# MultiScaleArrays_Test_2018-10-01

October 1, 2018

Following along with example from: https://github.com/JuliaDiffEq/MultiScaleArrays.jl

```
In [7]: using Pkg
        Pkg.add("MultiScaleArrays")
        Pkg.resolve()
        using MultiScaleArrays;
```

```
  Updating registry at `~/.julia/registries/General`
  Updating git-repo `https://github.com/JuliaRegistries/General.git`
 Resolving package versions...
  Updating `Project.toml`
 [no changes]
  Updating `Manifest.toml`
 [no changes]
 Resolving package versions...
  Updating `Project.toml`
 [no changes]
  Updating `Manifest.toml`
 [no changes]
```

Build a hierarchy where Embryos contain Tissues which contain Populations which contain Cells, and the cells contain proteins whose concentrations are modeled as simply a vector of numbers (it can be anything linearly indexable).

```
In [8]: struct Cell{B} <: AbstractMultiScaleArrayLeaf{B}
            values::Vector{B}
        end
        struct Population{T<:AbstractMultiScaleArray,B<:Number} <: AbstractMultiScaleArray{B}
            nodes::Vector{T}
            values::Vector{B}
            end_idxs::Vector{Int}
        end
        struct Tissue{T<:AbstractMultiScaleArray,B<:Number} <: AbstractMultiScaleArray{B}
            nodes::Vector{T}
            values::Vector{B}
            end_idxs::Vector{Int}
        end
```

```
struct Embryo{T<:AbstractMultiScaleArray,B<:Number} <: AbstractMultiScaleArrayHead{B}
    nodes::Vector{T}
    values::Vector{B}
    end_idxs::Vector{Int}
end
```

Using the constructors we can directly construct leaf types:

```
In [18]: cell1 = Cell([1.0; 2.0; 3.0])
         cell2 = Cell([4.0; 5.0])

Out[18]: Cell{Float64}([4.0, 5.0])
```

build types higher up in the hierarchy by using the constuct method

```
In [19]: population = construct(Population, deepcopy([cell1, cell2])) # Make a Population from
         cell3 = Cell([3.0; 2.0; 5.0])
         cell4 = Cell([4.0; 6.0])
         population2 = construct(Population, deepcopy([cell3, cell4]))
         tissue1 = construct(Tissue, deepcopy([population, population2])) # Make a Tissue from
         tissue2 = construct(Tissue, deepcopy([population2, population]))
         embryo = construct(Embryo, deepcopy([tissue1, tissue2])) # Make an embryo from Tissue

Out[19]: Embryo{Tissue{Population{Cell{Float64},Float64},Float64},Float64}(Tissue{Population{Ce
```

Note that tuples can be used as well. This allows for type-stable indexing with heterogeneous nodes. For example:

```
In [20]: tissue1 = construct(Tissue, deepcopy([population, cell3]))

Out[20]: Tissue{AbstractMultiScaleArray{Float64},Float64}(AbstractMultiScaleArray{Float64}[Popu
```

The head node then acts as the king. It is designed to have functionality which mimics a vector in order for usage in DifferentialEquations or Optim. This returns the "12th protein", counting by Embryo > Tissue > Population > Cell in order of the vectors.

```
In [24]: embryo[12]

Out[24]: 2.0

In [23]: embryo[:]

Out[23]: 20-element Array{Float64,1}:
          1.0
          2.0
          3.0
          4.0
          5.0
          3.0
          2.0
```

```
 5.0
 4.0
 6.0
 3.0
 2.0
 5.0
 4.0
 6.0
 1.0
 2.0
 3.0
 4.0
 5.0
```

```
In [22]: size(embryo)
```

```
Out[22]: (20,)
```

The linear indexing exists for every AbstractMultiScaleArray. These types act as full linear vectors, so standard operations do the sensical operations:

```
In [25]: embryo[10] = 4.0 # changes protein concentration 10
```

```
Out[25]: 4.0
```

```
In [30]: embryo[2,3,1] # Gives the 1st cell in the 3rd population of the second tissue
```

```
BoundsError: attempt to access 2-element Array{Population{Cell{Float64},Float64},1} at

Stacktrace:
 [1] getindex at ./array.jl:731 [inlined]
 [2] nodechild at /Users/mcfefa/.julia/packages/MultiScaleArrays/k0Sa9/src/indexing.jl
 [3] getindex at /Users/mcfefa/.julia/packages/MultiScaleArrays/k0Sa9/src/indexing.jl:3
 [4] nodeselect at /Users/mcfefa/.julia/packages/MultiScaleArrays/k0Sa9/src/indexing.jl
 [5] getindex(::Embryo{Tissue{Population{Cell{Float64},Float64},Float64},Float64}, ::In
 [6] top-level scope at none:0
```

```
In [31]: embryo[:] # generates a vector of all of the protein concentrations
```

```
Out[31]: 20-element Array{Float64,1}:
          1.0
          2.0
          3.0
          4.0
          5.0
          3.0
          2.0
          5.0
          4.0
          4.0
          3.0
          2.0
          5.0
          4.0
          6.0
          1.0
          2.0
          3.0
          4.0
          5.0
```

```
In [32]: eachindex(embryo) # generates an iterator for the indices
```

```
Out[32]: 1:20
```

Continuous models can thus be written at the protein level and will work seamlessly with DifferentialEquations or Optim which will treat it like a vector of protein concentrations. Using the iterators, note that we can get each cell population by looping through 2 levels below the top, so

```
In [33]: for cell in level_iter(embryo,3)
           # Do something with the cells!
         end
```

To apply a function cell-by-cell, you can write a dispatch f on the type for the level. Assuming we have d_embryo as similar to embryo, using level_iter_idx we can have its changes update some other head node d_embryo via:

```
In [34]: for (cell, y, z) in LevelIterIdx(embryo, 3)
             f(t, cell, @view d_embryo[y:z])
         end
```

```
        UndefVarError: d_embryo not defined
```

```
        Stacktrace:
```

```
[1] top-level scope at ./In[34]:2 [inlined]

[2] top-level scope at ./none:0
```

Since embryo will be the "vector" for the differential equation or optimization problem, it will be the value passed to the event handling. MultiScaleArrays includes behavior for changing the structure. For example:

```
In [35]: tissue3 = construct(Tissue, deepcopy([population, population2]))
         add_node!(embryo, tissue3) # Adds a new tissue to the embryo
         remove_node!(embryo, 2, 1) # Removes population 1 from tissue 2 of the embryo

In [37]: tissue3[:]

Out[37]: 10-element Array{Float64,1}:
          1.0
          2.0
          3.0
          4.0
          5.0
          3.0
          2.0
          5.0
          4.0
          6.0

In [38]: embryo[:]

Out[38]: 25-element Array{Float64,1}:
          1.0
          2.0
          3.0
          4.0
          5.0
          3.0
          2.0
          5.0
          4.0
          4.0
          1.0
          2.0
          3.0
          4.0
          5.0
          1.0
          2.0
          3.0
```

```
4.0
5.0
3.0
2.0
5.0
4.0
6.0
```