Constructors

Constructors [1] are functions that create new objects – specifically, instances of Composite Types. In Julia, type objects also serve as constructor functions: they create new instances of themselves when applied to an argument tuple as a function. This much was already mentioned briefly when composite types were introduced. For example:

```
julia> struct Foo
           bar
           baz
       end

julia> foo = Foo(1, 2)
Foo(1, 2)

julia> foo.bar
1

julia> foo.baz
2
```

For many types, forming new objects by binding their field values together is all that is ever needed to create instances. However, in some cases more functionality is required when creating composite objects. Sometimes invariants must be enforced, either by checking arguments or by transforming them. Recursive data structures, especially those that may be self-referential, often cannot be constructed cleanly without first being created in an incomplete state and then altered programmatically to be made whole, as a separate step from object creation. Sometimes, it's just convenient to be able to construct objects with fewer or different types of parameters than they have fields. Julia's system for object construction addresses all of these cases and more.

Outer Constructor Methods

A constructor is just like any other function in Julia in that its overall behavior is defined by the combined behavior of its methods. Accordingly, you can add functionality to a constructor by simply defining new methods. For example, let's say you want to add a constructor method for `Foo` objects that takes only one argument and uses the given value for both the `bar` and `baz` fields. This is simple:

```
julia> Foo(x) = Foo(x,x)
Foo

julia> Foo(1)
Foo(1, 1)
```

You could also add a zero-argument `Foo` constructor method that supplies default values for both of the `bar` and `baz` fields:

```
julia> Foo() = Foo(0)
Foo

julia> Foo()
Foo(0, 0)
```

Here the zero-argument constructor method calls the single-argument constructor method, which in turn calls the automatically provided two-argument constructor method. For reasons that will become clear very shortly, additional constructor methods declared as normal methods like this are called *outer* constructor methods. Outer constructor methods can only ever create a new instance by calling another constructor method, such as the automatically provided default ones.

Inner Constructor Methods

While outer constructor methods succeed in addressing the problem of providing additional convenience methods for constructing objects, they fail to address the other two use cases mentioned in the introduction of this chapter: enforcing invariants, and allowing construction of self-referential objects. For these problems, one needs *inner* constructor methods. An inner constructor method is like an outer constructor method, except for two differences:

1. It is declared inside the block of a type declaration, rather than outside of it like normal methods.
2. It has access to a special locally existent function called `new` that creates objects of the block's type.

For example, suppose one wants to declare a type that holds a pair of real numbers, subject to the constraint that the first number is not greater than the second one. One could declare it like this:

```
julia> struct OrderedPair
           x::Real
           y::Real
           OrderedPair(x,y) = x > y ? error("out of order") : new(x,y)
       end
```

Now `OrderedPair` objects can only be constructed such that `x <= y`:

```
julia> OrderedPair(1, 2)
OrderedPair(1, 2)

julia> OrderedPair(2,1)
ERROR: out of order
Stacktrace:
 [1] error at ./error.jl:33 [inlined]
 [2] OrderedPair(::Int64, ::Int64) at ./none:4
 [3] top-level scope
```

If the type were declared `mutable`, you could reach in and directly change the field values to violate this invariant. Of course, messing around with an object's internals uninvited is bad practice. You (or someone else) can also provide additional outer constructor methods at any later point, but once a type is declared, there is no way to add more inner constructor methods. Since outer constructor methods can only create objects by calling other constructor methods, ultimately, some inner constructor must be called to create an object. This guarantees that all objects of the declared type must come into existence by a call to one of the inner constructor methods provided with the type, thereby giving some degree of enforcement of a type's invariants.

If any inner constructor method is defined, no default constructor method is provided: it is presumed that you have supplied yourself with all the inner constructors you need. The default constructor is equivalent to writing your own inner constructor method that takes all of the object's fields as parameters (constrained to be of the correct type, if the corresponding field has a type), and passes them to `new`, returning the resulting object:

```
julia> struct Foo
           bar
           baz
           Foo(bar,baz) = new(bar,baz)
       end
```

This declaration has the same effect as the earlier definition of the `Foo` type without an explicit inner constructor method. The following two types are equivalent – one with a default constructor, the other with an explicit constructor:

```
julia> struct T1
           x::Int64
       end

julia> struct T2
```

```
            x::Int64
            T2(x) = new(x)
        end

julia> T1(1)
T1(1)

julia> T2(1)
T2(1)

julia> T1(1.0)
T1(1)

julia> T2(1.0)
T2(1)
```

It is good practice to provide as few inner constructor methods as possible: only those taking all arguments explicitly and enforcing essential error checking and transformation. Additional convenience constructor methods, supplying default values or auxiliary transformations, should be provided as outer constructors that call the inner constructors to do the heavy lifting. This separation is typically quite natural.

Incomplete Initialization

The final problem which has still not been addressed is construction of self-referential objects, or more generally, recursive data structures. Since the fundamental difficulty may not be immediately obvious, let us briefly explain it. Consider the following recursive type declaration:

```
julia> mutable struct SelfReferential
            obj::SelfReferential
        end
```

This type may appear innocuous enough, until one considers how to construct an instance of it. If `a` is an instance of `SelfReferential`, then a second instance can be created by the call:

```
julia> b = SelfReferential(a)
```

But how does one construct the first instance when no instance exists to provide as a valid value for its `obj` field? The only solution is to allow creating an incompletely initialized instance of `SelfReferential` with an unassigned `obj` field, and using that incomplete instance as a valid value for the `obj` field of another instance, such as, for example, itself.

To allow for the creation of incompletely initialized objects, Julia allows the `new` function to be called with fewer than the number of fields that the type has, returning an object with the unspecified fields uninitialized. The inner constructor method can then use the incomplete object, finishing its initialization before returning it. Here, for example, is another attempt at defining the `SelfReferential` type, this time using a zero-argument inner constructor returning instances having `obj` fields pointing to themselves:

```
julia> mutable struct SelfReferential
           obj::SelfReferential
           SelfReferential() = (x = new(); x.obj = x)
       end
```

We can verify that this constructor works and constructs objects that are, in fact, self-referential:

```
julia> x = SelfReferential();

julia> x === x
true

julia> x === x.obj
true

julia> x === x.obj.obj
true
```

Although it is generally a good idea to return a fully initialized object from an inner constructor, it is possible to return incompletely initialized objects:

```
julia> mutable struct Incomplete
           data
           Incomplete() = new()
       end

julia> z = Incomplete();
```

While you are allowed to create objects with uninitialized fields, any access to an uninitialized reference is an immediate error:

```
julia> z.data
ERROR: UndefRefError: access to undefined reference
```

This avoids the need to continually check for `null` values. However, not all object fields are references. Julia considers some types to be "plain data", meaning all of their data is self-contained and does not reference other objects. The plain data types consist of primitive types (e.g. `Int`) and immutable structs of other plain data types. The initial contents of a plain data type is undefined:

```
julia> struct HasPlain
           n::Int
           HasPlain() = new()
       end

julia> HasPlain()
HasPlain(438103441441)
```

Arrays of plain data types exhibit the same behavior.

You can pass incomplete objects to other functions from inner constructors to delegate their completion:

```
julia> mutable struct Lazy
           data
           Lazy(v) = complete_me(new(), v)
       end
```

As with incomplete objects returned from constructors, if `complete_me` or any of its callees try to access the `data` field of the `Lazy` object before it has been initialized, an error will be thrown immediately.

Parametric Constructors

Parametric types add a few wrinkles to the constructor story. Recall from Parametric Types that, by default, instances of parametric composite types can be constructed either with explicitly given type parameters or with type parameters implied by the types of the arguments given to the constructor. Here are some examples:

```
julia> struct Point{T<:Real}
           x::T
           y::T
       end

julia> Point(1,2) ## implicit T ##
Point{Int64}(1, 2)
```

```
julia> Point(1.0,2.5) ## implicit T ##
Point{Float64}(1.0, 2.5)

julia> Point(1,2.5) ## implicit T ##
ERROR: MethodError: no method matching Point(::Int64, ::Float64)
Closest candidates are:
  Point(::T, ::T) where T<:Real at none:2

julia> Point{Int64}(1, 2) ## explicit T ##
Point{Int64}(1, 2)

julia> Point{Int64}(1.0,2.5) ## explicit T ##
ERROR: InexactError: Int64(2.5)
Stacktrace:
[...]

julia> Point{Float64}(1.0, 2.5) ## explicit T ##
Point{Float64}(1.0, 2.5)

julia> Point{Float64}(1,2) ## explicit T ##
Point{Float64}(1.0, 2.0)
```

As you can see, for constructor calls with explicit type parameters, the arguments are converted to the implied field types: `Point{Int64}(1,2)` works, but `Point{Int64}(1.0,2.5)` raises an `InexactError` when converting `2.5` to `Int64`. When the type is implied by the arguments to the constructor call, as in `Point(1,2)`, then the types of the arguments must agree – otherwise the `T` cannot be determined – but any pair of real arguments with matching type may be given to the generic `Point` constructor.

What's really going on here is that `Point`, `Point{Float64}` and `Point{Int64}` are all different constructor functions. In fact, `Point{T}` is a distinct constructor function for each type `T`. Without any explicitly provided inner constructors, the declaration of the composite type `Point{T<:Real}` automatically provides an inner constructor, `Point{T}`, for each possible type `T<:Real`, that behaves just like non-parametric default inner constructors do. It also provides a single general outer `Point` constructor that takes pairs of real arguments, which must be of the same type. This automatic provision of constructors is equivalent to the following explicit declaration:

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

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

Notice that each definition looks like the form of constructor call that it handles. The call `Point{Int64}(1,2)` will invoke the definition `Point{T}(x,y)` inside the `struct` block. The outer constructor declaration, on the other hand, defines a method for the general `Point` constructor which only applies to pairs of values of the same real type. This declaration makes constructor calls without explicit type parameters, like `Point(1,2)` and `Point(1.0,2.5)`, work. Since the method declaration restricts the arguments to being of the same type, calls like `Point(1,2.5)`, with arguments of different types, result in "no method" errors.

Suppose we wanted to make the constructor call `Point(1,2.5)` work by "promoting" the integer value `1` to the floating-point value `1.0`. The simplest way to achieve this is to define the following additional outer constructor method:

```
julia> Point(x::Int64, y::Float64) = Point(convert(Float64,x),y);
```

This method uses the `convert` function to explicitly convert `x` to `Float64` and then delegates construction to the general constructor for the case where both arguments are `Float64`. With this method definition what was previously a `MethodError` now successfully creates a point of type `Point{Float64}`:

```
julia> Point(1,2.5)
Point{Float64}(1.0, 2.5)

julia> typeof(ans)
Point{Float64}
```

However, other similar calls still don't work:

```
julia> Point(1.5,2)
ERROR: MethodError: no method matching Point(::Float64, ::Int64)
Closest candidates are:
  Point(::T, !Matched::T) where T<:Real at none:1
```

For a more general way to make all such calls work sensibly, see Conversion and Promotion. At the risk of spoiling the suspense, we can reveal here that all it takes is the following outer method definition to make all calls to the general `Point` constructor work as one would expect:

```
julia> Point(x::Real, y::Real) = Point(promote(x,y)...);
```

The `promote` function converts all its arguments to a common type – in this case `Float64`. With this method definition, the `Point` constructor promotes its arguments the same way that numeric operators like `+` do, and works for all kinds of real numbers:

```
julia> Point(1.5,2)
Point{Float64}(1.5, 2.0)

julia> Point(1,1//2)
Point{Rational{Int64}}(1//1, 1//2)

julia> Point(1.0,1//2)
Point{Float64}(1.0, 0.5)
```

Thus, while the implicit type parameter constructors provided by default in Julia are fairly strict, it is possible to make them behave in a more relaxed but sensible manner quite easily. Moreover, since constructors can leverage all of the power of the type system, methods, and multiple dispatch, defining sophisticated behavior is typically quite simple.

Case Study: Rational

Perhaps the best way to tie all these pieces together is to present a real world example of a parametric composite type and its constructor methods. To that end, we implement our own rational number type `OurRational`, similar to Julia's built-in `Rational` type, defined in `rational.jl`:

```
julia> struct OurRational{T<:Integer} <: Real
           num::T
           den::T
           function OurRational{T}(num::T, den::T) where T<:Integer
               if num == 0 && den == 0
                    error("invalid rational: 0//0")
               end
               g = gcd(den, num)
               num = div(num, g)
               den = div(den, g)
               new(num, den)
           end
       end

julia> OurRational(n::T, d::T) where {T<:Integer} = OurRational{T}(n,d)
OurRational

julia> OurRational(n::Integer, d::Integer) = OurRational(promote(n,d)...)
```

```
OurRational

julia> OurRational(n::Integer) = OurRational(n,one(n))
OurRational

julia> ⊘(n::Integer, d::Integer) = OurRational(n,d)
⊘ (generic function with 1 method)

julia> ⊘(x::OurRational, y::Integer) = x.num ⊘ (x.den*y)
⊘ (generic function with 2 methods)

julia> ⊘(x::Integer, y::OurRational) = (x*y.den) ⊘ y.num
⊘ (generic function with 3 methods)

julia> ⊘(x::Complex, y::Real) = complex(real(x) ⊘ y, imag(x) ⊘ y)
⊘ (generic function with 4 methods)

julia> ⊘(x::Real, y::Complex) = (x*y') ⊘ real(y*y')
⊘ (generic function with 5 methods)

julia> function ⊘(x::Complex, y::Complex)
           xy = x*y'
           yy = real(y*y')
           complex(real(xy) ⊘ yy, imag(xy) ⊘ yy)
       end
⊘ (generic function with 6 methods)
```

The first line – `struct OurRational{T<:Integer} <: Real` – declares that `OurRational` takes one type parameter of an integer type, and is itself a real type. The field declarations `num::T` and `den::T` indicate that the data held in a `OurRational{T}` object are a pair of integers of type `T`, one representing the rational value's numerator and the other representing its denominator.

Now things get interesting. `OurRational` has a single inner constructor method which checks that both of `num` and `den` aren't zero and ensures that every rational is constructed in "lowest terms" with a non-negative denominator. This is accomplished by dividing the given numerator and denominator values by their greatest common divisor, computed using the `gcd` function. Since `gcd` returns the greatest common divisor of its arguments with sign matching the first argument (`den` here), after this division the new value of `den` is guaranteed to be non-negative. Because this is the only inner constructor for `OurRational`, we can be certain that `OurRational` objects are always constructed in this normalized form.

`OurRational` also provides several outer constructor methods for convenience. The first is the "standard" general constructor that infers the type parameter `T` from the type of the numerator and

denominator when they have the same type. The second applies when the given numerator and denominator values have different types: it promotes them to a common type and then delegates construction to the outer constructor for arguments of matching type. The third outer constructor turns integer values into rationals by supplying a value of `1` as the denominator.

Following the outer constructor definitions, we defined a number of methods for the `⊘` operator, which provides a syntax for writing rationals (e.g. `1 ⊘ 2`). Julia's `Rational` type uses the `//` operator for this purpose. Before these definitions, `⊘` is a completely undefined operator with only syntax and no meaning. Afterwards, it behaves just as described in Rational Numbers – its entire behavior is defined in these few lines. The first and most basic definition just makes `a ⊘ b` construct a `OurRational` by applying the `OurRational` constructor to `a` and `b` when they are integers. When one of the operands of `⊘` is already a rational number, we construct a new rational for the resulting ratio slightly differently; this behavior is actually identical to division of a rational with an integer. Finally, applying `⊘` to complex integral values creates an instance of `Complex{OurRational}` – a complex number whose real and imaginary parts are rationals:

```
julia> z = (1 + 2im) ⊘ (1 - 2im);

julia> typeof(z)
Complex{OurRational{Int64}}

julia> typeof(z) <: Complex{OurRational}
false
```

Thus, although the `⊘` operator usually returns an instance of `OurRational`, if either of its arguments are complex integers, it will return an instance of `Complex{OurRational}` instead. The interested reader should consider perusing the rest of `rational.jl`: it is short, self-contained, and implements an entire basic Julia type.

Outer-only constructors

As we have seen, a typical parametric type has inner constructors that are called when type parameters are known; e.g. they apply to `Point{Int}` but not to `Point`. Optionally, outer constructors that determine type parameters automatically can be added, for example constructing a `Point{Int}` from the call `Point(1,2)`. Outer constructors call inner constructors to actually make instances. However, in some cases one would rather not provide inner constructors, so that specific type parameters cannot be requested manually.

For example, say we define a type that stores a vector along with an accurate representation of its sum:

```
julia> struct SummedArray{T<:Number,S<:Number}
           data::Vector{T}
           sum::S
       end

julia> SummedArray(Int32[1; 2; 3], Int32(6))
SummedArray{Int32,Int32}(Int32[1, 2, 3], 6)
```

The problem is that we want `S` to be a larger type than `T`, so that we can sum many elements with less information loss. For example, when `T` is `Int32`, we would like `S` to be `Int64`. Therefore we want to avoid an interface that allows the user to construct instances of the type `SummedArray{Int32,Int32}`. One way to do this is to provide a constructor only for `SummedArray`, but inside the `struct` definition block to suppress generation of default constructors:

```
julia> struct SummedArray{T<:Number,S<:Number}
           data::Vector{T}
           sum::S
           function SummedArray(a::Vector{T}) where T
               S = widen(T)
               new{T,S}(a, sum(S, a))
           end
       end

julia> SummedArray(Int32[1; 2; 3], Int32(6))
ERROR: MethodError: no method matching SummedArray(::Array{Int32,1}, ::Int32)
Closest candidates are:
  SummedArray(::Array{T,1}) where T at none:4
```

This constructor will be invoked by the syntax `SummedArray(a)`. The syntax `new{T,S}` allows specifying parameters for the type to be constructed, i.e. this call will return a `SummedArray{T,S}`. `new{T,S}` can be used in any constructor definition, but for convenience the parameters to `new{}` are automatically derived from the type being constructed when possible.

- 1 Nomenclature: while the term "constructor" generally refers to the entire function which constructs objects of a type, it is common to abuse terminology slightly and refer to specific constructor methods as "constructors". In such situations, it is generally clear from the context that the term is used to mean "constructor method" rather than "constructor function", especially as it is often used in the sense of singling out a particular method of the constructor from all of the others.

« Methods Conversion and Promotion »