Julia | Scientific Programming Language

Medina Padilla Rodrigo Buezo Rojas Iván

Monday, March 6, 2017

Abstract

Programming languages were originally made for scientific computing. This kind of computing has always required the highest performance, yet domain experts have largely moved to slower dynamic languages for daily work. Julia is a flexible dynamic language, appropriate for scientific and numerical computing, with performance comparable to traditional statically-typed languages. This document presents a research made for the *Programming Languages* course at ITAM, about Julia. Analyzing the benefits and disadvantages of the language, including syntax, structure, and logic.

1 Introduction

Julia is a flexible dynamic language, appropriate for scientific and numerical computing, with performance comparable to traditional statically-typed languages. Because Julia's compiler is different from the interpreters used for languages like Python or R, it can seems like Julia's performance is unintuitive at first. When using good practices of programming in Julia code, the script written is nearly as fast as C.

The main characteristics of Julia from typical dynamic languages are:

1. The core language imposes very little; the standard library is written in Julia itself, including primitive operations like integer arithmetic.
2. A rich language of types for constructing and describing objects, that can also optionally be used to make type declarations.
3. The ability to define function behavior across many combinations of argument types via multiple dispatch.
4. Automatic generation of efficient, specialized code for different argument types.
5. **Good performance, approaching that of statically-compiled languages like C.**

Sometimes dynamic languages can be find as typeless, but this is not always true. Every object, wheter primitive or user-defined, has a type. In Julia, types are themselves run-time objects, and can also be used to convey information to the compiler.

Julia aims to create an unprecedented combination of ease-of-use, power, and efficiency in a single language. In addition to the above, some advantages of Julia over comparable systems include:

1. Free and open source (MIT licensed)
2. User-defined types are as fast and compact as built-ins

3. Designed for parallelism and distributed computation
4. Lightweight 'green' threading (coroutines)
5. Unobtrusive yet powerful type system
6. Elegant and extensible conversions and promotions for numeric and other types
7. Efficient support for Unicode, including but not limited to UTF-8
8. Call C functions directly (no wrappers or special APIs needed)
9. Powerful shell-like capabilities for managing other processes
10. Lisp-like macros and other metaprogramming facilities

2 Programming in Julia

This section explains the syntax of Julia like declaring variables, cycles, objects, and examples that are specific from this language.

2.1 Variables

A variable in Julia, as in other programing languages, is a name associated (or bound) to a value. It's useful when you want to store a value for later use. For example:

```
  # Assign the value 10 to the
variable x

  julia> x = 10

  10
```

This Julia code examples has three lines. In the first one the *"#"* symbol is used to begin a comment. This line is going to be ignored by the compiler, and help the programmer to document his code.

The second line has the text *julia> x*, this is a terminal instruction that will always appear when you are in the interactive mode of julia **(this is the only example in the document that this notation will appear, for the next code examples is going to be implicit)**. The Julia command is followed by declaring the name *x* which contains the value *10*. Is important to notice than in Julia is not necessary to declare the type of variable like in C or JAVA.

2.1.1 UNICODE

Julia allows to use certain unicode characters as shown in the next example:

```
U+023F3 = 8
-> 8 # => 8
# These are especially handy for
mathematical notation
2 * pi # => 6.283185307179586
```

2.2 Integers and Floating Point

Integers and floating-point values are the basic building blocks of arithmetic and computation. Built-in representations of such values are called numeric primitives, while representations of integers and floating-point numbers as immediate values in code are known as numeric literals. For example 1 is an integer literal and 1.5 is a floting point literal.

type	signed	number of bits
Int8	X	8
UInt8		8
Int16	X	16
UInt16		16
Int32	X	32
UInt32		32
Int64	X	64
UInt64		64
Int128	X	128
UInt128		128
Bool		8
Integers		

type	precision	number of bits
Float16	half	16
Float32	single	32
Float64	double	64
Floating Point		

2.3 Mathematical Operations and Elementary Functions

Julia provides a complete collection of basic arithmetic and bitwise operators across all of its numeric primitive types, as well as providing portable, efficient implementations of a comprehensive collection of standard mathematical functions.

2.3.1 Arithmetic Operations

The following arithmetic operators are supported on all primitive numeric types:

Expression	Name
+x	unary plus
-x	unary minus
x + y	binary plus
x - y	binary minus
x * y	times
x / y	divide
x \y	inverse divide
x ^y	power

2.3.2 Numeric comparison

Standard comparison operations are defined for all the primitive numeric types:

Operator	Name
==	equality
!=	inequality
<=	less than or equal to
<	less than
>	greater than
>=	greater than or equal to

2.3.3 Operator Precedence

Julia applies the following order of operations, from highest precedence to lowest:

Category	Operators
Syntax	. followed by ::
Exponentiation	^and its elementwise equivalent .^
Fractions	// and .//
Multiplication	* / % & and .* ./ .% .
Bitshifts	<<>>>>> *and.* << . >> . >>>
Addition	+ - | $ and .+ .-
Syntax	: .. followed by |¿

The updating versions of all the binary arithmetic and bitwise operators are:

+= -= *= /= \= ÷= %= ^= &= |= $=

>>>=>>=<<=

2.4 Complex and Rational Numbers

Julia ships with predefined types representing both complex and rational numbers, and supports all standard mathematical operations on them. Conversion and Promotion are defined so that operations on any combination of predefined numeric types, whether primitive or composite, behave as expected.

2.4.1 Complex Numbers

The global constant *im* is bound to the complex number i, representing the principal square root of -1. It was deemed harmful to co-opt the name i for a global constant, since it is such a popular index variable name. Since Julia allows numeric literals to be juxtaposed with identifiers as coefficients, this binding suffices to provide convenient syntax for complex numbers, similar to the traditional mathematical notation:

```
-> 1 + 2im = 10
1 + 2im
```

2.4.2 Rational Numbers

Julia has a rational number type to represent exact ratios of integers. Rationals are constructed using the // operator:

```
-> 2 // 3
2 // 3

-> 6 // 9
2 // 3
```

2.5 Arrays

To initialize an array you can use the = operator along with [ ] no set the n elements. It is important to remember that every array index begins in number 1.

```
-> a = [1,4,2,7,3,5]
4-element Array{Int64,1}:
1
4
2
7
3
5

-> a = [1 4 2 7 3 5]
1x4 ArrayInt64,2:
1 2 3 5

->a = [1 2; 2 4; 5 6]
3»2 ArrayInt64,2:
1 2
2 4
5 6
```

2.5.1 Basic functions

ndims(A::AbstractArray) > Integer Returns the number of dimensions of A.

```
-> A = ones(3,4,5);

-> ndims(A)
3
```

size(A::AbstractArray[, dim...]) Returns a tuple containing the dimensions of A. Optionally you can specify the dimension(s) you want the length of, and get the length of that dimension, or a tuple of the lengths of dimensions you asked for.

```
-> A = ones(2,3,4);

-> size(A, 2)
3

-> size(A,3,2)
(4,3)
```

2.5.2 Indexing, Assignment, and Concatenation

getindex(A, inds...) Returns a subset of array A as specified by inds, where each ind may be an Int, a Range, or a Vector. To acces an array you can use [index] where index is a number n. Julia's arrays begin indexing from number 1 to the length of the array, or you can use the notation 1:5, which is read as "from 1 to 5".

view(A, inds...) Like getindex(), but returns a view into the parent array A with the given indices instead of making a copy. Calling getindex() or setindex!() on the returned SubArray computes the indices to the parent array on the fly without checking bounds.

@view A[inds...] Creates a SubArray from an indexing expression. This can only be applied directly to a reference expression (e.g. @view A[1,2:end]), and should not be used as the target of an assignment (e.g. @view(A[1,2:end]) = ...).

parent(A) Returns the "parent array" of an array view type (e.g., SubArray), or the array itself if it is not a view.

parentindexes(A) From an array view A, returns the corresponding indexes in the parent.

slicedim(A, d, i) Return all the data of A where the index for dimension d equals i. Equivalent to A[:,:,...,i,:,:,...] where i is in position d.

setindex!(A, X, inds...) Store values from array X within some subset of A as specified by inds.

isassigned(array, i) - Bool Tests whether the given array has a value associated with index i. Returns false if the index is out of bounds, or has an undefined reference.

cat(dims, A...) Concatenate the input arrays along the specified dimensions in the iterable dims. For dimensions not in dims, all input arrays should have the same size, which will also be the size of the output array along that dimension. For dimensions in dims, the size of the output array is the sum of the sizes of the input arrays along that dimension. If dims is a single number, the different arrays are tightly stacked along that dimension. If dims is an iterable containing several dimensions, this allows one to construct block diagonal matrices and their higher-dimensional analogues by simultaneously increasing several dimensions for every new input array and putting zero blocks elsewhere. For example, cat([1,2], matrices...) builds a block diagonal matrix, i.e. a block matrix with matrices[1], matrices[2], ... as diagonal blocks and matching zero blocks away from the diagonal.

vcat(A...) Concatenate along dimension 1.

```
-> a = [1 2 3 4 5]
1»5 ArrayInt64,2:
1 2 3 4 5

-> b = [6 7 8 9 10; 11 12 13 14
15]
2»5 ArrayInt64,2:
6 7 8 9 10
11 12 13 14 15

-> vcat(a,b)
3»5 ArrayInt64,2:
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
```

```
-> c = ([1 2 3], [4 5 6])
(
[123]
[456])

-> vcat(c...)
2»3 ArrayInt64,2:
1 2 3
4 5 6
```

2.6 Strings

Strings are finite sequences of characters. Of course, the real trouble comes when one asks what a character is. The characters that English speakers are familiar with are the letters A, B, C, etc., together with numerals and common punctuation symbols. These characters are standardized together with a mapping to integer values between 0 and 127 by the ASCII standard. There are, of course, many other characters used in non-English languages, including variants of the ASCII characters with accents and other modifications, related scripts such as Cyrillic and Greek, and scripts completely unrelated to ASCII and English, including Arabic, Chinese, Hebrew, Hindi, Japanese, and Korean.

2.6.1 Characters

A Char value represents a single character: it is just a 32-bit bitstype with a special literal representation and appropriate arithmetic behaviors, whose numeric value is interpreted as a Unicode code point. Here is how Char values are input and shown:

```
-> 'x'
'x'

-> typeof(ans)
Char
```

You can convert a Char to its integer value, i.e. code point, easily:

```
-> Int( 'x' )
'x'

-> typeof(ans)
Int64
```

You can do comparisons and a limited amount of arithmetic with Char values:

```
-> 'A' < 'a' true

-> 'A' <= 'a' <= 'Z' false

-> 'A' <= 'X' <= 'Z' true

-> 'x' - 'a' 23

-> 'A' + 1 'B'
```

2.6.2 String basics

String literals are delimited by double quotes or triple double quotes:

```
-> str = "Hello, world.  \n"
"Hello, world."
```

If you want to extract a character from a string, you index into it:

```
-> str[1] 'H'

-> str[6] ','

-> str[end] '\n'
```

All indexing in Julia is 1-based: the first element of any integer-indexed object is found at

index 1, and the last element is found at index n, when the string has a length of n.

In any indexing expression, the keyword end can be used as a shorthand for the last index (computed by endof(str)). You can perform arithmetic and other operations with end, just like a normal value:

```
-> str[end-1] '.'

  -> str[end÷2] ' '
```

2.6.3 Interpolation

One of the most common and useful string operations is concatenation:

```
-> greet = "Hello" "Hello"

  -> whom = "world" "world"

  -> string(greet, ", ", whom,
".\n") "Hello, world.\n"
```

Constructing strings like this can become a bit cumbersome, however. To reduce the need for these verbose calls to string(), Julia allows interpolation into string literals using $, as in Perl:

```
-> greet = "Hello"
"Hello"

-> whom = "world"
"world"

-> string(greet, ", ", whom, ".\n")
"Hello, world.\n"
```

You can lexicographically compare strings using the standard comparison operators:

```
-> "abracadabra" < "xylophone"
true

-> "abracadabra" == "xylophone"
false

-> "Hello, world." != "Goodbye,
world."
true

-> "1 + 2 = 3" == "1 + 2 = $(1 +
2)"
true
```

2.6.4 Regular Expressions

Julia has Perl-compatible regular expressions (regexes), as provided by the PCRE library. Regular expressions are related to strings in two ways: the obvious connection is that regular expressions are used to find regular patterns in strings; the other connection is that regular expressions are themselves input as strings, which are parsed into a state machine that can be used to efficiently search for patterns in strings. In Julia, regular expressions are input using non-standard string literals prefixed with various identifiers beginning with r. The most basic regular expression literal without any options turned on just uses r"...":

```
-> r"^\s*(?:#|$)"
r"^\s*(?:#|$)"

-> typeof(ans)
Regex
```

2.7 Functions

In Julia, a function is an object that maps a tuple of argument values to a return value. Julia functions are not pure mathematical functions, in the sense that functions can alter and be af-

fected by the global state of the program. The basic syntax for defining functions in Julia is:

```
function f(x,y)
x + y
end
```

There is a second, more terse syntax for defining a function in Julia. The traditional function declaration syntax demonstrated above is equivalent to the following compact "$assignment form$":

```
f(x,y) = x + y
```

2.7.1 Argument Passing Behavior

Julia function arguments follow a convention sometimes called "pass-by-sharing", which means that values are not copied when they are passed to functions. Function arguments themselves act as new variable bindings (new locations that can refer to values), but the values they refer to are identical to the passed values. Modifications to mutable values (such as Arrays) made within a function will be visible to the caller. This is the same behavior found in Scheme, most Lisps, Python, Ruby and Perl, among other dynamic languages.

The value returned by a function is the value of the last expression evaluated, which, by default, is the last expression in the body of the function definition. In the example function, f, from the previous section this is the value of the expression x + y. As in C and most other imperative or functional languages, the return keyword causes a function to return immediately, providing an expression whose value is returned:

```
g(x,y)
return x * y
end
```

In Julia, one returns a tuple of values to simulate returning multiple values. However, tuples can be created and destructured without needing parentheses, thereby providing an illusion that multiple values are being returned, rather than a single tuple value. For example, the following function returns a pair of values:

```
g(x,y)
return x * y, x + y
end
```

2.8 Scope of Variables

The scope of a variable is the region of code within which a variable is visible. Variable scoping helps avoid variable naming conflicts. The concept is intuitive: two functions can both have arguments called x without the two x's referring to the same thing. Similarly there are many other cases where different blocks of code can use the same name without referring to the same thing. The rules for when the same variable name does or doesn't refer to the same thing are called scope rules; this section spells them out in detail.

Certain constructs in the language introduce scope blocks, which are regions of code that are eligible to be the scope of some set of variables. The scope of a variable cannot be an arbitrary set of source lines; instead, it will always line up with one of these blocks. There are two main types of scopes in Julia, global scope and local scope, the latter can be nested.

Julia uses lexical scoping, meaning that a function's scope does not inherit from its caller's scope, but from the scope in which the function was defined. For example, in the following code the x inside foo refers to the x in the global scope of its module Bar:

```
module Bar
```

```
x = 1
foo() = x
end
```

and not a x in the scope where foo is used:

```
-> import Bar
-> x = -1
-> Bar.foo()
1
```

Thus lexical scope means that the scope of variables can be inferred from the source code alone.

2.9 Constructors

Constructor are functions that create new objects specifically, instances of Composite Types. In Julia, type objects also serve as constructor functions: they create new instances of themselves when applied to an argument tuple as a function. For example:

```
-> type Foo
bar
baz
end
1

-> foo = Foo(1,2)
Foo(1,2)
-> foo.bar
1


-> foo.baz
2
```

2.10 Modules

Modules in Julia are separate variable workspaces, i.e. they introduce a new global scope. They are delimited syntactically, inside module Name ... end. Modules allow you to create top-level definitions (aka global variables) without worrying about name conflicts when your code is used together with somebody else's. Within a module, you can control which names from other modules are visible (via importing), and specify which of your names are intended to be public (via exporting).

```
module MyModule
using Lib

using BigLib:  thing1, thing2

import Base.show

importall OtherLib

export MyType, foo

type MyType
x
end

bar(x) = 2x
foo(a::MyType) = bar(a.x) + 1

show(io::IO, a::MyType) = print(io,
"MyType $(a.x)")
end
```

Note that the style is not to indent the body of the module, since that would typically lead to whole files being indented.

This module defines a type MyType, and two functions. Function foo and type MyType are exported, and so will be available for importing into other modules. Function bar is private to MyModule.

The statement using Lib means that a module called Lib will be available for resolving names as needed. When a global variable is encountered that has no definition in the current module, the system will search for it among variables exported by Lib and import it if it is found there. This means that all uses of that global within the current module will resolve to the definition of that variable in Lib.

The statement using BigLib: thing1, thing2 is a syntactic shortcut for using BigLib.thing1, BigLib.thing2.

The import keyword supports all the same syntax as using, but only operates on a single name at a time. It does not add modules to be searched the way using does. import also differs from using in that functions must be imported using import to be extended with new methods.

In MyModule above we wanted to add a method to the standard show function, so we had to write import Base.show. Functions whose names are only visible via using cannot be extended.

The keyword importall explicitly imports all names exported by the specified module, as if import were individually used on all of them.

Once a variable is made visible via using or import, a module may not create its own variable with the same name. Imported variables are read-only; assigning to a global variable always affects a variable owned by the current module, or else raises an error.

2.11 Control Flow

Julia provides a variety of control flow constructs:

Compound Expressions: begin and (;).
Conditional Evaluation: if-elseif-else and ?: (ternary operator).
Short-Circuit Evaluation: &&, || and chained comparisons.
Repeated Evaluation: Loops: while and for.
Exception Handling: try-catch, error() and throw().
Tasks (aka Coroutines): yieldto().
The first five control flow mechanisms are standard to high-level programming languages. Tasks are not so standard: they provide non-local control flow, making it possible to switch between temporarily-suspended computations. This is a powerful construct: both exception handling and cooperative multitasking are implemented in Julia using tasks. Everyday programming requires no direct usage of tasks, but certain problems can be solved much more easily by using tasks.

2.11.1 Compound Expression

Sometimes it is convenient to have a single expression which evaluates several subexpressions in order, returning the value of the last subexpression as its value. There are two Julia constructs that accomplish this: begin blocks and (;) chains. The value of both compound expression constructs is that of the last subexpression. Here?s an example of a begin block:

```
-> z = begin
x = 1
y = 2
x + y
end
3
```

Since these are fairly small, simple expressions, they could easily be placed onto a single line, which is where the (;) chain syntax comes in handy:

```
-> begin x = 1; y = 2; x + y end
```

2.11.2 Conditional Evaluation

Conditional evaluation allows portions of code to be evaluated or not evaluated depending on the value of a boolean expression. Here is the anatomy of the if-elseif-else conditional syntax:

```
if x < y
println("x is less than y")
elseif x > y
println("x is greater than y")
else
println("x is equal to y")
end
```

2.11.3 Short-Circuit Evaluation

Short-circuit evaluation is quite similar to conditional evaluation. The behavior is found in most imperative programming languages having the && and || boolean operators: in a series of boolean expressions connected by these operators, only the minimum number of expressions are evaluated as are necessary to determine the final boolean value of the entire chain. Explicitly, this means that:

```
-> t(x) = (println(x); true)
t (generic function with 1 method)

->f(x) = (println(x); false)
f (generic function with 1 method)

-> t(1) && t(2)
1
2
true
```

2.11.4 Repeared Evaluation: Loops

There are two constructs for repeated evaluation of expressions: the while loop and the for loop. Here is an example of a while loop:

```
-> i = 1
->while i <= 5
println(i)
i += 1
end
1
2
3
4
5
```

The while loop evaluates the condition expression (i ¡= 5 in this case), and as long it remains true, keeps also evaluating the body of the while loop. If the condition expression is false when the while loop is first reached, the body is never evaluated.

The for loop makes common repeated evaluation idioms easier to write. Since counting up and down like the above while loop does is so common, it can be expressed more concisely with a for loop:

```
-> for i = 1:5
println(i)
end
1
2
3
4
5
```

2.11.5 The try/catch statement

The try/catch statement allows for Exceptions to be tested for. For example, a customized square root function can be written to automatically call either the real or complex square root method on demand using Exceptions :

```
-> f(x) = try
sqrt(x)
catch
sqrt(complex(x, 0))
end
f (generic function with 1 method)

-> f(1)
1.0

-> f(-1)
0.0 + 1.0im
```

It is important to note that in real code computing this function, one would compare x to zero instead of catching an exception. The exception is much slower than simply comparing and branching.

2.12 Type

Type systems have traditionally fallen into two quite different camps: static type systems, where every program expression must have a type computable before the execution of the program, and dynamic type systems, where nothing is known about types until run time, when the actual values manipulated by the program are available. Object orientation allows some flexibility in statically typed languages by letting code be written without the precise types of values being known at compile time. The ability to write code that can operate on different types is called polymorphism. All code in classic dynamically typed languages is polymorphic: only by explicitly checking types, or when objects fail to support operations at run-time, are the types of any values ever restricted.

Julia's type system is dynamic, but gains some of the advantages of static type systems by making it possible to indicate that certain values are of specific types. This can be of great assistance in generating efficient code, but even more significantly, it allows method dispatch on the types of function arguments to be deeply integrated with the language. Method dispatch is explored in detail in Methods, but is rooted in the type system presented here.

The :: operator can be used to attach type annotations to expressions and variables in programs. There are two primary reasons to do this:

As an assertion to help confirm that your program works the way you expect,
To provide extra type information to the compiler, which can then improve performance in some cases.

When appended to an expression computing a value, the :: operator is read as "is an instance of". It can be used anywhere to assert that the value of the expression on the left is an instance of the type on the right. When the type on the right is concrete, the value on the left must have that type as its implementation - recall that all concrete types are final, so no implementation is a subtype of any other. When the type is abstract, it suffices for the value to be implemented by a concrete type that is a subtype of the abstract type. If the type assertion is not true, an exception is thrown, otherwise, the left-hand value is returned:

```
-> > (1+2)::AbstractFloat
ERROR: TypeError: typeassert: ex-
pected AbstractFloat, got Int64
...
```

```
-> (1+2)::Int
3
```

This allows a type assertion to be attached to any expression in-place.

When appended to a variable on the left-hand side of an assignment, or as part of a local declaration, the :: operator means something a bit different: it declares the variable to always have the specified type, like a type declaration in a statically-typed language such as C. Every value assigned to the variable will be converted to the declared type using convert():

```
-> function foo() x::Int8 = 100
x
end
foo (generic function with 1
method)
->foo()
100
->typeof(ans)
Int8
```

2.12.1 User Defined Methods

You can define custom types using the keyword 'type'.

type Employee firstName::ASCIIString lastName::ASCIIString id::Int64 end

Above snippet create new type Employee, it stores firstName, lastName and id of an employee. You can access the properties of employee using '.' Notation.

emp1=Employee("Hari Krishna", "Gurram", 1) Above statement defines employee emp1 with firstName "Hari Krishna", lastName "Gurram" and id 1.

```
->workspace()
type Employee
firstName::String
lastName::String
id::Int64
end
```

Custom type is similar to a class without methods in java. Custom types created with the keyword ?type? are mutable, i.e, you can change the values of these type.

```
->emp1=Employee("John", "Frus-
ciante", 1)
->println(emp1.firstName)
John
->println(emp1.lastName)
Frusciante
->println(emp1.id)
1
```

Julia pass the objects to function by reference, so any changes you made to the object inside a function are visible outside also.

```
  # User defined types can be
changed

-> emp1.firstName="Frederick"
emp1.lastName="Chopin"
emp1
Employee("Frederick","Chopin",1)
```

2.12.2 Multiple Dispatch

There is a very important topic concerning Julia's data model, methods and multiple dispatch

Look at the error message:

```
->100 + "100"
```

```
ERROR: `+` has no method matching
+(::Int64, ::ASCIIString)
```

You can rewrite that call using functional notation to obtain exactly the same result.

```
->+(100, "100") ERROR: `+` has no
method matching +(::Int64, ::ASCI-
IString)
```

Multiplication is similar

```
-> 100 * "100"
ERROR: `*` has no method matching
*(::Int64, ::ASCIIString)

-> *(100, "100")
ERROR: `*` has no method matching
*(::Int64, ::ASCIIString)
```

What the message tells is that *(a, b) doesn't work when a is an integer and b is a string

In particular, the function * has no matching method

In essence, a method in Julia is a version of a function that acts on a particular tuple of data types

For example, if a and b are integers then a method for multiplying integers is invoked

```
-> *(100, 100)
10000
```

On the other hand, if a and b are strings then a method for string concatenation is invoked

```
-> *("foo", "bar")
"foobar"
```

In fact we can see the precise methods being invoked by applying @which

```
->@which *(100, 100)
(x::Int64,y::Int64) at int.jl:47

-> @which *("foo", "bar")
(s1::AbstractString,
ss::AbstractString...)  at
strings/basic.jl:50
```

You can see the same process with other functions and their methods

```
->isfinite(1.0)
# Call isfinite on a float
true

-> @which isfinite(1)
isfinite(x::Integer) at
float.jl:311

-> @which isfinite(1.0)
isfinite(x::AbstractFloat) at
float.jl:309
```

2.12.3 Immutable types

Some times you want to create immutable types. You can create immutable type using the keyword immutable.

```
immutable Student
id::Int64
name::String
end

stud1=Student(1, "Bill")
```