR"),Lion("brown","rarrr")) # => prints The victorious cat says
rarrr
# The result may be different in other versions of Julia

fight(l::Lion,l2::Lion) = println("The lions come to a tie")
fight(Lion("RAR"),Lion("brown","rarrr")) # => prints The lions come to a tie
```

برای اطلاع از آنچه در پس پرده روی میدهد میتوانید کدهایی که توسط ماشین مجری جولیا (llvm) تولید میشود را بررسی نمایید:

```
# Under the hood
# You can take a look at the llvm  and the assembly code generated.

square_area(l) = l * l      # square_area (generic function with 1 method)

square_area(5) #25
```

اگر پارامتر ورودی از نوع صحیح باشد:

```
# What happens when we feed square_area an integer?
code_native(square_area, (Int32,))
        #           .section    __TEXT,__text,regular,pure_instructions
        #       Filename: none
        #       Source line: 1              # Prologue
        #           push    RBP
        #           mov RBP, RSP
        #       Source line: 1
        #           movsxd  RAX, EDI        # Fetch l from memory?
        #           imul    RAX, RAX        # Square l and store the result in
RAX
        #           pop RBP                 # Restore old base pointer
        #           ret                     # Result will still be in RAX

code_native(square_area, (Float32,))
        #           .section    __TEXT,__text,regular,pure_instructions
        #       Filename: none
        #       Source line: 1
        #           push    RBP
        #           mov RBP, RSP
        #       Source line: 1
        #           vmulss  XMM0, XMM0, XMM0  # Scalar single precision
multiply (AVX)
        #           pop RBP
        #           ret

code_native(square_area, (Float64,))
        #           .section    __TEXT,__text,regular,pure_instructions
        #       Filename: none
```

```
        #       Source line: 1
        #           push    RBP
        #           mov RBP, RSP
        #       Source line: 1
        #           vmulsd  XMM0, XMM0, XMM0 # Scalar double precision
multiply (AVX)
        #           pop RBP
        #           ret
        #
# Note that julia will use floating point instructions if any of the
# arguements are floats.
# Let's calculate the area of a circle
circle_area(r) = pi * r * r     # circle_area (generic function with 1
method)
circle_area(5)                  # 78.53981633974483

code_native(circle_area, (Int32,))
        #           .section    __TEXT,__text,regular,pure_instructions
        #       Filename: none
        #       Source line: 1
        #           push    RBP
        #           mov RBP, RSP
        #       Source line: 1
        #           vcvtsi2sd   XMM0, XMM0, EDI          # Load integer (r)
from memory
        #           movabs  RAX, 4593140240              # Load pi
        #           vmulsd  XMM1, XMM0, QWORD PTR [RAX]  # pi * r
        #           vmulsd  XMM0, XMM0, XMM1             # (pi * r) * r
        #           pop RBP
        #           ret
        #

code_native(circle_area, (Float64,))
        #           .section    __TEXT,__text,regular,pure_instructions
        #       Filename: none
        #       Source line: 1
        #           push    RBP
        #           mov RBP, RSP
        #           movabs  RAX, 4593140496
        #       Source line: 1
        #           vmulsd  XMM1, XMM0, QWORD PTR [RAX]
        #           vmulsd  XMM0, XMM1, XMM0
        #           pop RBP
        #           ret
        #
```