



# Type-Dispatch Design: Post Object-Oriented Programming for Julia

CHRISTOPHER RACKAUCKAS

READ REVIEWS

WRITE A REVIEW

## CORRESPONDENCE:

me@chrisrackauckas.com

## DATE RECEIVED:

August 18, 2018

## DOI:

10.15200/winn.153459.98988

## ARCHIVED:

August 18, 2018

## CITATION:

Christopher Rackauckas, Type-Dispatch Design: Post Object-Oriented Programming for Julia, *The Winnower* 5:e153459.98988, 2018, DOI: 10.15200/winn.153459.98988

© Rackauckas This article is distributed under the terms of the [Creative Commons Attribution 4.0 International License](#), which permits unrestricted use, distribution, and redistribution in any medium, provided that the original author and source are credited.



In this post I am going to try to explain in detail the type-dispatch design which is used in Julian software architectures. It's modeled after the design of many different packages and Julia Base, and has been discussed in parts elsewhere. This is actually just a blog post translation from my "[A Deep Introduction to Julia for Data Science and Scientific Computing](#)" workshop notes. I think it's an important enough topic to share more broadly.

The tl;dr: Julia is built around types. Software architectures in Julia are built around good use of the type system. This makes it easy to build generic code which works over a large range of types and gets good performance. The result is high-performance code that has many features. In fact, with generic typing, your code may have more features than you know of! The purpose of this tutorial is to introduce the multiple dispatch designs that allow this to happen.

Now let's discuss the main components of this design!

## DUCK TYPING

If it quacks like a duck, it might as well be a duck. This is the idea of defining an object by the way that it acts. This idea is central to type-based designs: **abstract types are defined by how they act**. For example, a `Number` is some type that can do things like `+`, `-`, `*`, and `/`. In this category we have things like `Float64` and `Int32`. An `AbstractFloat` is some floating point number, and so it should have a dispatch of `eps(T)` that gives its machine epsilon. An `AbstractArray` is a type that can be indexed like `A[i]`. An `AbstractArray` may be mutable, meaning it can be `"set": A[i]=v`.

These abstract types then have actions which abstract from their underlying implementation. `A.*B` does element-wise multiplication, and in many cases it does not matter what kind of array this is done on. The default is `Array` which is a contiguous array on the CPU, but this action is common amongst `AbstractArray` types. If a user has a `DistributedArray` (`DArray`), then `A.*B` will work on multiple nodes of a cluster. If the user uses a `GPUArray`, then `A.*B` will be performed on the GPU. Thus, if you don't restrict the usage of your algorithm to `Array`, then your algorithm actually "just works" as many different algorithms.

This is all well and good, but this would not be worthwhile if it were not performant. Thankfully, Julia has an answer to this. Every function auto-specializes on the types which it is given. Thus if you look at something like:

```
my_square(x) = x^2
```

then we see that this function will be efficient for the types that we give it. Looking at the generated code:

```
@code_llvm my_square(1)
define i64 @julia_my_square_72669(i64) #0 {
top:
    %1 = mul i64 %0, %0
    ret i64 %1
}
@code_llvm my_square(1.0)
define double @julia_my_square_72684(double) #0 {
top:
    %1 = fmul double %0, %0
    ret double %1
}
```

See that the function which is generated by the compiler is different in each case. The first specifically

is an integer multiplication  $x \times x$  of the input  $x$ . The other is a floating point multiplication  $x \times x$  of the input  $x$ . But this means that it does not matter what kind of Number we put in here: this function will work as long as  $*$  is defined, and it will be efficient by Julia's multiple dispatch design.

Thus we don't need to restrict the types we allow in functions in order to get performance. That means that

```
my_restricted_square(x::Int) = x^2
```

is no more efficient than the version above, and actually generates the same exact compiled code:

```
@code_llvm my_restricted_square(1)
```

```
define i64 @julia_my_restricted_square_72686(i64) #0 {
top:
  %1 = mul i64 %0, %0
  ret i64 %1
}
```

Thus we can write generic and efficient code by leaving our functions unrestricted. This is the practice of duck-typing functions. We just let them work on any input types. If the type has the correct actions, the function will "just work". If it does not have the correct actions, for our example above say  $*$  is undefined, then a `MethodError` saying the action is not defined will be thrown.

We can be slightly more conservative by restricting to abstract types. For example:

```
my_number_restricted_square(x::Number) = x^2
```

will allow any Number. There are things which can square which aren't Numbers for which this will now throw an error (a matrix is a simple example). But, this can let us clearly define the interface for our package/script/code. Using these assertions, we can then dispatch differently for different type classes. For example:

```
my_number_restricted_square(x::AbstractArray) = (println(x); x.^2)
```

Now, `my_number_restricted_square` calculates  $x^2$  on a Number, and for an array it will print the array and calculate  $x^2$  element-wise. Thus we are controlling behavior with broad strokes using classes of types and their associated actions.

## TYPE HIERARCHIES

This idea of control leads to type hierarchies. In object-oriented programming languages, you sort objects by their implementation. Fields, the pieces of data that an object holds, are what is inherited.

There is an inherent limitation to that kind of thinking when looking to achieve good performance. In many cases, you don't need as much data to do an action. A good example of this is the range type, for example `1:10`.

```
a = 1:10
```

This type is an abstract array:

```
typeof(a)
```

It has actions like an Array

```
fieldnames(a)
```

```
# Output
```

```
2-element Array{Symbol,1}:
```

```
:start
```

```
:stop
```

It is an immutable type which just holds the start and stop values. This means that its indexing, `A[i]`, is just a function. What's nice about this

```
@time collect(1:10000000)
```

```
0.038615 seconds (308 allocations: 76.312 MB, 45.16% gc time)
```

But creating an immutable type of two numbers is essentially free, no matter what those two numbers are:

```
@time 1:10000000
```

```
0.000001 seconds (5 allocations: 192 bytes)
```

The array takes  $\mathcal{O}(n)$  memory to store its values while this type is  $\mathcal{O}(1)$ , using a constant 192 bytes (if the start and stop are `Int64`). Yet, it

Another nice example is the UniformScaling operator, which acts like an identity matrix without forming an identity matrix.

```
println(I[10,10]) # prints 1
println(I[10,2]) # prints 0
```

This can calculate expressions like  $A \cdot b \cdot I$  without ever forming the matrix  $(eye(n))$  which would take  $\mathcal{O}(n^2)$  memory.

This means that a lot of efficiency can be gained by generalizing our algorithms to allow for generic typing and organization around actions.

This is the key idea to keep in mind when building type hierarchies: things which subtype are inheriting behavior. You should setup your abs

```
abstract AbstractPerson
abstract AbstractStudent
```

This can be interpreted as follows. At the top we have AbstractPerson. Our interface here is "a Person is someone who has a name which c

```
get_name(x::AbstractPerson) = x.name
```

Thus codes which are written for an AbstractPerson can "know" (by our informal declaration of the interface) that get\_name will "just work" f

```
get_name(x::MusicStudent) = "Justin Bieber"
```

In this way, we can use get\_name to get the name, and how it was implemented (whether it's pulling something that had to be stored from n

## SMALL FUNCTIONS AND CONSTANT PROPAGATION

The next question to ask is, does storing information in functions and actions affect performance? The answer is yes, and in favor of the fun

```
likes_music(x::AbstractTeacher) = false
likes_music(x::AbstractStudent) = true
likes_music(x::AbstractPerson) = true
```

Now how many records would these people buy at a record store? If they don't like music, they will buy zero records. If they like music, then

```
function number_of_records(x::AbstractPerson)
    if !likes_music(x)
        return 0
    end
    num_records = rand(1:10)
    if typeof(x)

```

Let's check the code that is created:

```
x = Teacher("Randy",11)
println(number_of_records(x))
@code_llvm number_of_records(x)
```

on v0.6, we get:

on v0.6, we get:

```
; Function Attrs: uwtable
define i64 @julia_number_of_records_63848(i8** dereferenceable(16)) #0 !dbg !5 {
top:
    ret i64 0
}
```

Notice that the entire function compiled away, and the resulting compiled code is "return 0"! Then for a music student:

```
x = MusicStudent(10)
@code_typed number_of_records(x)
```

```
# Output
CodeInfo(:begin
  NewvarNode(:(num_records))
  goto 4 # line 30:
4: # line 32:
  @@0@@(Expr(:foreigncall, :(:jl_alloc_array_1d), Array{Float64,1}, svec(Any, Int64), Array{Float64,1}, 0, 10, 0))
  # meta: pop location
  # meta: pop location
  # meta: pop location
  # meta: pop location
  @@1@@(Expr(:invoke, MethodInstance for rand!{::MersenneTwister, ::Array{Float64,1}, ::Int64, ::Type{Base.Random.CloseOpen}}, :l
(MusicStudent Array{Float64,1}
```

we get a multiplication by 2, while for a regular person,

```
x = Person("Miguel")
@code_typed number_of_records(x)
```

```
# Output
CodeInfo(:begin
  NewvarNode(:(num_records))
  goto 4 # line 30:
4: # line 32:
  @@2@@(Expr(:foreigncall, :(:jl_alloc_array_1d), Array{Float64,1}, svec(Any, Int64), Array{Float64,1}, 0, 10, 0))
  # meta: pop location
  # meta: pop location
  # meta: pop location
  # meta: pop location
  @@3@@(Expr(:invoke, MethodInstance for rand!{::MersenneTwister, ::Array{Float64,1}, ::Int64, ::Type{Base.Random.CloseOpen}}, :l
(Person Array{Float64,1}
```

we do not get a multiplication by 2. This is all in the compiled-code, so this means that in one case the \*2 simply doesn't exist at runtime, no

The key thing to see from the typed code is that the "branches" (the if statements) all compiled away. Since types are known at compile time

This is the distinction between compile-time information and runtime information. At compile-time, what is known is:

1. The types of the inputs
2. Any types which can be inferred from the input types (via type-stability)
3. The function dispatches that will be internally called (from types which have been inferred)

Note that what cannot be inferred by the compiler is the information in fields. Information in fields is strictly runtime information. This is easy

Thus by putting our information into our functions and dispatches, we are actually giving the compiler more information to perform more opti

## TRAITS AND THTT

What we just saw is a "trait". Traits are compile-time designations about types which are distinct from their abstract hierarchy. likes\_music is

Traits can be more refined than just true/false. This can be done by having the return be a type itself. For example, we can create music ger

```
abstract MusicGenres
abstract RockGenre
These "simple types" are known as singleton types. This means that we can have traits like:
```

```
favorite_genre(x::AbstractPerson) = ClassicRock()
```

```
favorite_genre(x::MusicStudent) = Classical()
favorite_genre(x::AbstractTeacher) = AltRock()
```

This gives us all of the tools we need to compile the most efficient code, and structure our code around types/actions/dispatch to get high performance.

```
@traitfn ft(x::IsNice) = "Very nice!"
@traitfn ft(x::(!IsNice)) = "Not so nice!"
```

## COMPOSITION VS INHERITANCE

The last remark that is needed is a discussion of composition vs inheritance. While the previous discussions have all explained why "information hiding" is a good idea, composition vs inheritance is a more subtle issue.

Composition vs inheritance isn't a Julia issue, it's a long debate in object-oriented programming. The idea is that, inheritance is inherently (probably) more efficient than composition, but composition is more flexible.

<https://softwareengineering.stackexchange.com/questions/134097/why-should-i-prefer-composition-over-inheritance>

[https://en.wikipedia.org/wiki/Composition\\_over\\_inheritance](https://en.wikipedia.org/wiki/Composition_over_inheritance)

<https://www.thoughtworks.com/insights/blog/composition-vs-inheritance-how-choose>

So if possible, give composition a try. Say you have `MyType`, and it has some function `f` defined on it. This means that you can extend `MyType` with a new type `MyType2` that inherits from `MyType`.

```
struct MyType2
    mt::MyType
    ... # Other stuff
end

f(mt2::MyType2) = f(mt2.mt)
```

The pro here is that it's explicit: you've made the choice for each extension. The con is that this can require some extra code, though this can be mitigated by using macros.

What if you really really really want inheritance of fields? There are solutions via metaprogramming. One simple solution is the `@def` macro.

```
macro def(name, definition)
    return quote
        macro $(esc(name))()
            esc($(Expr(:quote, definition)))
        end
    end
end
```

This macro is very simple. What it does is compile-time copy/paste. For example:

```
@def give_it_a_name begin
    a = 2
    println(a)
end
```

defines a macro `@give_it_a_name` that will paste in those two lines of code wherever it is used. As another example, the reused fields of `Object` can be defined as follows:

```
@def add_generic_fields begin
    method_string::String
    n::Int64
    x::Array{T}
    f_x::T
    f_calls::Int64
    g_calls::Int64
    h_calls::Int64
end
```

and those fields can be copied around with

```
type LBGSSState{T}
  @add_generic_fields
  x_previous::Array{T}
  g::Array{T}
  g_previous::Array{T}
  rho::Array{T}
  # ... more fields ...
end
```

Because `@def` works at compile-time, there is no cost associated with this. Similar metaprogramming can be used to build an "inheritance f

```
# The abstract type
@base struct AbstractFoo{T}
  a
  b::Int
  c::T
  d::Vector{T}
end
```

```
# Inheritance
@extend struct Foo
where the @extend macro generates the type-definition:
```

```
struct Foo{T}
```

But it's just a package? Well, that's the beauty of Julia. Most of Julia is written in Julia, and Julia code is first class and performant (here, this

## CONCLUSION

Programming for type systems has a different architecture than object-oriented systems. Instead of being oriented around the objects and th

## EDITS 2/15/2018

Updated the type definitions to use "struct" instead of type, and updated the `@ode_def` macro to escape the name. Both of these are chang