Introduction to Julia Programming

Type system and multiple dispatch

by Yueh-Hua Tu

Outline

- Type
- Composite type
- Define methods
- Method ambiguity
- Abstract type
- Type hierarchy
- Incomplete initialization

Types

Everyone can use types in their work, such as `Int64`, `String`, `Float64`, `Bool`.

However, if you use these types directly, you will make your code messy and not easy to read. Thus, you will get difficult in debugging your code.

To address this, we need high-level encapsulation and design high-level types for ourselves.

Type assertion

In dynamic type languages, only value has type, not variable.

In [1]:
```
3::Int64
```
Out[1]:
```
3
```

The code above confirms that `3` is `Int64` type, so the code here is more like an assertion.（斷言）

In [2]:
```
x = 3::Int64
```
Out[2]:
```
3
```

How do you check the type of a variable?

In [3]:
```
typeof(x)
```

Out[3]:
```
Int64
```

Composite type

Composite type composes several variables in a type. Thus, we can encapsulate a concept into a type and implement that concept.

In [4]:

```
struct Rectangle
    height::Int64
    width::Int64
end
```

p.s. Naming of a type should follow the camel case.

p.s. Here type can be bound to variables.

Composite type

In [5]:
```
rec = Rectangle(4, 5)
```
Out[5]:
```
Rectangle(4, 5)
```
In [6]:
```
rec.height
```
Out[6]:
```
4
```
In [7]:
```
rec.width
```
Out[7]:
```
5
```
In [8]:
```
rec.width = 8
```
```
setfield! immutable struct of type Rectangle cannot be changed

Stacktrace:
 [1] setproperty!(::Rectangle, ::Symbol, ::Int64) at ./Base.jl:34
 [2] top-level scope at In[8]:1
```

Mutable and immutable type

In the default setting, `struct` provides immutable type definition for us. If you want mutable type, use `mutable struct` instead.

In [9]:
```
mutable struct MutableRectangle
   height::Int64
   width::Int64
end
```

In [10]:
```
mrec = MutableRectangle(4, 5)
```

Out[10]:
```
MutableRectangle(4, 5)
```

In [11]:
```
mrec.width = 8
```

Out[11]:
```
8
```

Define methods

In [12]:
```
f(x::Float64, y::Float64) = 2x + y
```
Out[12]:
```
f (generic function with 1 method)
```
In [13]:
```
f(2.0, 3.0)
```
Out[13]:
```
7.0
```

Isn't it looks like a function?

Difference between function and method

In [14]:
```
f(x::Number, y::Number) = 2x - y
f(2.0, 3)
```

Out[14]:
```
1.0
```

You will found that Julia dispatch f(Float64, Int64) to the method f(Number, Number) .

In [15]:
```
methods(f) # you can check the current methods implemented as this function name
```

Out[15]:

2 methods for generic function **f**:

- f(x::**Float64**, y::**Float64**) in Main at In[12]:1
- f(x::**Number**, y::**Number**) in Main at In[14]:1

Thus, in Julia, *function* means the **interface** f .

Method means the **implementation** f(x::Number, y::Number) = 2x - y .

Interface and implementation

Interface

fly

Implementation

```
fly(bird::Bird) = println("Bird flies.")

fly(airplane::Airplane) = println("Airplane flies.")
```

Programming language, just like natural language, in **different context**, one word has different meanings.

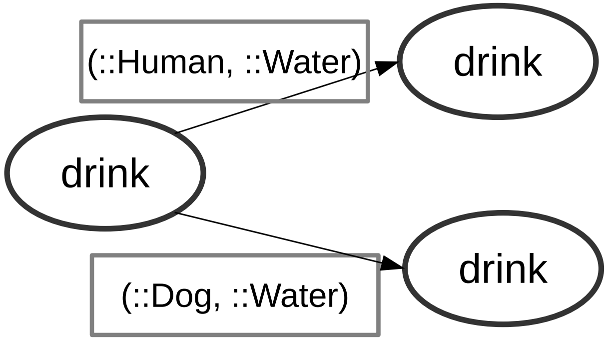
The most diverse thing is the behavior.

In different context, we have different implementation for different meanings.

How does Julia determine which method to call?

Multiple dispatch（多重分派）

In Julia, if there are many methods which has the same function name, which one to dispatch?



Julia will dispatch method accroding to the **number of parameters** and the **parameter types**.

Using all parameters in function call to determine which method to call is so called multiple dispatch.

Method ambiguity

In [16]:
```
g(x::Float64, y) = 2x + y
```

Out[16]:
```
g (generic function with 1 method)
```

In [17]:
```
g(x, y::Float64) = x + 2y
```

Out[17]:
```
g (generic function with 2 methods)
```

In [18]:
```
g(3.0, 4.0)
```

```
MethodError: g(::Float64, ::Float64) is ambiguous. Candidates:
  g(x::Float64, y) in Main at In[16]:1
  g(x, y::Float64) in Main at In[17]:1
Possible fix, define
  g(::Float64, ::Float64)

Stacktrace:
 [1] top-level scope at In[18]:1
```

這樣的定義會造成語意不清， g(Float64, Float64) 要執行哪一條呢

It is great to cover from general one to precise one

In [19]:
```
g(x::Float64, y::Float64) = 2x + 2y
g(x::Float64, y) = 2x + y
g(x, y::Float64) = x + 2y
```

Out[19]: g (generic function with 3 methods)

In [20]:
```
g(2.0, 3)
```

Out[20]: 7.0

In [21]:
```
g(2, 3.0)
```

Out[21]: 8.0

In [22]:
```
g(2.0, 3.0)
```

Out[22]: 10.0

Methods

Defining methods as usual.

In [23]:
```
area(x::Rectangle) = x.height * x.width
```

Out[23]:
```
area (generic function with 1 method)
```

Override the interface to customize the representation of an oject.

In [24]:
```
Base.show(io::IO, x::Rectangle) = println("Rectangle(h=$(x.height), w=$(x.width))")
```

In [25]:
```
rec
```

Out[25]:
```
Rectangle(h=4, w=5)
```

Abstract type

In [1]:

```
abstract type Shape end

struct Rectangle <: Shape
   height::Float64
   width::Float64
end

struct Triangle <: Shape
   base::Float64
   height::Float64
end
```

In [2]:

```
area(x::Rectangle) = x.height * x.width
area(x::Triangle) = 0.5 * x.base * x.height
```

Out[2]:

```
area (generic function with 2 methods)
```

In [3]:
```
rec = Rectangle(4, 5)
tri = Triangle(4, 5)
```
Out[3]:
```
Triangle(4.0, 5.0)
```
In [4]:
```
area(rec)
```
Out[4]:
```
20.0
```
In [5]:
```
area(tri)
```
Out[5]:
```
10.0
```

Type hierarchy

`Rectangle` and `Triangle` are so called concrete type（具體型別）, while `Shape` is abstract type（抽象型別）.

Concrete types can be instantiated（實體化） but abstract type cannot.

In [6]:

```
Shape()
```

```
MethodError: no constructors have been defined for Shape

Stacktrace:
 [1] top-level scope at In[6]:1
```

Type hierarchy

All types have super type（父型別）.

In [7]:
```
supertype(Rectangle)
```

Out[7]:
```
Shape
```

In [8]:
```
supertype(Shape)
```

Out[8]:
```
Any
```

Type hierarchy

Especially, **concrete type（具體型別） doesn't have subtype**. That is, you cannot make a type as the subtype of a concrete type, so all the concrete types are the leaves in type hierarchy.

In [9]:

```julia
struct Square <: Rectangle
end
```

```
invalid subtyping in definition of Square

Stacktrace:
 [1] top-level scope at /home/yuehhua/.julia/packages/IJulia/DrVMH/src/kernel.jl:52
```

Any type is the super type of all types

Once a new struct if defined, `Any` type will be the supertype of that type automatically.

In [10]:
```
Shape <: Any
```

Out[10]:
```
true
```

Union type

Union type is used in the context of allowing more than one types which don't belong to the same type hierarchy.

In [11]:
```
const IntOrFloat64 = Union{Int64, Float64}
```
Out[11]:
```
Union{Float64, Int64}
```
In [12]:
```
struct Foo
   x::IntOrFloat64
end
```
In [13]:
```
Foo(5)
```
Out[13]:
```
Foo(5)
```
In [14]:
```
Foo(5.)
```
Out[14]:
```
Foo(5.0)
```

Union type

In [15]:
```
foo(x::IntOrFloat64) = println(x)
```

Out[15]:
```
foo (generic function with 1 method)
```

In [16]:
```
foo(5)
```

```
5
```

In [17]:
```
foo(5.)
```

```
5.0
```

Union{} is the subtype of all types.

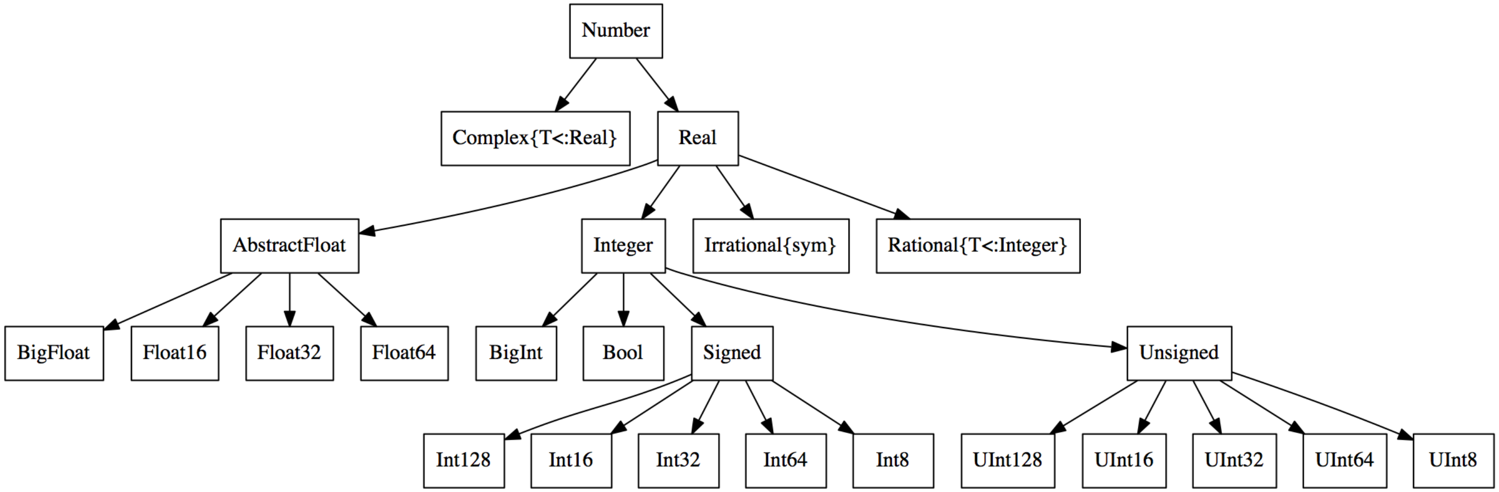
In [18]:
```
Union{} <: Shape
```

Out[18]:
```
true
```

In [19]:
```
Union{} <: Rectangle
```

Out[19]:
```
true
```

Declared type



Inner constructors

In [20]:
```
struct Square <: Shape
    rec::Rectangle

    Square(l) = new(Rectangle(l, l))
end
```

```
struct Square
    rec::Rectangle

    function Square()
        ...
    end
end
```

In [21]:
```
sq = Square(8)
```

Out[21]:
```
Square(Rectangle(8.0, 8.0))
```

In [22]:
```
sq.rec.height
```

Out[22]:
```
8.0
```

In [23]:
```
sq.rec.width
```

Out[23]:
```
8.0
```

Multiple inner constructor

In [24]:

```
struct Point <: Shape
    x::Float64
    y::Float64
end
```

In [25]:

```
p1 = Point(3.4, 2.9)
p2 = Point(5.0, 2.9)
```

Out[25]:

```
Point(5.0, 2.9)
```

Multiple inner constructor

In [26]:

```
struct Square2 <: Shape
    rec::Rectangle

    Square2(l) = new(Rectangle(l, l))

    function Square2(p1::Point, p2::Point)
        if p1.x == p2.x
            l = abs(p1.y - p2.y)
        elseif p1.y == p2.y
            l = abs(p1.x - p2.x)
        end
        new(Rectangle(l, l))
    end
end
```

In [27]:

```
sq = Square2(p1, p2)
```

Out[27]:

```
Square2(Rectangle(1.6, 1.6))
```

Outer Constructor

Literally, this constructor is defined outside of type definition, and it doesn't differ from other methods.

Thus, **you can simply add various constructors outside type definition**, as other language's constructor overloading do.

In [28]:
```
struct Circle <: Shape
    radius::Float64
end
```

In [29]:
```
Circle(r::Int64) = Circle(float(r))
```

Out[29]:
```
Circle
```

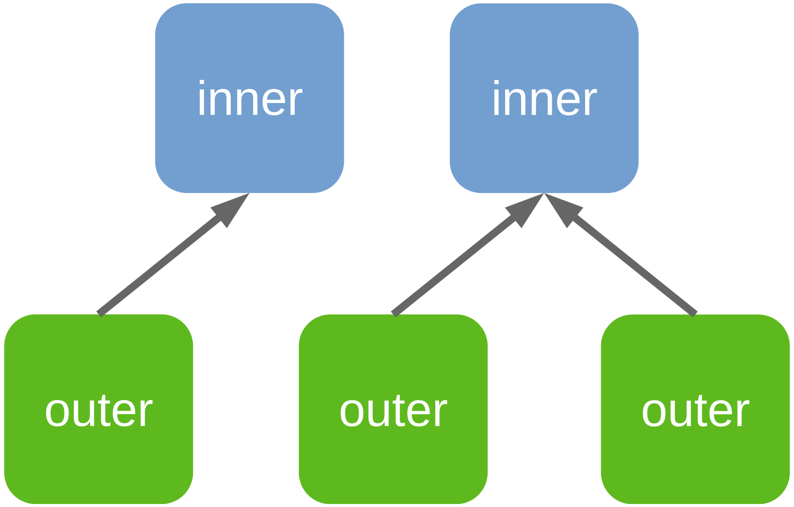
In [30]:
```
Circle(5.0)
```

Out[30]:
```
Circle(5.0)
```

In [31]:
```
Circle(5)
```

Out[31]:
```
Circle(5.0)
```

Design of constructors



We design the most strict constructor as inner constructor. Certain arguments/information are required to construct a type.

Outer constructors act as additional options for convenience. Outer constructors call inner constructors to construct a object.

Incomplete initialization

In [32]:

```
mutable struct SelfReferential
    obj::SelfReferential
end
```

In [33]:

```
sr = SelfReferential()
```

```
MethodError: no method matching SelfReferential()
Closest candidates are:
  SelfReferential(!Matched::SelfReferential) at In[32]:2
  SelfReferential(!Matched::Any) at In[32]:2

Stacktrace:
 [1] top-level scope at In[33]:1
```

In [34]:
```julia
mutable struct SelfReferential2
    obj::SelfReferential2
    SelfReferential2() = (x = new(); x.obj = x)
end
```

In [35]:
```julia
sr2 = SelfReferential2()
```

Out[35]:
```
SelfReferential2(SelfReferential2(#= circular reference @-1 =#))
```

In [36]:
```julia
sr2 === sr2.obj
```

Out[36]:
```
true
```

In [37]:
```julia
sr2 === sr2.obj.obj
```

Out[37]:
```
true
```

Special types

We introduce some special types which are associated to functions.

Most of them are directly a part of function constructs.

Nothing

When a function doesn't return anything, it returns a `nothing`.

`nothing` is the singleton of type `Nothing` which has only one object.

In [38]:
```
nothing == nothing
```

Out[38]:
```
true
```

In [39]:
```
typeof(nothing)
```

Out[39]:
```
Nothing
```

Use **missing** to represent a not existing value, not **nothing**

`missing` is also a singleton of type `Missing`.

In [40]:
```
missing
```

Out[40]:
```
missing
```

In [41]:
```
typeof(missing)
```

Out[41]:
```
Missing
```

missing can be calculated, but **nothing** cannot.

In [42]:
```
missing + 1
```
Out[42]:
```
missing
```

In [43]:
```
nothing + 1
```
```
MethodError: no method matching +(::Nothing, ::Int64)
Closest candidates are:
  +(::Any, ::Any, !Matched::Any, !Matched::Any...) at operators.jl:529
  +(!Matched::Complex{Bool}, ::Real) at complex.jl:301
  +(!Matched::Missing, ::Number) at missing.jl:115
  ...

Stacktrace:
 [1] top-level scope at In[43]:1
```

It follows our mathematical principles.

In [44]:
```
missing * Inf
```
Out[44]:
```
missing
```

Tuples

Tuple is a built-in type to wrap a bunch of things.

In [45]:
```
x = (1, 2., '3', "4")
```
Out[45]:
```
(1, 2.0, '3', "4")
```

In [46]:
```
x[1]
```
Out[46]:
```
1
```

In [47]:
```
x[2:3]
```
Out[47]:
```
(2.0, '3')
```

Tuple is immutable

In [48]:
```
objectid(x)
```
Out[48]:
```
0x77020e3c50be5e2b
```

In [49]:
```
objectid(x[2:3])
```
Out[49]:
```
0x1e56cfc0abb8aab3
```

Unpacking

In [50]:
```
a, b, c = x
```

Out[50]:
```
(1, 2.0, '3', "4")
```

In [51]:
```
a
```

Out[51]:
```
1
```

In [52]:
```
b
```

Out[52]:
```
2.0
```

In [53]:
```
c
```

Out[53]:
```
'3': ASCII/Unicode U+0033 (category Nd: Number, decimal digit)
```

Swap

In [54]:
```
b, a = a, b
```
Out[54]:
```
(1, 2.0)
```
In [55]:
```
a
```
Out[55]:
```
2.0
```
In [56]:
```
b
```
Out[56]:
```
1
```

Tuple is the data structure that pass arguments to function

Tuple is originally designed for argument passing in Julia. For example, function accepts arguments which are wrapped in a tuple. Function also return a tuple to carry a multiple return values.

In [57]:
```
h(x, y) = x + y
```
Out[57]:
```
h (generic function with 1 method)
```

In [58]:
```
h(1, 2)
```
Out[58]:
```
3
```

return keyword can be omitted.

In [59]:
```
function sumproduct(x, y, z)
    return (x + y) * z
end
```

Out[59]: sumproduct (generic function with 1 method)

In [60]:
```
function sumproduct(x, y, z)
    (x + y) * z
end
```

Out[60]: sumproduct (generic function with 1 method)

Multiple return values

If we want to return multiple values, we use tuple.

In [61]:
```
function shuffle_(x, y, z)
    (y, z, x)
end
```

Out[61]:
```
shuffle_ (generic function with 1 method)
```

Or if you want to write it in this way.

In [62]:
```
function shuffle_(x, y, z)
    y, z, x
end
```

Out[62]:
```
shuffle_ (generic function with 1 method)
```

In [63]:
```
shuffle_(1, 2., '3')
```
Out[63]:
```
(2.0, '3', 1)
```
In [64]:
```
x = shuffle_(1, 2., '3')
typeof(x)
```
Out[64]:
```
Tuple{Float64,Char,Int64}
```

(Argument) destruction

We can destruct many things, such as tuples, arrays, sets and dictionaries.

In [65]:

```
x = [1, 2, 3]
shuffle_(x...)
```

Out[65]:

```
(2, 3, 1)
```

is equivalent to `shuffle_(1, 2, 3)` .

Named tuple

Named tuple is a tuple equipped with name. Name is used to get access to specific value.

In [66]:
```
x = (a=1, b=2.)
```
Out[66]:
```
(a = 1, b = 2.0)
```

Similar to tuple, you can index a named tuple with property prefixed by `:`.

In [67]:
```
x[:a]
```
Out[67]:
```
1
```

Or you can use dot operator.

In [68]:
```
x.a
```
Out[68]:
```
1
```

Enum type

`Enum` type is the enumeration in other languages. Enumeration is used to represent several certain states. For example, the traffic light has three states: red light, yellow light and green light. `Enum` type is used to represent states which are not increasing.

```
@enum TrafficLight red=1 yellow=2 green=3
```

In [69]:
```
@enum TrafficLight::UInt8 begin
    red = 1
    yellow = 2
    green = 3
end
```

In [70]:
```
TrafficLight(1)
```

Out[70]:
```
red::TrafficLight = 0x01
```

Check Enum

What instances are in `TrafficLight`?

In [71]:
```
instances(TrafficLight)
```

Out[71]:
```
(red, yellow, green)
```

How many instances are in `TrafficLight`?

In [72]:
```
length(instances(TrafficLight))
```

Out[72]:
```
3
```

Convert to integers or strings

In [73]:
```
Integer(red)
```

Out[73]:
```
0x01
```

In [74]:
```
Int(red)
```

Out[74]:
```
1
```

In [75]:
```
string(yellow)
```

Out[75]:
```
"yellow"
```

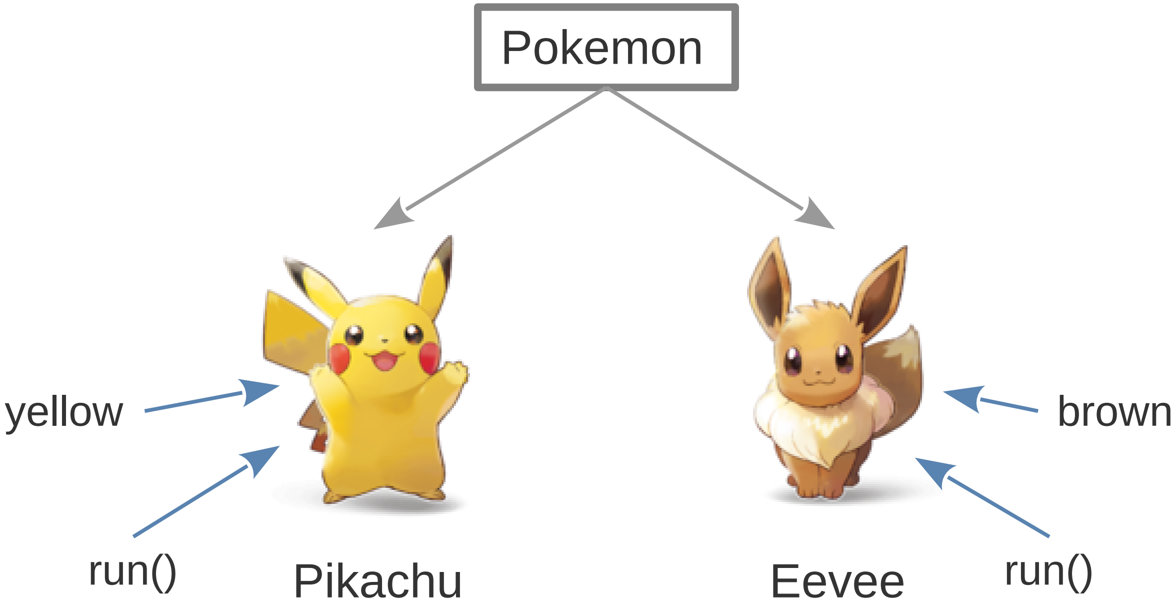
Defining methods on Enum

One can define methods on `TrafficLight`.

In [76]:
```
f(x::TrafficLight) = Int(x)
```
Out[76]:
```
f (generic function with 1 method)
```

In [77]:
```
f(green)
```
Out[77]:
```
3
```

Compare with traditional object-oriented programming



Compare with Python

Encapsulation

Julia doesn’t have access control

There is only *public* to the type property. Everyone can access those properties, once they get the object.

It is merely no encapsulation concept in Julia.

Julia don't hide any infromation from others.

Inheritance

Single inheritance

As analogy, julia's subtype is similar to the concept of inheritance in OOP, so Julia can only accept single inheritance.

```
struct Apple <: Fruit
    ...
end
```

Multiple inheritance

```
class Jet(Airplane,Weapon):
    ...
```

Polymorphism

The key difference between Julia and other OOP languages.

Single dispatch

```
class DNA:
  def transcribe(self):
    ...
```

Multiple dispatch

```
transcribe(dna::DNA) = ...
translate(rna::RNA, rib::Ribosome) = ...
```

Interface

Interface

```
public interface Repeatable{
   public void repeat();
}
```

Duck typing

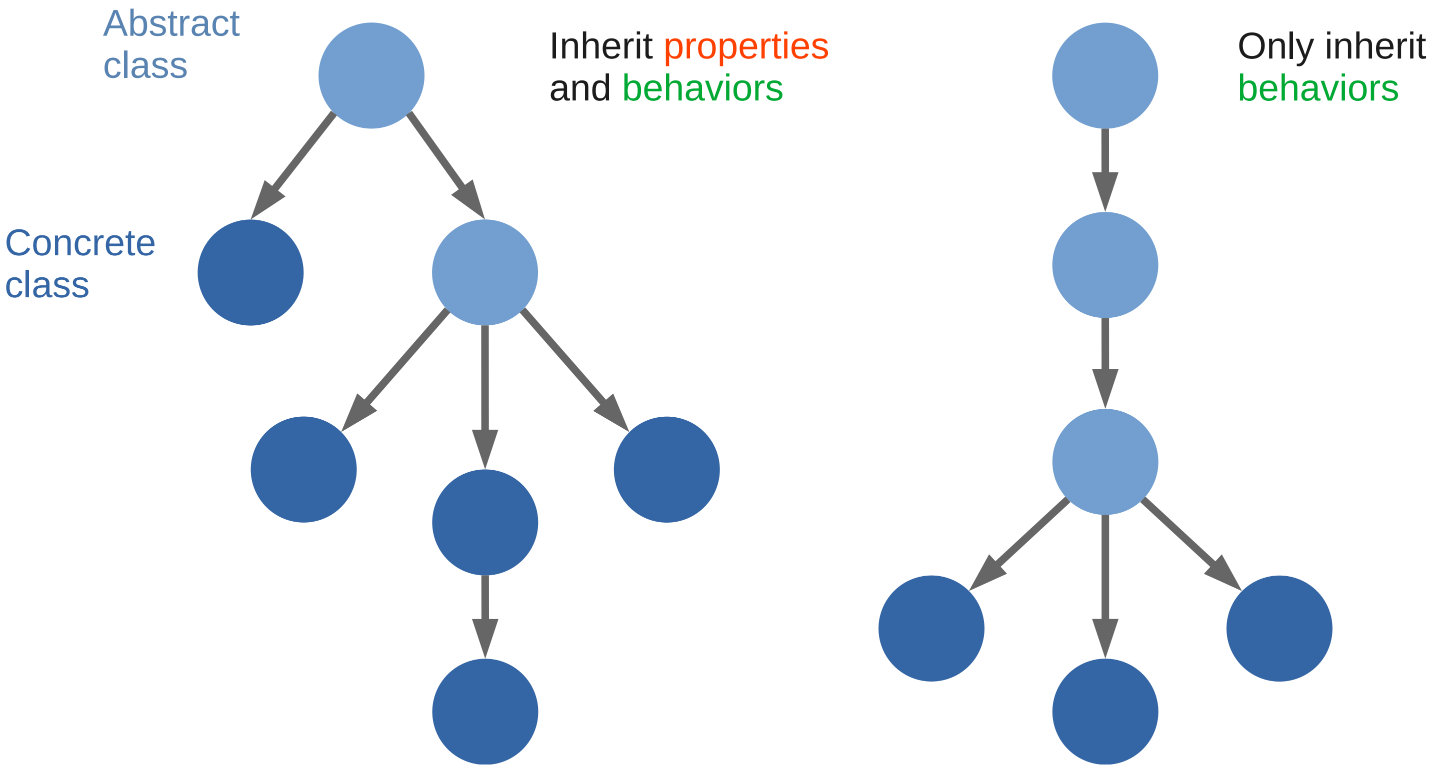
In [78]:
```
repeat([5,10], 2)
```

Out[78]:
```
4-element Array{Int64,1}:
  5
 10
  5
 10
```

In [79]:
```
repeat("10", 2)
```

Out[79]:
```
"1010"
```

Julia focus more on behavior



Q & A