Methods

Recall from Functions that a function is an object that maps a tuple of arguments to a return value, or throws an exception if no appropriate value can be returned. It is common for the same conceptual function or operation to be implemented quite differently for different types of arguments: adding two integers is very different from adding two floating-point numbers, both of which are distinct from adding an integer to a floating-point number. Despite their implementation differences, these operations all fall under the general concept of "addition". Accordingly, in Julia, these behaviors all belong to a single object: the + function.

To facilitate using many different implementations of the same concept smoothly, functions need not be defined all at once, but can rather be defined piecewise by providing specific behaviors for certain combinations of argument types and counts. A definition of one possible behavior for a function is called a *method*. Thus far, we have presented only examples of functions defined with a single method, applicable to all types of arguments. However, the signatures of method definitions can be annotated to indicate the types of arguments in addition to their number, and more than a single method definition may be provided. When a function is applied to a particular tuple of arguments, the most specific method applicable to those arguments is applied. Thus, the overall behavior of a function is a patchwork of the behaviors of its various method definitions. If the patchwork is well designed, even though the implementations of the methods may be quite different, the outward behavior of the function will appear seamless and consistent.

The choice of which method to execute when a function is applied is called *dispatch*. Julia allows the dispatch process to choose which of a function's methods to call based on the number of arguments given, and on the types of all of the function's arguments. This is different than traditional object-oriented languages, where dispatch occurs based only on the first argument, which often has a special argument syntax, and is sometimes implied rather than explicitly written as an argument. [1] Using all of a function's arguments to choose which method should be invoked, rather than just the first, is known as multiple dispatch. Multiple dispatch is particularly useful for mathematical code, where it makes little sense to artificially deem the operations to "belong" to one argument more than any of the others: does the addition operation in `x + y` belong to `x` any more than it does to `y`? The implementation of a mathematical operator generally depends on the types of all of its arguments. Even beyond mathematical operations, however, multiple dispatch ends up being a powerful and convenient paradigm for structuring and organizing programs.

Defining Methods

Until now, we have, in our examples, defined only functions with a single method having unconstrained argument types. Such functions behave just like they would in traditional dynamically typed languages. Nevertheless, we have used multiple dispatch and methods almost continually without being aware of it: all of Julia's standard functions and operators, like the aforementioned + function, have many methods defining their behavior over various possible combinations of argument type and count.

When defining a function, one can optionally constrain the types of parameters it is applicable to, using the `::` type-assertion operator, introduced in the section on Composite Types:

```
julia> f(x::Float64, y::Float64) = 2x + y
f (generic function with 1 method)
```

This function definition applies only to calls where `x` and `y` are both values of type `Float64`:

```
julia> f(2.0, 3.0)
7.0
```

Applying it to any other types of arguments will result in a `MethodError`:

```
julia> f(2.0, 3)
ERROR: MethodError: no method matching f(::Float64, ::Int64)
Closest candidates are:
  f(::Float64, !Matched::Float64) at none:1

julia> f(Float32(2.0), 3.0)
ERROR: MethodError: no method matching f(::Float32, ::Float64)
Closest candidates are:
  f(!Matched::Float64, ::Float64) at none:1

julia> f(2.0, "3.0")
ERROR: MethodError: no method matching f(::Float64, ::String)
Closest candidates are:
  f(::Float64, !Matched::Float64) at none:1

julia> f("2.0", "3.0")
ERROR: MethodError: no method matching f(::String, ::String)
```

As you can see, the arguments must be precisely of type `Float64`. Other numeric types, such as integers or 32-bit floating-point values, are not automatically converted to 64-bit floating-point, nor are strings parsed as numbers. Because `Float64` is a concrete type and concrete types cannot be subclassed in Julia, such a definition can only be applied to arguments that are exactly of type `Float64`. It may often be useful, however, to write more general methods where the declared parameter types are abstract:

```
julia> f(x::Number, y::Number) = 2x - y
f (generic function with 2 methods)

julia> f(2.0, 3)
1.0
```

This method definition applies to any pair of arguments that are instances of `Number`. They need not be of the same type, so long as they are each numeric values. The problem of handling disparate numeric types is delegated to the arithmetic operations in the expression `2x - y`.

To define a function with multiple methods, one simply defines the function multiple times, with different numbers and types of arguments. The first method definition for a function creates the function object, and subsequent method definitions add new methods to the existing function object. The most specific method definition matching the number and types of the arguments will be executed when the function is applied. Thus, the two method definitions above, taken together, define the behavior for `f` over all pairs of instances of the abstract type `Number` – but with a different behavior specific to pairs of `Float64` values. If one of the arguments is a 64-bit float but the other one is not, then the `f(Float64,Float64)` method cannot be called and the more general `f(Number,Number)` method must be used:

```
julia> f(2.0, 3.0)
7.0

julia> f(2, 3.0)
1.0

julia> f(2.0, 3)
1.0

julia> f(2, 3)
1
```

The `2x + y` definition is only used in the first case, while the `2x - y` definition is used in the others. No automatic casting or conversion of function arguments is ever performed: all conversion in Julia is non-magical and completely explicit. Conversion and Promotion, however, shows how clever application of sufficiently advanced technology can be indistinguishable from magic. [Clarke61]

For non-numeric values, and for fewer or more than two arguments, the function `f` remains undefined, and applying it will still result in a `MethodError`:

```
julia> f("foo", 3)
```

```
ERROR: MethodError: no method matching f(::String, ::Int64)
Closest candidates are:
  f(!Matched::Number, ::Number) at none:1

julia> f()
ERROR: MethodError: no method matching f()
Closest candidates are:
  f(!Matched::Float64, !Matched::Float64) at none:1
  f(!Matched::Number, !Matched::Number) at none:1
```

You can easily see which methods exist for a function by entering the function object itself in an interactive session:

```
julia> f
f (generic function with 2 methods)
```

This output tells us that `f` is a function object with two methods. To find out what the signatures of those methods are, use the `methods` function:

```
julia> methods(f)
# 2 methods for generic function "f":
[1] f(x::Float64, y::Float64) in Main at none:1
[2] f(x::Number, y::Number) in Main at none:1
```

which shows that `f` has two methods, one taking two `Float64` arguments and one taking arguments of type `Number`. It also indicates the file and line number where the methods were defined: because these methods were defined at the REPL, we get the apparent line number `none:1`.

In the absence of a type declaration with `::`, the type of a method parameter is `Any` by default, meaning that it is unconstrained since all values in Julia are instances of the abstract type `Any`. Thus, we can define a catch-all method for `f` like so:

```
julia> f(x,y) = println("Whoa there, Nelly.")
f (generic function with 3 methods)

julia> f("foo", 1)
Whoa there, Nelly.
```

This catch-all is less specific than any other possible method definition for a pair of parameter values, so it will only be called on pairs of arguments to which no other method definition applies.

Although it seems a simple concept, multiple dispatch on the types of values is perhaps the single most

powerful and central feature of the Julia language. Core operations typically have dozens of methods:

```
julia> methods(+)
# 180 methods for generic function "+":
[1] +(x::Bool, z::Complex{Bool}) in Base at complex.jl:227
[2] +(x::Bool, y::Bool) in Base at bool.jl:89
[3] +(x::Bool) in Base at bool.jl:86
[4] +(x::Bool, y::T) where T<:AbstractFloat in Base at bool.jl:96
[5] +(x::Bool, z::Complex) in Base at complex.jl:234
[6] +(a::Float16, b::Float16) in Base at float.jl:373
[7] +(x::Float32, y::Float32) in Base at float.jl:375
[8] +(x::Float64, y::Float64) in Base at float.jl:376
[9] +(z::Complex{Bool}, x::Bool) in Base at complex.jl:228
[10] +(z::Complex{Bool}, x::Real) in Base at complex.jl:242
[11] +(x::Char, y::Integer) in Base at char.jl:40
[12] +(c::BigInt, x::BigFloat) in Base.MPFR at mpfr.jl:307
[13] +(a::BigInt, b::BigInt, c::BigInt, d::BigInt, e::BigInt) in Base.GMP at gmp.jl:
[14] +(a::BigInt, b::BigInt, c::BigInt, d::BigInt) in Base.GMP at gmp.jl:391
[15] +(a::BigInt, b::BigInt, c::BigInt) in Base.GMP at gmp.jl:390
[16] +(x::BigInt, y::BigInt) in Base.GMP at gmp.jl:361
[17] +(x::BigInt, c::Union{UInt16, UInt32, UInt64, UInt8}) in Base.GMP at gmp.jl:398
...
[180] +(a, b, c, xs...) in Base at operators.jl:424
```

Multiple dispatch together with the flexible parametric type system give Julia its ability to abstractly express high-level algorithms decoupled from implementation details, yet generate efficient, specialized code to handle each case at run time.

Method Ambiguities

It is possible to define a set of function methods such that there is no unique most specific method applicable to some combinations of arguments:

```
julia> g(x::Float64, y) = 2x + y
g (generic function with 1 method)

julia> g(x, y::Float64) = x + 2y
g (generic function with 2 methods)

julia> g(2.0, 3)
7.0

julia> g(2, 3.0)
```

```
8.0

julia> g(2.0, 3.0)
ERROR: MethodError: g(::Float64, ::Float64) is ambiguous. Candidates:
  g(x::Float64, y) in Main at none:1
  g(x, y::Float64) in Main at none:1
Possible fix, define
  g(::Float64, ::Float64)
```

Here the call `g(2.0, 3.0)` could be handled by either the `g(Float64, Any)` or the `g(Any, Float64)` method, and neither is more specific than the other. In such cases, Julia raises a `MethodError` rather than arbitrarily picking a method. You can avoid method ambiguities by specifying an appropriate method for the intersection case:

```
julia> g(x::Float64, y::Float64) = 2x + 2y
g (generic function with 3 methods)

julia> g(2.0, 3)
7.0

julia> g(2, 3.0)
8.0

julia> g(2.0, 3.0)
10.0
```

It is recommended that the disambiguating method be defined first, since otherwise the ambiguity exists, if transiently, until the more specific method is defined.

In more complex cases, resolving method ambiguities involves a certain element of design; this topic is explored further below.

Parametric Methods

Method definitions can optionally have type parameters qualifying the signature:

```
julia> same_type(x::T, y::T) where {T} = true
same_type (generic function with 1 method)

julia> same_type(x,y) = false
same_type (generic function with 2 methods)
```

The first method applies whenever both arguments are of the same concrete type, regardless of what type that is, while the second method acts as a catch-all, covering all other cases. Thus, overall, this defines a boolean function that checks whether its two arguments are of the same type:

```
julia> same_type(1, 2)
true

julia> same_type(1, 2.0)
false

julia> same_type(1.0, 2.0)
true

julia> same_type("foo", 2.0)
false

julia> same_type("foo", "bar")
true

julia> same_type(Int32(1), Int64(2))
false
```

Such definitions correspond to methods whose type signatures are `UnionAll` types (see UnionAll Types).

This kind of definition of function behavior by dispatch is quite common – idiomatic, even – in Julia. Method type parameters are not restricted to being used as the types of arguments: they can be used anywhere a value would be in the signature of the function or body of the function. Here's an example where the method type parameter `T` is used as the type parameter to the parametric type `Vector{T}` in the method signature:

```
julia> myappend(v::Vector{T}, x::T) where {T} = [v..., x]
myappend (generic function with 1 method)

julia> myappend([1,2,3],4)
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> myappend([1,2,3],2.5)
ERROR: MethodError: no method matching myappend(::Array{Int64,1}, ::Float64)
```

```
Closest candidates are:
  myappend(::Array{T,1}, !Matched::T) where T at none:1

julia> myappend([1.0,2.0,3.0],4.0)
4-element Array{Float64,1}:
 1.0
 2.0
 3.0
 4.0

julia> myappend([1.0,2.0,3.0],4)
ERROR: MethodError: no method matching myappend(::Array{Float64,1}, ::Int64)
Closest candidates are:
  myappend(::Array{T,1}, !Matched::T) where T at none:1
```

As you can see, the type of the appended element must match the element type of the vector it is appended to, or else a `MethodError` is raised. In the following example, the method type parameter `T` is used as the return value:

```
julia> mytypeof(x::T) where {T} = T
mytypeof (generic function with 1 method)

julia> mytypeof(1)
Int64

julia> mytypeof(1.0)
Float64
```

Just as you can put subtype constraints on type parameters in type declarations (see Parametric Types), you can also constrain type parameters of methods:

```
julia> same_type_numeric(x::T, y::T) where {T<:Number} = true
same_type_numeric (generic function with 1 method)

julia> same_type_numeric(x::Number, y::Number) = false
same_type_numeric (generic function with 2 methods)

julia> same_type_numeric(1, 2)
true

julia> same_type_numeric(1, 2.0)
false
```

```
julia> same_type_numeric(1.0, 2.0)
true

julia> same_type_numeric("foo", 2.0)
ERROR: MethodError: no method matching same_type_numeric(::String, ::Float64)
Closest candidates are:
  same_type_numeric(!Matched::T, ::T) where T<:Number at none:1
  same_type_numeric(!Matched::Number, ::Number) at none:1

julia> same_type_numeric("foo", "bar")
ERROR: MethodError: no method matching same_type_numeric(::String, ::String)

julia> same_type_numeric(Int32(1), Int64(2))
false
```

The `same_type_numeric` function behaves much like the `same_type` function defined above, but is only defined for pairs of numbers.

Parametric methods allow the same syntax as `where` expressions used to write types (see UnionAll Types). If there is only a single parameter, the enclosing curly braces (in `where {T}`) can be omitted, but are often preferred for clarity. Multiple parameters can be separated with commas, e.g. `where {T, S<:Real}`, or written using nested `where`, e.g. `where S<:Real where T`.

Redefining Methods

When redefining a method or adding new methods, it is important to realize that these changes don't take effect immediately. This is key to Julia's ability to statically infer and compile code to run fast, without the usual JIT tricks and overhead. Indeed, any new method definition won't be visible to the current runtime environment, including Tasks and Threads (and any previously defined `@generated` functions). Let's start with an example to see what this means:

```
julia> function tryeval()
           @eval newfun() = 1
           newfun()
       end
tryeval (generic function with 1 method)

julia> tryeval()
ERROR: MethodError: no method matching newfun()
The applicable method may be too new: running in world age xxxx1, while current worl
Closest candidates are:
  newfun() at none:1 (method too new to be called from this world context.)
 in tryeval() at none:1
```

```
 ...

julia> newfun()
1
```

In this example, observe that the new definition for `newfun` has been created, but can't be immediately called. The new global is immediately visible to the `tryeval` function, so you could write `return newfun` (without parentheses). But neither you, nor any of your callers, nor the functions they call, or etc. can call this new method definition!

But there's an exception: future calls to `newfun` *from the REPL* work as expected, being able to both see and call the new definition of `newfun`.

However, future calls to `tryeval` will continue to see the definition of `newfun` as it was *at the previous statement at the REPL*, and thus before that call to `tryeval`.

You may want to try this for yourself to see how it works.

The implementation of this behavior is a "world age counter". This monotonically increasing value tracks each method definition operation. This allows describing "the set of method definitions visible to a given runtime environment" as a single number, or "world age". It also allows comparing the methods available in two worlds just by comparing their ordinal value. In the example above, we see that the "current world" (in which the method `newfun` exists), is one greater than the task-local "runtime world" that was fixed when the execution of `tryeval` started.

Sometimes it is necessary to get around this (for example, if you are implementing the above REPL). Fortunately, there is an easy solution: call the function using `Base.invokelatest`:

```
julia> function tryeval2()
           @eval newfun2() = 2
           Base.invokelatest(newfun2)
       end
tryeval2 (generic function with 1 method)

julia> tryeval2()
2
```

Finally, let's take a look at some more complex examples where this rule comes into play. Define a function `f(x)`, which initially has one method:

```
julia> f(x) = "original definition"
f (generic function with 1 method)
```

Start some other operations that use `f(x)`:

```
julia> g(x) = f(x)
g (generic function with 1 method)

julia> t = @async f(wait()); yield();
```

Now we add some new methods to `f(x)`:

```
julia> f(x::Int) = "definition for Int"
f (generic function with 2 methods)

julia> f(x::Type{Int}) = "definition for Type{Int}"
f (generic function with 3 methods)
```

Compare how these results differ:

```
julia> f(1)
"definition for Int"

julia> g(1)
"definition for Int"

julia> fetch(schedule(t, 1))
"original definition"

julia> t = @async f(wait()); yield();

julia> fetch(schedule(t, 1))
"definition for Int"
```

Design Patterns with Parametric Methods

While complex dispatch logic is not required for performance or usability, sometimes it can be the best way to express some algorithm. Here are a few common design patterns that come up sometimes when using dispatch in this way.

Extracting the type parameter from a super-type

Here is the correct code template for returning the element-type `T` of any arbitrary subtype of `AbstractArray`:

```
abstract type AbstractArray{T, N} end
eltype(::Type{<:AbstractArray{T}}) where {T} = T
```

using so-called triangular dispatch. Note that if `T` is a `UnionAll` type, as e.g. `eltype(Array{T} where T <: Integer)`, then `Any` is returned (as does the version of `eltype` in `Base`).

Another way, which used to be the only correct way before the advent of triangular dispatch in Julia v0.6, is:

```
abstract type AbstractArray{T, N} end
eltype(::Type{AbstractArray}) = Any
eltype(::Type{AbstractArray{T}}) where {T} = T
eltype(::Type{AbstractArray{T, N}}) where {T, N} = T
eltype(::Type{A}) where {A<:AbstractArray} = eltype(supertype(A))
```

Another possibility is the following, which could useful to adapt to cases where the parameter `T` would need to be matched more narrowly:

```
eltype(::Type{AbstractArray{T, N} where {T<:S, N<:M}}) where {M, S} = Any
eltype(::Type{AbstractArray{T, N} where {T<:S}}) where {N, S} = Any
eltype(::Type{AbstractArray{T, N} where {N<:M}}) where {M, T} = T
eltype(::Type{AbstractArray{T, N}}) where {T, N} = T
eltype(::Type{A}) where {A <: AbstractArray} = eltype(supertype(A))
```

One common mistake is to try and get the element-type by using introspection:

```
eltype_wrong(::Type{A}) where {A<:AbstractArray} = A.parameters[1]
```

However, it is not hard to construct cases where this will fail:

```
struct BitVector <: AbstractArray{Bool, 1}; end
```

Here we have created a type `BitVector` which has no parameters, but where the element-type is still fully specified, with `T` equal to `Bool`!

Building a similar type with a different type parameter

When building generic code, there is often a need for constructing a similar object with some change made to the layout of the type, also necessitating a change of the type parameters. For instance, you might have some sort of abstract array with an arbitrary element type and want to write your

computation on it with a specific element type. We must implement a method for each `AbstractArray{T}` subtype that describes how to compute this type transform. There is no general transform of one subtype into another subtype with a different parameter. (Quick review: do you see why this is?)

The subtypes of `AbstractArray` typically implement two methods to achieve this: A method to convert the input array to a subtype of a specific `AbstractArray{T, N}` abstract type; and a method to make a new uninitialized array with a specific element type. Sample implementations of these can be found in Julia Base. Here is a basic example usage of them, guaranteeing that `input` and `output` are of the same type:

```
input = convert(AbstractArray{Eltype}, input)
output = similar(input, Eltype)
```

As an extension of this, in cases where the algorithm needs a copy of the input array, `convert` is insufficient as the return value may alias the original input. Combining `similar` (to make the output array) and `copyto!` (to fill it with the input data) is a generic way to express the requirement for a mutable copy of the input argument:

```
copy_with_eltype(input, Eltype) = copyto!(similar(input, Eltype), input)
```

Iterated dispatch

In order to dispatch a multi-level parametric argument list, often it is best to separate each level of dispatch into distinct functions. This may sound similar in approach to single-dispatch, but as we shall see below, it is still more flexible.

For example, trying to dispatch on the element-type of an array will often run into ambiguous situations. Instead, commonly code will dispatch first on the container type, then recurse down to a more specific method based on eltype. In most cases, the algorithms lend themselves conveniently to this hierarchical approach, while in other cases, this rigor must be resolved manually. This dispatching branching can be observed, for example, in the logic to sum two matrices:

```
# First dispatch selects the map algorithm for element-wise summation.
+(a::Matrix, b::Matrix) = map(+, a, b)
# Then dispatch handles each element and selects the appropriate
# common element type for the computation.
+(a, b) = +(promote(a, b)...)
# Once the elements have the same type, they can be added.
# For example, via primitive operations exposed by the processor.
```

```
+(a::Float64, b::Float64) = Core.add(a, b)
```

Trait-based dispatch

A natural extension to the iterated dispatch above is to add a layer to method selection that allows to dispatch on sets of types which are independent from the sets defined by the type hierarchy. We could construct such a set by writing out a `Union` of the types in question, but then this set would not be extensible as `Union`-types cannot be altered after creation. However, such an extensible set can be programmed with a design pattern often referred to as a "Holy-trait".

This pattern is implemented by defining a generic function which computes a different singleton value (or type) for each trait-set to which the function arguments may belong to. If this function is pure there is no impact on performance compared to normal dispatch.

The example in the previous section glossed over the implementation details of `map` and `promote`, which both operate in terms of these traits. When iterating over a matrix, such as in the implementation of `map`, one important question is what order to use to traverse the data. When `AbstractArray` subtypes implement the `Base.IndexStyle` trait, other functions such as `map` can dispatch on this information to pick the best algorithm (see Abstract Array Interface). This means that each subtype does not need to implement a custom version of `map`, since the generic definitions + trait classes will enable the system to select the fastest version. Here a toy implementation of `map` illustrating the trait-based dispatch:

```
map(f, a::AbstractArray, b::AbstractArray) = map(Base.IndexStyle(a, b), f, a, b)
# generic implementation:
map(::Base.IndexCartesian, f, a::AbstractArray, b::AbstractArray) = ...
# linear-indexing implementation (faster)
map(::Base.IndexLinear, f, a::AbstractArray, b::AbstractArray) = ...
```

This trait-based approach is also present in the `promote` mechanism employed by the scalar `+`. It uses `promote_type`, which returns the optimal common type to compute the operation given the two types of the operands. This makes it possible to reduce the problem of implementing every function for every pair of possible type arguments, to the much smaller problem of implementing a conversion operation from each type to a common type, plus a table of preferred pair-wise promotion rules.

Output-type computation

The discussion of trait-based promotion provides a transition into our next design pattern: computing the output element type for a matrix operation.

For implementing primitive operations, such as addition, we use the `promote_type` function to compute

the desired output type. (As before, we saw this at work in the `promote` call in the call to `+`).

For more complex functions on matrices, it may be necessary to compute the expected return type for a more complex sequence of operations. This is often performed by the following steps:

1. Write a small function `op` that expresses the set of operations performed by the kernel of the algorithm.
2. Compute the element type `R` of the result matrix as `promote_op(op, argument_types...)`, where `argument_types` is computed from `eltype` applied to each input array.
3. Build the output matrix as `similar(R, dims)`, where `dims` are the desired dimensions of the output array.

For a more specific example, a generic square-matrix multiply pseudo-code might look like:

```
function matmul(a::AbstractMatrix, b::AbstractMatrix)
    op = (ai, bi) -> ai * bi + ai * bi

    ## this is insufficient because it assumes `one(eltype(a))` is constructable:
    # R = typeof(op(one(eltype(a)), one(eltype(b))))

    ## this fails because it assumes `a[1]` exists and is representative of all elem
    # R = typeof(op(a[1], b[1]))

    ## this is incorrect because it assumes that `+` calls `promote_type`
    ## but this is not true for some types, such as Bool:
    # R = promote_type(ai, bi)

    # this is wrong, since depending on the return value
    # of type-inference is very brittle (as well as not being optimizable):
    # R = Base.return_types(op, (eltype(a), eltype(b)))

    ## but, finally, this works:
    R = promote_op(op, eltype(a), eltype(b))
    ## although sometimes it may give a larger type than desired
    ## it will always give a correct type

    output = similar(b, R, (size(a, 1), size(b, 2)))
    if size(a, 2) > 0
        for j in 1:size(b, 2)
            for i in 1:size(a, 1)
                ## here we don't use `ab = zero(R)`,
                ## since `R` might be `Any` and `zero(Any)` is not defined
                ## we also must declare `ab::R` to make the type of `ab` constant ir
                ## since it is possible that typeof(a * b) != typeof(a * b + a * b)
```

```
                ab::R = a[i, 1] * b[1, j]
                for k in 2:size(a, 2)
                    ab += a[i, k] * b[k, j]
                end
                output[i, j] = ab
            end
        end
    end
    return output
end
```

Separate convert and kernel logic

One way to significantly cut down on compile-times and testing complexity is to isolate the logic for converting to the desired type and the computation. This lets the compiler specialize and inline the conversion logic independent from the rest of the body of the larger kernel.

This is a common pattern seen when converting from a larger class of types to the one specific argument type that is actually supported by the algorithm:

```
complexfunction(arg::Int) = ...
complexfunction(arg::Any) = complexfunction(convert(Int, arg))

matmul(a::T, b::T) = ...
matmul(a, b) = matmul(promote(a, b)...)
```

Parametrically-constrained Varargs methods

Function parameters can also be used to constrain the number of arguments that may be supplied to a "varargs" function (Varargs Functions). The notation `Vararg{T,N}` is used to indicate such a constraint. For example:

```
julia> bar(a,b,x::Vararg{Any,2}) = (a,b,x)
bar (generic function with 1 method)

julia> bar(1,2,3)
ERROR: MethodError: no method matching bar(::Int64, ::Int64, ::Int64)
Closest candidates are:
  bar(::Any, ::Any, ::Any, !Matched::Any) at none:1

julia> bar(1,2,3,4)
```

```
(1, 2, (3, 4))

julia> bar(1,2,3,4,5)
ERROR: MethodError: no method matching bar(::Int64, ::Int64, ::Int64, ::Int64, ::Int
Closest candidates are:
  bar(::Any, ::Any, ::Any, ::Any) at none:1
```

More usefully, it is possible to constrain varargs methods by a parameter. For example:

```
function getindex(A::AbstractArray{T,N}, indices::Vararg{Number,N}) where {T,N}
```

would be called only when the number of `indices` matches the dimensionality of the array.

When only the type of supplied arguments needs to be constrained `Vararg{T}` can be equivalently written as `T...`. For instance `f(x::Int...) = x` is a shorthand for `f(x::Vararg{Int}) = x`.

Note on Optional and keyword Arguments

As mentioned briefly in Functions, optional arguments are implemented as syntax for multiple method definitions. For example, this definition:

```
f(a=1,b=2) = a+2b
```

translates to the following three methods:

```
f(a,b) = a+2b
f(a) = f(a,2)
f() = f(1,2)
```

This means that calling `f()` is equivalent to calling `f(1,2)`. In this case the result is `5`, because `f(1,2)` invokes the first method of `f` above. However, this need not always be the case. If you define a fourth method that is more specialized for integers:

```
f(a::Int,b::Int) = a-2b
```

then the result of both `f()` and `f(1,2)` is `-3`. In other words, optional arguments are tied to a function, not to any specific method of that function. It depends on the types of the optional arguments which method is invoked. When optional arguments are defined in terms of a global variable, the type of the optional argument may even change at run-time.

Keyword arguments behave quite differently from ordinary positional arguments. In particular, they do not participate in method dispatch. Methods are dispatched based only on positional arguments, with keyword arguments processed after the matching method is identified.

Function-like objects

Methods are associated with types, so it is possible to make any arbitrary Julia object "callable" by adding methods to its type. (Such "callable" objects are sometimes called "functors.")

For example, you can define a type that stores the coefficients of a polynomial, but behaves like a function evaluating the polynomial:

```
julia> struct Polynomial{R}
           coeffs::Vector{R}
       end

julia> function (p::Polynomial)(x)
           v = p.coeffs[end]
           for i = (length(p.coeffs)-1):-1:1
               v = v*x + p.coeffs[i]
           end
           return v
       end

julia> (p::Polynomial)() = p(5)
```

Notice that the function is specified by type instead of by name. As with normal functions there is a terse syntax form. In the function body, `p` will refer to the object that was called. A `Polynomial` can be used as follows:

```
julia> p = Polynomial([1,10,100])
Polynomial{Int64}([1, 10, 100])

julia> p(3)
931

julia> p()
2551
```

This mechanism is also the key to how type constructors and closures (inner functions that refer to their surrounding environment) work in Julia.

Empty generic functions

Occasionally it is useful to introduce a generic function without yet adding methods. This can be used to separate interface definitions from implementations. It might also be done for the purpose of documentation or code readability. The syntax for this is an empty `function` block without a tuple of arguments:

```
function emptyfunc
end
```

Method design and the avoidance of ambiguities

Julia's method polymorphism is one of its most powerful features, yet exploiting this power can pose design challenges. In particular, in more complex method hierarchies it is not uncommon for ambiguities to arise.

Above, it was pointed out that one can resolve ambiguities like

```
f(x, y::Int) = 1
f(x::Int, y) = 2
```

by defining a method

```
f(x::Int, y::Int) = 3
```

This is often the right strategy; however, there are circumstances where following this advice blindly can be counterproductive. In particular, the more methods a generic function has, the more possibilities there are for ambiguities. When your method hierarchies get more complicated than this simple example, it can be worth your while to think carefully about alternative strategies.

Below we discuss particular challenges and some alternative ways to resolve such issues.

Tuple and NTuple arguments

`Tuple` (and `NTuple`) arguments present special challenges. For example,

```
f(x::NTuple{N,Int}) where {N} = 1
f(x::NTuple{N,Float64}) where {N} = 2
```

are ambiguous because of the possibility that `N == 0`: there are no elements to determine whether the `Int` or `Float64` variant should be called. To resolve the ambiguity, one approach is define a method for the empty tuple:

```
f(x::Tuple{}) = 3
```

Alternatively, for all methods but one you can insist that there is at least one element in the tuple:

```
f(x::NTuple{N,Int}) where {N} = 1           # this is the fallback
f(x::Tuple{Float64, Vararg{Float64}}) = 2   # this requires at least one Float64
```

Orthogonalize your design

When you might be tempted to dispatch on two or more arguments, consider whether a "wrapper" function might make for a simpler design. For example, instead of writing multiple variants:

```
f(x::A, y::A) = ...
f(x::A, y::B) = ...
f(x::B, y::A) = ...
f(x::B, y::B) = ...
```

you might consider defining

```
f(x::A, y::A) = ...
f(x, y) = f(g(x), g(y))
```

where `g` converts the argument to type `A`. This is a very specific example of the more general principle of orthogonal design, in which separate concepts are assigned to separate methods. Here, `g` will most likely need a fallback definition

```
g(x::A) = x
```

A related strategy exploits `promote` to bring `x` and `y` to a common type:

```
f(x::T, y::T) where {T} = ...
f(x, y) = f(promote(x, y)...)
```

One risk with this design is the possibility that if there is no suitable promotion method converting `x` and `y` to the same type, the second method will recurse on itself infinitely and trigger a stack overflow.

Dispatch on one argument at a time

If you need to dispatch on multiple arguments, and there are many fallbacks with too many combinations to make it practical to define all possible variants, then consider introducing a "name cascade" where (for example) you dispatch on the first argument and then call an internal method:

```
f(x::A, y) = _fA(x, y)
f(x::B, y) = _fB(x, y)
```

Then the internal methods `_fA` and `_fB` can dispatch on `y` without concern about ambiguities with each other with respect to `x`.

Be aware that this strategy has at least one major disadvantage: in many cases, it is not possible for users to further customize the behavior of `f` by defining further specializations of your exported function `f`. Instead, they have to define specializations for your internal methods `_fA` and `_fB`, and this blurs the lines between exported and internal methods.

Abstract containers and element types

Where possible, try to avoid defining methods that dispatch on specific element types of abstract containers. For example,

```
-(A::AbstractArray{T}, b::Date) where {T<:Date}
```

generates ambiguities for anyone who defines a method

```
-(A::MyArrayType{T}, b::T) where {T}
```

The best approach is to avoid defining *either* of these methods: instead, rely on a generic method `-(A::AbstractArray, b)` and make sure this method is implemented with generic calls (like `similar` and `-`) that do the right thing for each container type and element type *separately*. This is just a more complex variant of the advice to orthogonalize your methods.

When this approach is not possible, it may be worth starting a discussion with other developers about resolving the ambiguity; just because one method was defined first does not necessarily mean that it can't be modified or eliminated. As a last resort, one developer can define the "band-aid" method

```
-(A::MyArrayType{T}, b::Date) where {T<:Date} = ...
```

that resolves the ambiguity by brute force.

Complex method "cascades" with default arguments

If you are defining a method "cascade" that supplies defaults, be careful about dropping any arguments that correspond to potential defaults. For example, suppose you're writing a digital filtering algorithm and you have a method that handles the edges of the signal by applying padding:

```
function myfilter(A, kernel, ::Replicate)
    Apadded = replicate_edges(A, size(kernel))
    myfilter(Apadded, kernel)  # now perform the "real" computation
end
```

This will run afoul of a method that supplies default padding:

```
myfilter(A, kernel) = myfilter(A, kernel, Replicate()) # replicate the edge by defau
```

Together, these two methods generate an infinite recursion with `A` constantly growing bigger.

The better design would be to define your call hierarchy like this:

```
struct NoPad end  # indicate that no padding is desired, or that it's already applie

myfilter(A, kernel) = myfilter(A, kernel, Replicate())  # default boundary conditior

function myfilter(A, kernel, ::Replicate)
    Apadded = replicate_edges(A, size(kernel))
    myfilter(Apadded, kernel, NoPad())  # indicate the new boundary conditions
end

# other padding methods go here

function myfilter(A, kernel, ::NoPad)
    # Here's the "real" implementation of the core computation
end
```

`NoPad` is supplied in the same argument position as any other kind of padding, so it keeps the dispatch hierarchy well organized and with reduced likelihood of ambiguities. Moreover, it extends the "public" `myfilter` interface: a user who wants to control the padding explicitly can call the `NoPad` variant directly.

- 1 In C++ or Java, for example, in a method call like `obj.meth(arg1,arg2)`, the object obj "receives" the method call and is implicitly passed to the method via the `this` keyword, rather than as an explicit method argument. When the current `this` object is the receiver of a method call, it can be omitted altogether, writing just `meth(arg1,arg2)`, with `this` implied as the receiving object.

- Clarke61 Arthur C. Clarke, *Profiles of the Future* (1961): Clarke's Third Law.

« Types Constructors »