Introduction to Julia Programming

Generic programming

by Yueh-Hua Tu

Outline

- Parametric type
- Parametric method
- Parametric Constructors
- Design patterns
- Empty generic function
- Examples

Generic programming

- Generic programming provide a way to write generic code, which doesn't depend on specific type.
- Generic programming is a kind of programming paradigm, just like object-oriented or functional programming.
- Generic programming let specific type plug-in the data structure or algorithm after declaration.
- Julia support generic programming in two ways: parametric types, parametric methods

Generic programming in other languages

- C++: template
- Java: List<String>
- Haskell: typeclass

Parametric types

Parametric types are like containers, which include distinct types of elements.

In [1]:
```
struct FreePoint{T,S}
    x::T
    y::S
end
```

In [2]:
```
p1 = FreePoint(2.5, 3)
```

Out[2]:
```
FreePoint{Float64,Int64}(2.5, 3)
```

In [3]:
```
typeof(p1)
```

Out[3]:
```
FreePoint{Float64,Int64}
```

Parametric types

Restrict types to be the same.

In [4]:

```
struct Point{T}
    x::T
    y::T
end
```

In [5]:

```
p1 = Point(3, 4)
```

Out[5]:

```
Point{Int64}(3, 4)
```

In [6]:

```
typeof(p1)
```

Out[6]:

```
Point{Int64}
```

Parametric types

Parametric types are distinct depending on the specific element types.

In [7]:
```
p2 = Point(3., 4.)
```
Out[7]:
```
Point{Float64}(3.0, 4.0)
```

In [8]:
```
typeof(p2)
```
Out[8]:
```
Point{Float64}
```

In [9]:
```
typeof(p1) == typeof(p2)
```
Out[9]:
```
false
```

Parametric types can help you add some constraints on types

In [10]:
```
Point(3, 4.)
```

```
MethodError: no method matching Point(::Int64, ::Float64)
Closest candidates are:
  Point(::T, !Matched::T) where T at In[4]:2

Stacktrace:
 [1] top-level scope at In[10]:1
```

In [11]:
```
Point("abc", "λ")
```

Out[11]:
```
Point{String}("abc", "λ")
```

Parametric types

Restrict types should be the subtypes of `Real`.

In [12]:
```
struct Point2{T<:Real}
    x::T
    y::T
end
```

In [13]:
```
Point(3, 4)
```

Out[13]:
```
Point{Int64}(3, 4)
```

In [14]:
```
Point2("3", "4")
```

```
MethodError: no method matching Point2(::String, ::String)

Stacktrace:
 [1] top-level scope at In[14]:1
```

Define methods on parametric types

In [15]:
```
function sum(p1::Point, p2::Point)
    Point(p1.x + p2.x, p1.y + p2.y)
end
```

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

In [16]:
```
p1
```

Out[16]:
```
Point{Int64}(3, 4)
```

In [17]:
```
p2
```

Out[17]:
```
Point{Float64}(3.0, 4.0)
```

In [18]:
```
sum(p1, p2)
```

Out[18]:
```
Point{Float64}(6.0, 8.0)
```

Example: Array

In [19]:

```julia
Matrix{Int64}(undef, 8, 8)
```

Out[19]:

```
8×8 Array{Int64,2}:
 139719740428400  139719740428912  …  139719740431472  139719740431984
 139719740428464  139719740428976     139719740431536  139719740432048
 139719740428528  139719740429040     139719740431600  139719740432112
 139719740428592  139719740429104     139719740431664  139719740432176
 139719818970736  139719740429168     139719740431728  139719740432240
 139719740428720  139719740429232  …  139719740431792  139719740432304
 139719740428784  139719740429296     139719740431856  139719740432368
 139719740428848  139719740429360     139719740431920  139719740432432
```

Trick to put integers on parametric type

To store integer as a property of type.

In [20]:

```
struct Point3{T}
    x
    y
end
```

In [21]:

```
Point3{2}(3, 4)
```

Out[21]:

```
Point3{2}(3, 4)
```

Parametric methods

Restrict argument types with parameters.

In [22]:

```
function foo(p1::Point2{T}, p2::Point2{T}) where {T}
  Point(p1.x + p2.x, p1.y + p2.y)
end
```

Out[22]:

```
foo (generic function with 1 method)
```

In [23]:

```
p1 = Point2(3, 4)
p2 = Point2(3., 4.)
foo(p1, p2)
```

```
MethodError: no method matching foo(::Point2{Int64}, ::Point2{Float64})
Closest candidates are:
 foo(::Point2{T}, !Matched::Point2{T}) where T at In[22]:2

Stacktrace:
 [1] top-level scope at In[23]:3
```

In [24]:

```
p3 = Point2(3., 4.)
foo(p3, p2)
```

Out[24]:

```
Point{Float64}(6.0, 8.0)
```

Parametric methods coupling with multiple dispatch

In [25]:

```
same_type(x::T, y::T) where {T} = true
same_type(x, y) = false
```

Out[25]:

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

In [26]:

```
same_type(1, 2)  # the same type
```

Out[26]:

```
true
```

In [27]:

```
same_type(1, 2.0)  # distinct type
```

Out[27]:

```
false
```

Example

In [28]:
```
concat(v::Vector{T}, x::T) where {T} = [v..., x]
```

Out[28]:
```
concat (generic function with 1 method)
```

In [29]:
```
concat([1, 2, 3], 4)
```

Out[29]:
```
4-element Array{Int64,1}:
 1
 2
 3
 4
```

In [30]:
```
concat([1, 2, 3], 4.0)
```

```
MethodError: no method matching concat(::Array{Int64,1}, ::Float64)
Closest candidates are:
  concat(::Array{T,1}, !Matched::T) where T at In[28]:1

Stacktrace:
 [1] top-level scope at In[30]:1
```

Constraint argument types in method

In [31]:
```
foobar(a, b, x::T) where {T <: Integer} = (a, b, x)
```

Out[31]:
```
foobar (generic function with 1 method)
```

In [32]:
```
foobar(1, 2, 3)
```

Out[32]:
```
(1, 2, 3)
```

In [33]:
```
foobar(1, 2.0, 3)
```

Out[33]:
```
(1, 2.0, 3)
```

In [34]:
```
foobar(1, 2.0, 3.0)
```

```
MethodError: no method matching foobar(::Int64, ::Float64, ::Float64)
Closest candidates are:
  foobar(::Any, ::Any, !Matched::T) where T<:Integer at In[31]:1

Stacktrace:
 [1] top-level scope at In[34]:1
```

Use parametric method to get types

Parametric method can help us extract types from parametric types.

In [35]:
```
Base.eltype(::Point{T}) where {T} = T
```

is equivalent to

```
Base.eltype(p::Point{T}) where {T} = T
```

In [36]:
```
p1 = Point(3., 4.)
eltype(p1)
```

Out[36]:
```
Float64
```

Utilize types

In Julia, types are themselves objects. You can do anything what an object can do for them.

In [37]:

```julia
function Base.zero(p::Point{T}) where {T}
    Point(T(0), T(0))
end
```

In [38]:

```julia
zero(p1)
```

Out[38]:

```
Point{Float64}(0.0, 0.0)
```

In [39]:

```julia
zero(Point(3, 4))
```

Out[39]:

```
Point{Int64}(0, 0)
```

Parametric Constructors

To instantiate a parametric type, a constructor is needed.

In [1]:
```
struct Point{T<:Real}
  x::T
  y::T
end
```

is equivalent to

```
type Point{T<:Real}
  x::T
  y::T

  Point{T}(x,y) where {T<:Real} = new(x,y)
end

Point(x::T, y::T) where {T<:Real} = Point{T}(x,y)
```

Implicit Parametric Type Constructors

Parametric type is provided in `Point(x::T, y::T)` implicitly.

In [2]:
```
Point(1,2)
```
Out[2]:
```
Point{Int64}(1, 2)
```
In [3]:
```
Point(1.0,2.5)
```
Out[3]:
```
Point{Float64}(1.0, 2.5)
```
In [4]:
```
Point(1,2.5)
```
```
MethodError: no method matching Point(::Int64, ::Float64)
Closest candidates are:
  Point(::T, !Matched::T) where T<:Real at In[1]:2

Stacktrace:
 [1] top-level scope at In[4]:1
```

Explicit Parametric Type Constructors

Parametric type is provided in `Point{T}(x,y)` explicity. Thus, you may want to cast values by specifing types.

In [5]:
```
Point{Int64}(1, 2)
```
Out[5]:
```
Point{Int64}(1, 2)
```
In [6]:
```
Point{Int64}(1.0, 2.5)
```
```
InexactError: Int64(2.5)

Stacktrace:
 [1] Int64 at ./float.jl:710 [inlined]
 [2] convert at ./number.jl:7 [inlined]
 [3] Point{Int64}(::Float64, ::Float64) at ./In[1]:2
 [4] top-level scope at In[6]:1
```
In [7]:
```
Point{Float64}(1.0, 2.5)
```
Out[7]:
```
Point{Float64}(1.0, 2.5)
```
In [8]:
```
Point{Float64}(1, 2)
```
Out[8]:
```
Point{Float64}(1.0, 2.0)
```

Design Patterns

Design patterns with parametric methods

In [9]:

```julia
abstract type Animal end
abstract type Dog <: Animal end
abstract type Cat <: Animal end

struct Labrador <: Dog
end

struct GoldenRetriever <: Dog
end
```

Subtype

In [10]:
```
isanimal(::T) where {T <: Animal} = true
isanimal(x) = false
```
Out[10]:
```
isanimal (generic function with 2 methods)
```
In [11]:
```
isanimal(Labrador())
```
Out[11]:
```
true
```
In [12]:
```
isanimal(GoldenRetriever())
```
Out[12]:
```
true
```
In [13]:
```
isanimal(0)
```
Out[13]:
```
false
```

Equivalence

In [14]:
```
concat(v::Vector{T}, x::T) where {T} = [v..., x]
```
Out[14]:
```
concat (generic function with 1 method)
```

In [15]:
```
concat([1,2,3], 4)
```
Out[15]:
```
4-element Array{Int64,1}:
 1
 2
 3
 4
```

In [16]:
```
concat([1,2,3], 4.)
```
```
MethodError: no method matching concat(::Array{Int64,1}, ::Float64)
Closest candidates are:
  concat(::Array{T,1}, !Matched::T) where T at In[14]:1

Stacktrace:
 [1] top-level scope at In[16]:1
```

Replace if-else by dispatching

Suppose you have some data to be flatten...

In [17]:
```
xs = [["a", "b", "c"], "d", "e", ["f", "g"]];
```

In [18]:
```
collections = []
for x in xs
    if x isa String
        push!(collections, x)
    elseif x isa Vector
        for i in x
            push!(collections, i)
        end
    end
end
```

In [19]:
```
collections
```

Out[19]:
```
7-element Array{Any,1}:
 "a"
 "b"
 "c"
 "d"
 "e"
 "f"
 "g"
```

Replace if-else by dispatching

In [20]:

```
flatten!(collections, x::String) = (push!(collections, x))
function flatten!(collections, x::Vector)
   for i in x
      push!(collections, i)
   end
end
```

Out[20]:

```
flatten! (generic function with 2 methods)
```

In [21]:

```
collections = []
for x in xs
   flatten!(collections, x)
end
```

In [22]:

```
collections
```

Out[22]:

```
7-element Array{Any,1}:
 "a"
 "b"
 "c"
 "d"
 "e"
 "f"
 "g"
```

Empty generic function

Sometimes, you want to define an **interface** for your method, **instead of implementation**. It is usually used in a framework to provide features which are compatible to user-defined methods.

In [23]:
```
function generic  # no arguments, as a placeholder
end
```

Out[23]:
```
generic (generic function with 0 methods)
```

In [24]:
```
methods(generic)
```

Out[24]: # 0 methods for generic function **generic**:

In [25]:

```
generic()
```

```
MethodError: no method matching generic()

Stacktrace:
 [1] top-level scope at In[25]:1
```

Example 1 - OOP in multiple dispatch

In [1]:

```
abstract type Animal end

struct Dog <: Animal
    color::String
    species::String
end

struct Cat <: Animal
    color::String
    species::String
end
```

In [2]:

```julia
function color(a::Animal)
  return a.color
end

function voice(d::Dog)
  return "bark"
end

function voice(c::Cat)
  return "meow"
end
```

Out[2]:

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

In [3]:
```
d1 = Dog("yellow", "Labrador")
```
Out[3]:
```
Dog("yellow", "Labrador")
```
In [4]:
```
voice(d1)
```
Out[4]:
```
"bark"
```
In [5]:
```
c1 = Cat("brown", "?")
```
Out[5]:
```
Cat("brown", "?")
```
In [6]:
```
voice(c1)
```
Out[6]:
```
"meow"
```

Example 2 - OOP with generic programming

Simple point of service system (POS)

In [7]:

```
abstract type Item end

struct OrderList{T<:Item}
    item_list::Vector{T}

    OrderList{T}() where {T<:Item} = new(Item[])
end
```

In [8]:

```
struct Apple <: Item
    price
    Apple() = new(100)
end

struct Banana <: Item
    price
    Banana() = new(50)
end
```

In [9]:

```
add!(ol::OrderList, it::Item) = push!(ol.item_list, it)

function Base.sum(ol::OrderList)
    s = 0
    for it in ol.item_list
        s += it.price
    end
    return s
end
```

In [10]:
```
l = OrderList{Item}()
```
Out[10]:
```
OrderList{Item}(Item[])
```
In [11]:
```
add!(l, Apple())
```
Out[11]:
```
1-element Array{Item,1}:
 Apple(100)
```
In [12]:
```
add!(l, Apple())
add!(l, Apple())
add!(l, Banana())
add!(l, Banana())
```
Out[12]:
```
5-element Array{Item,1}:
 Apple(100)
 Apple(100)
 Apple(100)
 Banana(50)
 Banana(50)
```
In [13]:
```
sum(l)
```
Out[13]:
```
400
```

In [14]:
```
l = OrderList{Apple}()
```

Out[14]:
```
OrderList{Apple}(Apple[])
```

In [15]:
```
add!(l, Apple())
add!(l, Apple())
```

Out[15]:
```
2-element Array{Apple,1}:
 Apple(100)
 Apple(100)
```

In [16]:
```
add!(l, Banana())
```

```
MethodError: Cannot `convert` an object of type Banana to an object of type Apple
Closest candidates are:
  convert(::Type{T}, !Matched::T) where T at essentials.jl:171

Stacktrace:
 [1] push!(::Array{Apple,1}, ::Banana) at ./array.jl:913
 [2] add!(::OrderList{Apple}, ::Banana) at ./In[9]:1
 [3] top-level scope at In[16]:1
```

In [17]:

```
sum(l)
```

Out[17]:

```
200
```

Write a generic algorithm

Correlation coefficient

$$\rho = \frac{\sum_{n}^{i=1}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{n}^{i=1}(X_i - \bar{X})^2}\sqrt{\sum_{n}^{i=1}(Y_i - \bar{Y})^2}}$$

Write a generic algorithm

In [18]:

```
function correlation(x, y)
    n = length(x)
    @assert length(y) == n "Not matched sample size"
    x̄ = sum(x) / n
    ȳ = sum(y) / n
    x̂ = x .- x̄
    ŷ = y .- ȳ
    ρ = sum(x̂ .* ŷ) / (sqrt(sum(x̂.^2))*sqrt(sum(ŷ.^2)))
    return ρ
end
```

Out[18]:

```
correlation (generic function with 1 method)
```

In [19]:

```
x = [2, 3, 4, 5, 6, 2, 3, 4, 5]
y = [32, 32, 5, 42, 6, 17, 19, 20, 24];
```

In [20]:

```
correlation(x, y)
```

Out[20]:

```
-0.20034374130204088
```

Q&A