JuliaLang: The Ingredients for a Composable Programming Language

Dr Lyndon White

Research Software Engineer
Invenia Labs

What do I mean by composable ?

Examples:

- If you want to add tracking of measurement error to a scalar number, you shouldn't have to say anything about how your new type interacts with arrays **(Measurements.jl)**
- If you have a Differential Equation solver, and a Neural Network library, then you should just be able to have neural ODEs (**DiffEq.jl / Flux.jl**)
- If you have a package to add names to the dimensions of an array, and one to put arays on the GPU, then you shouldn't have to write code to have named arrays on the GPU (**NamedDims.jl / GPUArrays.jl**)

Why Julia is it this way?

I am going to tell you some things that may sound counter-intuitive.

I am going suggest, that julia code is so reusable, are because the language has not just good features, but **weak and missing features.**

Missing features like:

- Weak conventions about namespace polution
- Never got round to making it easy to use local modules, outside of packages
- A type system that can't be used to check correctness

But that these are countered by, or allow for other features:

- Strong convention about talking to other people
- Very easy to create packages
- Duck-typing, and multiple dispatch, together.

Julia namespacing is used in a leaky way

Common advise when loading code form another module in most languagage communities is: only import what you need.
e.g `using Foo: a, b c`

Common practice in Julia is to do: `using Foo`,
which imports everything everything that the author of `Foo` marked to be **exported**.

You don't have to, but it's common.

But what happens if one has package:

- `Bar` exporting `predict(::BarModel, data)`,
- and another `Foo` exporting `predict(::FooModel, data)`

and one does:

```
using Foo
using Bar
training_data, test_data = ...
mbar = BarModel(training_data)
mfoo = FooModel(training_data)
evaluate(predict(mbar), test_data)
evaluate(predict(mfoo), test_data)
```

If you have multiple `using`s trying to bring the same name into scope, then julia throws an error. Since it can't work out which to use.

As a user you can tell it what to use.

```
evaluate(Bar.predict(mbar), test_data)
evaluate(Foo.predict(mfoo), test_data)
```

But the package authors can solve this:

There is no name collision if both names *are* overloaded the from the same namespace.

If both `Foo` and `Bar` are overloading `StatsBase.predict` everything works.

```
using StatsBase  # exports predict
using Foo  # overloads `StatsBase.predict(::FooModel)
using Bar  # overloads `StatsBase.predict(::BarModel)
training_data, test_data = ...
mbar = BarModel(training_data)
mfoo = FooModel(training_data)
evaluate(predict(mbar), test_data)
evaluate(predict(mfoo), test_data)
```

This encourages people to work together

Name collisions makes package authors to come together and create base package (like `StatsBase`) and agree on what the functions mean.

They don't have to, since the user can still solve it, but it encourages it. Thus you get package authors thinking about other packages that might be used with theirs.

One can even overload functions from multiple namespaces if you want;
e.g. all of `MLJBase.predict`, `StatsBase.predict`, `SkLearn.predict`.
Which might all have slightly different interfaces targetting different use cases.

Its easier to create a package than a local module.

Many languages have one module per file, and you can load that module e.g. via
`import Filename`
from your current directory.

You can make this work in Julia also, but it is surprisingly fiddly.

What is easy however, is to create and use a package.

What does making a local module generally give you?

- Namespacing
- The feeling you are doing good software engineering
- Easier to transition later to a package

What does making a Julia package give you?

- All the above plus
- Standard directory structure, `src`, `test` etc
- Managed dependencies, both what they are, and what versions
- Easy re-distributivity -- harder to have local state
- Test-able using package manager's `pkg> test MyPackage`

The recommended way to create packages also ensures:

- Continuous Integration(/s) Setup
- Code coverage
- Documentation setup
- License set

Testing Julia code is important.

JIT compiler: even compilation errors don't arive til run-time.

Dynamic language: type system says nothing about correctness.

Testing julia code is important.

So its good to have CI etc all setup

Multiple Dispatch + Duck-typing

Assume it walks like a duck and talks like a duck, and if it doesn't fix that.

Another closely related factor is **Open Classes.**
But I'm not going to talk about that today, its uninteresting.
You need to allow new methods to be added to existing classes, in the first place.

Consider on might have a type from the **Ducks** library.

In [3]:
```
struct Duck end

walk(self) = println("🚶 Waddle")
talk(self) = println("🦆 Quack")

raise_young(self, child) = println("🐤 ➡️ 💧 Lead to water")
```

Out[3]: raise_young (generic function with 1 method)

and I have some code I want to run, that I wrote:

In [4]:
```
function simulate_farm(adult_animals, baby_animals)
    for animal in adult_animals
        walk(animal)
        talk(animal)
    end

    parent = first(adult_animals)
    for child in baby_animals
        raise_young(parent, child)
    end
end
```

Out[4]: simulate_farm (generic function with 1 method)

In [5]:
```
simulate_farm([Duck(), Duck(), Duck()], [Duck(), Duck()])
```

```
🚶 Waddle
🦆 Quack
🚶 Waddle
🦆 Quack
🚶 Waddle
🦆 Quack
🐤 ➡️ 💧 Lead to water
🐤 ➡️ 💧 Lead to water
```

Ok now I want to extend it with my own type. A Swan

In [7]:
```
struct Swan end
```

In [8]:
```
# Lets test with just 1 first:
simulate_farm([Swan()], [])
```

```
🚶 Waddle
🦆 Quack
```

The **Waddle** was right, but Swans don't **Quack**.

We did some duck-typing -- Swans walk like ducks, but they don't talk like ducks.

We can solve that with **single dispatch**.

In [12]:
```
talk(self::Swan) = println("🦢 Hiss")
```

Out[12]:
```
talk (generic function with 2 methods)
```

In [13]:
```
# Lets test with just 1 first:
simulate_farm([Swan()], [])
```

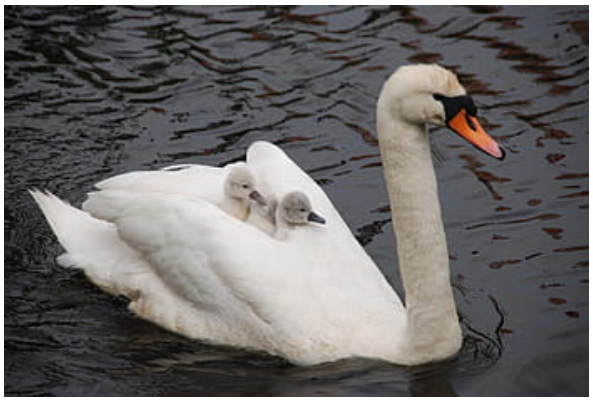
```
🚶 Waddle
🦢 Hiss
```

In [14]:
```
# Now the whole farm
simulate_farm([Swan(), Swan(), Swan()], [Swan(), Swan()])
```

```
🚶 Waddle
🦢 Hiss
🚶 Waddle
🦢 Hiss
🚶 Waddle
🦢 Hiss
🐤➡️💧 Lead to water
🐤➡️💧 Lead to water
```

That's not right. Swans do not lead their young to water.

They carry them



In [16]:
```
# Same thing again:
raise_young(self::Swan, child::Swan) = println("🐤 ↗️ 🦢 Carry on back")
```

Out[16]:
```
raise_young (generic function with 2 methods)
```

In [17]:
```
# Now the whole farm
simulate_farm([Swan(), Swan(), Swan()], [Swan(), Swan()])
```

```
🚶 Waddle
🦢 Hiss
🚶 Waddle
🦢 Hiss
🚶 Waddle
🦢 Hiss
🐤 ↗️ 🦢 Carry on back
🐤 ↗️ 🦢 Carry on back
```

Now I want a Farm with mixed poultry.

- 2 Ducks, a Swan, and 2 baby swans

In [20]:
```
simulate_farm([Duck(), Duck(), Swan()], [Swan(), Swan()])
```

```
🚶 Waddle
🦆 Quack
🚶 Waddle
🦆 Quack
🚶 Waddle
🦢 Hiss
🐤 ➡️ 💧 Lead to water
🐤 ➡️ 💧 Lead to water
```

Thats not right again.

```
🐤 ➡️ 💧 Lead to water
```

What happened?

We had a Duck, raising a baby Swan, and it lead it to water.

Ducks given baby Swans to raise, will just abandon them.

But how will we code this?

Option 1: Rewrite the Duck

```
function raise_young(self::Duck, child::Any)
    if child isa Swan
        println("🐤😢 Abandon")
    else
        println("🐤 ➡️ 💧 Lead to water")
    end
end
```

Rewriting the Duck has problems

- Have to edit someone elses library, to add support for *my* type.
- This could mean adding a lot of code for them to maintain
- Does not scale, what if other people wanted to add Chickens, Geese etc.

Varient: Monkey-patch

- If the language supports monkey patching, could do it that way
- but it means copying their code into my library, will run in to issues like not being able to update.
- Scaled to other people adding new types even worse, since no longer a central canonical source to copy

Varient: could fork their code

- That is giving up on code reuse.

Option 2: Inherit from the Duck

(NB: this example is not valid julia code)

```
struct DuckWithSwanSupport <: Duck end

function raise_young(self::DuckWithSwanSupport, child::Any)
    if child isa Swan
        println("🐤😢 Abandon")
    else
        raise_young(upcast(Duck, self), child)
    end
end
```

Inheriting from the Duck has problems:

- Have to replace every `Duck` in my code-base with `DuckWithSwanSupport`
- If I am using other libraries that might return a `Duck` I have to deal with that also

Still does not scale.

If someone else implements `DuckWithChickenSupport`, and I want to use both there code and mine, what do?

- Inherit from both? `DuckWithChickenAndSwanSupport`
- This is the classic multiple inheritance Diamond problem.
- It's hard. Even in languages supporting multiple inheritance, they may not support it in a useful way for this without me writing special cases for many things.

Option 3: Multiple Dispatch

This is clean and easy:

In [26]:
```
raise_young(parent::Duck, child::Swan) = println("🐤😢 Abandon")
```

Out[26]:
```
raise_young (generic function with 3 methods)
```

In [27]:
```
simulate_farm([Duck(), Duck(), Swan()], [Swan(), Swan()])
```

```
🚶 Waddle
🦆 Quack
🚶 Waddle
🦆 Quack
🚶 Waddle
🦢 Hiss
🐤😢 Abandon
🐤😢 Abandon
```

We will now take a short break

But does this happen in the wild?

Turns out it does.

The need to extend operations to act on new combinations of types shows up all the time in scientific computing.
I suspect it shows up more generally, but we've learned to ignore it.

If you look at a list of BLAS, methods you will see just this, encoded in the function name E.g.

- `SGEMM` - matrix matrix multiply
- `SSYMM` - symmetric-matrix matrix multiply
- ...
- `ZHBMV` - complex hermitian-banded-matrix vector multiply

And turns out people keep wanting to make more and more matrix types.

In [75]:
```
b = BlockArray(0.5.<sprand(3*4,3*4,0.9), [4,4,4],[4,4,4])
```

Out[75]:
```
3×3-blocked 12×12 BlockArray{Bool,2,Array{SparseMatrixCSC{Boo
l,Int64},2},Tuple{BlockedUnitRange{Array{Int64,1}},BlockedUni
tRange{Array{Int64,1}}}}:
 1  0  0  0  │  0  1  0  0  │  1  0  0  0
 0  1  1  1  │  1  1  1  1  │  1  1  0  0
 0  0  1  0  │  1  0  0  0  │  1  0  0  1
 0  0  0  0  │  0  0  1  0  │  1  0  0  0
 ────────────┼──────────────┼─────────────
 1  0  0  0  │  1  1  0  0  │  1  1  0  1
 0  1  1  1  │  1  1  1  0  │  0  1  0  0
 0  1  0  1  │  1  0  0  0  │  0  1  1  0
 1  0  0  0  │  0  0  1  1  │  1  1  0  0
 ────────────┼──────────────┼─────────────
 0  0  1  0  │  1  1  0  0  │  0  0  1  0
 1  0  0  0  │  0  0  0  0  │  0  1  1  0
 0  1  0  0  │  1  0  0  0  │  1  0  1  1
 1  1  0  1  │  0  1  1  1  │  0  1  1  1
```

In [65]:
```
# creates a banded matrix of 8, with 1 sub-diagonals and u super-
diagonals
BandedMatrix(Ones{Int}(10,10), (1,u))
```

Out[65]:
```
10×10 BandedMatrix{Int64,Array{Int64,2},Base.OneTo{Int64}}:
 1  1  ·  ·  ·  ·  ·  ·  ·  ·
 1  1  1  ·  ·  ·  ·  ·  ·  ·
 1  1  1  1  ·  ·  ·  ·  ·  ·
 ·  1  1  1  1  ·  ·  ·  ·  ·
 ·  ·  1  1  1  1  ·  ·  ·  ·
 ·  ·  ·  1  1  1  1  ·  ·  ·
 ·  ·  ·  ·  1  1  1  1  ·  ·
 ·  ·  ·  ·  ·  1  1  1  1  ·
 ·  ·  ·  ·  ·  ·  1  1  1  1
 ·  ·  ·  ·  ·  ·  ·  1  1  1
```

In [59]:
```
# creates a block-banded matrix with ones in the non-zero entries
x = BlockBandedMatrix(Ones{Int}(sum(rows),sum(cols)), rows,cols,
(l,u))
```

Out[59]:
```
4×4-blocked 10×10 BlockSkylineMatrix{Int64,Array{Int64,1},Blo
ckBandedMatrices.BlockSkylineSizes{Tuple{BlockedUnitRange{Arr
ay{Int64,1}},BlockedUnitRange{Array{Int64,1}}},Fill{Int64,1,T
uple{Base.OneTo{Int64}}},Fill{Int64,1,Tuple{Base.OneTo{Int6
4}}},BandedMatrix{Int64,Array{Int64,2},Base.OneTo{Int64}},Arr
ay{Int64,1}}}:
 1  │  1  1  │  ·  ·  ·  │  ·  ·  ·  ·
────┼────────┼───────────┼────────────
 1  │  1  1  │  1  1  1  │  ·  ·  ·  ·
 1  │  1  1  │  1  1  1  │  ·  ·  ·  ·
────┼────────┼───────────┼────────────
 1  │  1  1  │  1  1  1  │  1  1  1  1
 1  │  1  1  │  1  1  1  │  1  1  1  1
 1  │  1  1  │  1  1  1  │  1  1  1  1
────┼────────┼───────────┼────────────
 ·  │  1  1  │  1  1  1  │  1  1  1  1
 ·  │  1  1  │  1  1  1  │  1  1  1  1
 ·  │  1  1  │  1  1  1  │  1  1  1  1
 ·  │  1  1  │  1  1  1  │  1  1  1  1
```

In [60]:
```
# creates a banded-block-banded matrix with 8 in the non-zero ent
ries
y = BandedBlockBandedMatrix(Ones{Int}(sum(rows),sum(cols)), rows,
cols, (l,u), (λ,μ))
```

Out[60]:
```
4×4-blocked 10×10 BandedBlockBandedMatrix{Int64,PseudoBlockAr
ray{Int64,2,Array{Int64,2},Tuple{BlockedUnitRange{Array{Int6
4,1}},BlockedUnitRange{Array{Int64,1}}}},BlockedUnitRange{Arr
ay{Int64,1}}}:
 1  │  1  1  │  ·  ·  ·  │  ·  ·  ·  ·
────┼────────┼───────────┼────────────
 1  │  1  1  │  1  1  1  │  ·  ·  ·  ·
 1  │  1  1  │  1  1  1  │  ·  ·  ·  ·
────┼────────┼───────────┼────────────
 1  │  1  1  │  1  1  1  │  1  1  1  ·
 1  │  1  1  │  1  1  1  │  1  1  1  1
 ·  │  ·  1  │  ·  1  1  │  ·  1  1  1
────┼────────┼───────────┼────────────
 ·  │  1  1  │  1  1  1  │  1  1  1  ·
 ·  │  1  1  │  1  1  1  │  1  1  1  1
 ·  │  ·  1  │  ·  1  1  │  ·  1  1  1
 ·  │  ·  ·  │  ·  ·  1  │  ·  ·  1  1
```

And that is before other things you might like to to to a Matrix, which you'd like to encode in its type:

- Running on a GPU
- Tracking Operations for AutoDiff
- Naming dimensions, for easy lookup
- Distributing over a cluster

These are all important and show up in crucial applications.
When you start applying things across disciplines, they show up even more.
Like advancements in Neural Differential Equations, which needs:

- all the types machine learning research has invented,
- and all the types differential equation solving research has invented,

and wants to use them together.

So its not a reasonable thing for a numerical language to say that they've enumerated all the matrix types you might ever need.

Inserting a human into the JIT

Basic functionality of an Tracing JIT:

- Detect important cases via tracing
- Compile specialized methods for them

This is called specialization.

Basic functionalitionality of Julia's JIT:

- Specialize all methods on all types that they are called on as they are called

This is pretty good: its a reasonable assumption that the types are going to an important case.

What does multiple dispatch add ontop of Julia's JIT?

It lets a human tell it how that specialization should be done.
Which can add a lot of information.

Consider Matrix multiplication.

We have

- `*(::Dense, ::Dense)`:
  - multiply rows by columns and sum.
  - Takes $O(n^3)$ time

- `*(::Dense, ::Diagonal)` or `*(::Diagonal, ::Dense)`:
  - column-wise/row-wise scaling.
  - $O(n^2)$ time.

- `*(::OneHot, ::Dense)` or `*(::Dense, ::OneHot)`:
  - column-wise/row-wise slicing.
  - $O(n)$ time.

- `*(::Identity, ::Dense)` or `*(::Dense, ::Identity)`:
  - no change.
  - $O(1)$ time.

Anyone can have basic fast array processing by throwing the problem to BLAS, or a GPU.

But not everyone has Array types that are parametric on their scalar types; and the ability to be equally fast in both.

Without this, your array code, and your scalar code can not be disentangled.

BLAS for example does not have this.
It has a unique code for each combination of scalar and matrix type.

With this seperation, one can add new scalar types:

- Dual numbers
- Measument Error tracking numbers
- Symbolic Algebra numbers

Without ever having to touch array code, except as a late-stage optimization.

Otherwise, one needs to implement array support into one's scalars, to have reasonable performance at all.

People need to invent new languages. Its a good time to be inventing new languages. It's good for the world.

I'ld just really like those new languages to please have:

- multiple dispatch
- open classes, so you can add methods to things.
- array types that are parametric on their scalar types, at the type level
- A package management solution built-in, that everyone uses.