LabMeeting20150327_IntroJuliaPart2

March 27, 2015

0.1 Hello World Revisit

```
In [157]: println("Hello World")
          println("%s", "Hello World")
```

```
Hello World
%sHello World
```

Alternative way, use macro **@print**

```
In [158]: @printf "%s\n" "Hello World"
          @printf "Macro: %s %s\n" "Hello" "World"

          number = 12
          text = "Hello"
          text2 = "Worlds"
          @printf "%s %d %s\n" text number "$text2"
```

```
Hello World
Macro: Hello World
Hello 12 Worlds
```

0.2 Basic Functions

```
In [159]: square(x) = x*x
          println(square(4))
          println(square(5))
```

```
16
25
```

```
In [160]: function x²(x)
              return x*x
          end
          #x\^2 tab
          x²(8)
```

```
Out[160]: 64
```

0.2.1 Keyword Arguments with semicolon ;

```
In [161]: function xʸ( ; x=null, y=null)
              return x^y
          end

          println( xʸ(x=4, y=3) )
          println( xʸ(y=4, x=3) )
```

```
64
81
```

What happen without the semicolon ; ?

```
In [162]: function power(x=null, y=null)
              return x^y
          end
          println( power(4,3) )
          println( power(x=4, y=3) ) #error here!
```

```
64


        function power does not accept keyword arguments
    while loading In[162], in expression starting on line 5
```

0.2.2 Varargs Functions with ...

This allows variable number of arguments.

```
In [163]: function ∑(arg...)
              sum = 0
              for i in arg
                  sum+=i
              end
              return sum
          end

          println( ∑(3,4,5) )
          println( ∑(1,2,3,4,5,6,7,8,9) )
```

```
12
45
```

0.3 Coding convention - Append ! to names of functions that modify their arguments

http://julia.readthedocs.org/en/latest/manual/style-guide/#append-to-names-of-functions-that-modify-their-arguments

```
In [164]: arr = rand(5)
          println("Orig:\t", arr)

          sort(arr)
          println("sort:\t", arr)## order unchanged

          sort!(arr)
          println("sort!:\t", arr)
```

```
Orig:        [0.7949082860194716,0.3487890794155215,0.35763668258127046,0.00042488823133357556,0.1298514
sort:        [0.7949082860194716,0.3487890794155215,0.35763668258127046,0.00042488823133357556,0.1298514
sort!:        [0.00042488823133357556,0.1298514591993345,0.3487890794155215,0.35763668258127046,0.794908
```

0.4 Unittest

@test(ex) Test the expression `ex` and calls the current handler to handle the result.

@test_throws(extype, ex) Test that the expression `ex` throws an exception of type `extype` and calls the current handler to handle the result.

@test_approx_eq(a, b) Test two floating point numbers `a` and `b` for equality taking in account small numerical errors.

@test_approx_eq_eps(a, b, tol) Test two floating point numbers `a` and `b` for equality taking in account a margin of tolerance given by `tol`.

```
In [165]: using Base.Test

          @test 1 == 1

In [166]: @test 1 == 0
```

```
        test failed: 1 == 0
    while loading In[166], in expression starting on line 1
```

```
In [167]: @test_throws ErrorException error("An error")  #pass

In [168]: @test_throws BoundsError error("An error")  #fail
```

```
        test failed: error("An error")
    while loading In[168], in expression starting on line 1
```

```
In [169]: @test_throws DomainError throw(DomainError()) #pass

In [170]: @test_throws DomainError throw(EOFError()) #fail
```

```
        test failed: throw(EOFError())
    while loading In[170], in expression starting on line 1
```

```
In [171]: @test_approx_eq 1. 0.999999999 #fail
```

```
        assertion failed: |1.0 - 0.999999999| <= 2.220446049250313e-12
      1.0 = 1.0
      0.999999999 = 0.999999999
      difference = 9.999999717180685e-10 > 2.220446049250313e-12
    while loading In[171], in expression starting on line 1
```

```
In [172]: @test_approx_eq 1. 0.9999999999999 #pass
In [173]: @test_approx_eq_eps 1. 0.999 1e-2  #pass

In [174]: @test_approx_eq_eps 1. 0.999 1e-4  #fail
```

```
        assertion failed: |1.0 - 0.999| <= 0.0001
      1.0 = 1.0
      0.999 = 0.999
      difference = 0.0010000000000000009 > 0.0001
    while loading In[174], in expression starting on line 1
```

0.5 Type (class-like)

http://julia.readthedocs.org/en/latest/manual/types/

Describing Julia in the lingo of type systems, it is: **dynamic, nominative and parametric.**

Generic types can be parameterized, and the hierarchical relationships between types are explicitly declared, rather than implied by compatible structure. One particularly distinctive feature of Julia's type system is that concrete types may not subtype each other: all concrete types are final and may only have abstract types as their supertypes.

- **Dynamic (wiki):** Dynamic type-checking and runtime type information Dynamic type-checking is the process of verifying the type safety of a program at runtime. Implementations of dynamically type-checked languages generally associate each runtime object with a "type tag" (i.e., a reference to a type) containing its type information.
- **Nominative (wiki):** In computer science, a nominal or nominative type system (or name-based type system) is a major class of type system, in which compatibility and equivalence of data types is determined by explicit declarations and/or the name of the types. Nominal systems are used to determine if types are equivalent, as well as if a type is a subtype of another. It contrasts with structural systems, where comparisons are based on the structure of the types in question and do not require explicit declarations.

  The types Bool, Int8 and UInt8 all have identical representations: they are eight-bit chunks of memory. Since Julia's type system is nominative, however, they are not interchangeable despite having identical structure. Another fundamental difference between them is that they have different supertypes: Bool‘s direct supertype is Integer, Int8‘s is Signed, and UInt8‘s is Unsigned. All other differences between Bool, Int8, and UInt8 are matters of behavior — the way functions are defined to act when given objects of these types as arguments. This is why a nominative type system is necessary: if structure determined type, which in turn dictates behavior, then it would be impossible to make Bool behave any differently than Int8 or UInt8.t
- **Parametric (julia):** Types can take parameters, so that type declarations actually introduce a whole family of new types — one for each possible combination of parameter values.

```
In [175]: println(  Integer <: Number  )
          println(  Integer <: FloatingPoint  )
          println(  issubtype(Integer, Number)  )
          println(  issubtype(Integer, FloatingPoint)  )
```

```
true
false
true
false
```

```
In [176]: println(  super(Integer)  )
          println(  super(Real)  )

Real
Number
```

```
In [177]: subtypes(Number)

Out[177]: 2-element Array{Any,1}:
           Complex{T<:Real}
           Real
```

```
In [178]: subtypes(Real)

Out[178]: 4-element Array{Any,1}:
           FloatingPoint
           Integer
           MathConst{sym}
           Rational{T<:Integer}
```

```
In [179]: println(  isa(1, Number)  )
          println(  isa(1.1, Integer)  )

true
false
```

0.5.1 Parametric Types

```
In [180]: function test{T <: Any}(a::T)
              println("$a is a $T")
          end

          test(3)
          test(3.2)
          test(1:3)
          test(22//7)
          test("test")

3 is a Int64
3.2 is a Float64
1:3 is a UnitRange{Int64}
22//7 is a Rational{Int64}
test is a ASCIIString
```

```
In [181]: function testType{T <: Int}(a::T)
              println("$a is a Int")
          end

          function testType{T <: Number}(a::T)
              println("$a is a Number")
          end

          function testType{T <: String}(a::T)
              println("$a is a String")
          end
          testType(3)
          testType(3.2)
          testType(22//7)
          testType("this")
          testType(1:3) ## Error! has no method matching testType(::UnitRange{Int64})
```

```
3 is a Int
3.2 is a Number
22//7 is a Number
this is a String


        `testType` has no method matching testType(::UnitRange{Int64})
    while loading In[181], in expression starting on line 17
```

```
In [182]: methods(testType) ## Find out all methods assocated with testType
```

```
Out[182]: # 3 methods for generic function "testType":
          testType{T<:Int64}(a::T<:Int64) at In[181]:2
          testType{T<:Number}(a::T<:Number) at In[181]:6
          testType{T<:String}(a::T<:String) at In[181]:10
```

0.5.2 abstract and concrete types

Julia's type system is that concrete types may not subtype each other: all concrete types are final and may only have abstract types as their supertypes.

```
In [183]: abstract Person

          type Postdoc <: Person
              id::Int64
          end

          p1 = Postdoc(101)
          println(p1)
          println("super(Postdoc):", super(Postdoc))
```

```
Postdoc(101)
super(Postdoc):Person
```

```
In [184]: abstract Minion

          type Postdoc <: Minion
                  id::Int64
                  name::String
                  project::String
                  Postdoc(id, name, project) = new(id, name, project)
                  Postdoc(id, name) = new(id, name, "Nothing to do")
          end
          Postdoc(id) = Postdoc(id, "No Name", "Nothing to do")

          ## you will get "invalid redefinition of constant Postdoc"
```

```
        invalid redefinition of constant Postdoc
    while loading In[184], in expression starting on line 3
```

0.5.3 Using module part 1

Often you will get "Error: invalid redefinition of constant Postdoc" or something similar
http://julia.readthedocs.org/en/latest/manual/faq/?highlight=redefine
http://julia.readthedocs.org/en/latest/manual/modules/
use `module` to redefine this. Read about `using`, `import`, `export`

In [185]:
```
module MinionModule
#http://julia.readthedocs.org/en/latest/manual/modules/
# using vs import

abstract Minion ## abstract type

function printMinion(p)
    println("print: \t", p.id, "\t", p.name, "\t", p.project)
end

function getID(p::Minion)
        return p.id
end

type Postdoc <: Minion
        id::Int64
        name::String
        project::String
        Postdoc(id, name, project) = new(id, name, project)
        Postdoc(id, name) = new(id, name, "Nothing to do")
end
Postdoc(id) = Postdoc(id, "No Name", "Nothing to do")
## multiple constructors


type Student <: Minion
        id::Int64
        name::String
        project::String
        Student(id, name, project) = new(id, name, project)
        Student(id) = new(id, "No Name", "Nothing to do")
end

end
```

```
Warning: replacing module MinionModule
```

In [186]:
```
using MinionModule

println( super(MinionModule.Postdoc) )
println( super(MinionModule.Student) )
println( super(MinionModule.Minion) )
```

```
Minion
Minion
Any
```

```
In [187]: using MinionModule

          p1 = MinionModule.Postdoc(101)
          p2 = MinionModule.Postdoc(102, "Name2")
          p3 = MinionModule.Postdoc(103, "Name3", "work hard")

          println("ID: ",MinionModule.getID(p1))
          println("ID: ",MinionModule.getID(p2))
          println("ID: ",MinionModule.getID(p3))

          MinionModule.printMinion(p1)
          MinionModule.printMinion(p2)
          MinionModule.printMinion(p3)

          s1 = MinionModule.Student(201)
          s2 = MinionModule.Student(202)

          println("ID: ",MinionModule.getID(s1))
          MinionModule.printMinion(s1)
          println("ID: ",MinionModule.getID(s2))
          MinionModule.printMinion(s2)
```

```
ID: 101
ID: 102
ID: 103
print:         101        No Name        Nothing to do
print:         102        Name2        Nothing to do
print:         103        Name3        work hard
ID: 201
print:         201        No Name        Nothing to do
ID: 202
print:         202        No Name        Nothing to do
```

0.5.4 Using module part 2

Let's add a few more types to this module.

```
In [188]: module MinionModule
          #http://julia.readthedocs.org/en/latest/manual/modules/
          # using vs import

          abstract Minion ## abstract type

          function printMinion(p) ## Take all type
              println("print: \t", p.id, "\t", p.name, "\t", p.project)
          end

          function getID(p::Minion) ## only take Minion type
                  return p.id
          end
          type Postdoc <: Minion
                  id::Int64
                  name::String
                  project::String
                  Postdoc(id, name, project) = new(id, name, project)
                  Postdoc(id, name) = new(id, name, "Nothing to do")
          end
          Postdoc(id) = Postdoc(id, "No Name", "Nothing to do")
          ## multiple constructors


          type Student <: Minion
                  id::Int64
                  name::String
                  project::String
                  Student(id, name, project) = new(id, name, project)
                  Student(id) = new(id, "No Name", "Nothing to do")
          end

          ##Visitor is belong to ::Any
          type Visitor
                  id::Int64
                  name::String
                  project::String
          end

          end
```

```
Warning: replacing module MinionModule
```

The `Visitor` type does not belong to `Minion`. So it will work with `printMinion()` but **NOT** `getID()`

```
In [189]: using MinionModule
          v1 = MinionModule.Visitor(800, "V1", "N/A")
          MinionModule.printMinion(v1)
          MinionModule.getID(v1)# ERROR!
          #This is **NOT** what do we want. let's change it below
```

```
print:         800        V1        N/A


        ‘getID‘ has no method matching getID(::Visitor)
    while loading In[189], in expression starting on line 4
```

0.5.5 Using module part 3

Let's add some another `abstract` class call `person`. And change the type for `printMinion` and `getID`.

And let's also add a function `minionType` associate with each type.

```
In [190]: module MinionModule
          #http://julia.readthedocs.org/en/latest/manual/modules/
          # using vs import
```

```
abstract Person ## abstract type
abstract Minion <: Person

function printMinion(p::Minion)
    println("print: \t", p.id, "\t", p.name, "\t", p.project)
end

function getID{T <: Person}(p::T)
        return p.id
end

type Postdoc <: Minion
        id::Int64
        name::String
        project::String
        Postdoc(id, name, project) = new(id, name, project)
        Postdoc(id, name) = new(id, name, "Nothing to do")
end
Postdoc(id) = Postdoc(id, "No Name", "Nothing to do")
## multiple constructors


type Student <: Minion
        id::Int64
        name::String
        project::String
        Student(id, name, project) = new(id, name, project)
        Student(id) = new(id, "No Name", "Nothing to do")
end

##Visitor is belong to ::Person
type Visitor <: Person
        id::Int64
        name::String
        project::String
end


function minionType(p::Minion)
    "Minion ", p.id # access type properties using dot notation
end

function minionType(p::Student)
    "Student", p.id
end

function minionType{T <: Person}(p::T)
    "Person", p.id
end


end
```

```
Warning: replacing module MinionModule
```

```
In [191]: using MinionModule


          p3 = MinionModule.Postdoc(103, "Name3", "work hard")

          println("ID: ",MinionModule.getID(p3))
          MinionModule.printMinion(p3)

          s1 = MinionModule.Student(201)
          println("ID: ",MinionModule.getID(s1))
          MinionModule.printMinion(s1)


          v1 = MinionModule.Visitor(800, "V1", "N/A")
          print("ID: ",MinionModule.getID(v1) )
          MinionModule.printMinion(v1) # ERROR! has no method matching printMinion(::Visitor)
          #This is what we expected, getID works on all ‘Person‘ but printMinion only works no ‘Minion‘
          #Visitor is not a Minion
```

```
ID: 103
print:         103        Name3        work hard
ID: 201
print:         201        No Name        Nothing to do
ID: 800


        ‘printMinion‘ has no method matching printMinion(::Visitor)
    while loading In[191], in expression starting on line 16
```

```
In [192]: using MinionModule
          println(subtypes(MinionModule.Person))
          println(subtypes(MinionModule.Minion))
          print(super(MinionModule.Postdoc))
          methods(MinionModule.getID)
```

```
{Minion,Visitor}
{Postdoc,Student}
Minion
```

```
Out[192]: # 1 method for generic function "getID":
          getID{T<:Person}(p::T<:Person) at In[190]:13
```

0.5.6 Not the julia way

What you can **NOT** do in Julia is bind function to the type.

Well, yes, it's doable if you google it, but people sort of agree that this is not the julia way https://thenewphalls.wordpress.com/2014/02/19/understanding-object-oriented-programming-in-julia-part-1/ https://thenewphalls.wordpress.com/2014/03/06/understanding-object-oriented-programming-in-julia-inheritance-part-2/

```
type Programmer
    ...
    ## Many OO program will "link/associate" function with it's type
    function AssignProjcet (newProjcet)
        project = newProjcet
    end
    # NOT quite what julia is design for
end


prog1 = Programmer(10, "old project")
prog1.AssignProjcet("new project") ## Does NOT work here
```

```
In [193]: module Fail
          type Programmer
              id::Int64
              project::String
              Programmer(id, project) = new(id, project)

              ## Many OO program will "link/associate/bind/boundle" function with it's type/class
              function AssignProjcet (newProjcet)
                  project = newProjcet
              end
              # NOT quite what julia is design for
          end
          end


          using Fail
          prog1 = Fail.Programmer(10, "old project")
          println (prog1)

          prog1.AssignProjcet("new project") ## Error! you get `type Programmer has no field AssignProj
```

```
Warning: replacing module Fail
```

```
Programmer(10,"old project")
```

```
        type Programmer has no field AssignProjcet
    while loading In[193], in expression starting on line 20
```