Sunday, April 17, 2005

Seeing Metaclasses Clearly

If you're new to metaprogramming in Ruby and you'd like to start using it, perhaps these four methods could give you a bit more vision.

```
class Object
  # The hidden singleton lurks behind everyone
  def metaclass; class << self; self; end; end
  def meta_eval &blk; metaclass.instance_eval &blk; end

  # Adds methods to a metaclass
  def meta_def name, &blk
    meta_eval { define_method name, &blk }
  end

  # Defines an instance method within a class
  def class_def name, &blk
    class_eval { define_method name, &blk }
  end
end
```

I've been keeping these methods in a file called `metaid.rb` and it's a start toward building a little library that can simplify use of metaclasses. Let's talk about metaclasses and I advise you to keep `metaid.rb` at your side. Take time to run some code from this article and you'll understand much better.

About Classes

Well, what is a `Class`? Let’s create a simple object and see.

```
>> class MailTruck
>>   attr_accessor :driver, :route
>>   def initialize( driver, route )
>>     @driver, @route = driver, route
>>   end
>> end

>> m = MailTruck.new( "Harold", ['12 Corrigan Way', '23 Antler Ave'] )
=> #<MailTruck:0x81cfb94 @route=["12 Corrigan Way", "23 Antler Ave"],
                          @driver="Harold">
>> m.class
=> MailTruck
```

An Object is storage for variables. Instance variables. A `MailTruck` object, once initialized, will have a `@driver` and a `@route` variable. It can hold any other variables as well.

```
>> m.instance_variable_set( "@speed", 45 )
=> 45
>> m.driver
=> "Harold"
```

Okay, so the `@driver` instance variable has an accessor. When Ruby sees `attr_accessor :driver` in the `MailTruck` class definition, you get reader and writer methods. The methods `driver` and `driver=`.

These methods are stored in the class. So the instance variable is in the object and the accessor methods are in the class. They’re in two completely different spots.

It’s an important lesson: **objects do not store methods, only classes can**.

Classes Are Objects

Okay, but classes are objects, right? I mean if *everything is an object* in Ruby, then classes and objects should both be objects. Which makes them the same?

Sure, classes are objects. You can run all the same methods on classes that you can run on object.

Look, they each have their own ID in the symbol table.

```
>> m.object_id
=> 68058570
>> MailTruck.object_id
=> 68069450
```

But I’ve already told you: classes store methods. They’re different. Now I know you’re probably a bit confused wondering, “If a class is an object, but objects are built on classes, isn’t there a big confusing infinite cycle here that you’re not explaining?”

No, there’s not. I hate to break it to you, but a *class isn’t really an object*. From Ruby’s source code:

```
struct RObject {
  struct RBasic basic;
  struct st_table *iv_tbl;
};

struct RClass {
  struct RBasic basic;
  struct st_table *iv_tbl;
  struct st_table *m_tbl;
  VALUE super;
};
```

Look! A class has an `m_tbl` (a symbol table for storing methods) and a `superclass` (pointer to a superclass).

But let me reassure you. *To a Ruby programmer, a class is an object*. Because it meets the two big criteria: you can store instance variables in a class and it is descended from the `Object` class. That’s it.

```
>> o = Object.new
=> #<Object:0x815c45c>
>> o.class
=> Object

>> Class.superclass.superclass
=> Object
```

```
>> Object.class
=> Class
>> Object.superclass
=> nil
```

The `Object` class sits at the very head of the table and comes down to participate only when it has methods that can't be found anywhere else.

What On Earth Are Metaclasses?

The term *metaclass* is supposed mean "a class which defines classes." This definition doesn't really work with Ruby, though, since "a class which defines a class" is simply: a `Class`.

Look at how you can add a method in the `Class` class and then use it in class definitions.

```
>> class Class
>>   def attr_abort( *args )
>>     abort "Please no more attributes today."
>>   end
>> end
>>
>> class MyNewClass
>>   attr_abort :id, :diagram, :telegram
>> end
```

Which prints `Please no more attributes today.` The `attr_abort` method can be used in definitions.

You're constantly defining and redefining classes in Ruby. It's not meta, it's just part of the code. Classes hold methods. How can you complicate that?

Since the earlier definition doesn't really work, I like to think of the Ruby metaclass as "a class which an object uses to redefine itself."

Do Objects Need Metaclasses?

Objects can't hold methods. Most objects don't need to hold methods.

But sometimes you may want an object to have some methods. Sometimes that's your answer to a problem. You can't do that. But Matz has given us metaclasses which are good enough.

In the YAML library, you can customize the properties shown when an object is output.

```
>> require 'yaml'
>> class << m
>>   def to_yaml_properties
>>     ['@driver', '@route']
>>   end
>> end

>> YAML::dump m
--- !ruby/object:MailTruck
driver: Harold
route:
  - 12 Corrigan Way
  - 23 Antler Ave
```

This is handy if you want to dump one specific object with a certain style of YAML without effecting every object from that class. In the above example, only the object in the `m` variable will be output with its properties in order. All other `MailTruck` objects will be output in whatever way the YAML library chooses. Sometimes we may want to display a certain string one way without needing to modify the `String` class (which affects every string in your code).

So the object in the `m` variable has its own special `to_yaml_properties` method. It's stored in a metaclass. The metaclass stores methods for the object and sits right in the inheritance chain.

We could also add the `to_yaml_properties` method with this convenient syntax:

```
def m.to_yaml_properties
  ['@driver', '@route']
end
```

If you have the `metaid.rb` methods from the top of this article loaded, try this:

```
>> m.metaclass
=> #<Class:#<MailTruck:0x81cfb94>>
>> m.metaclass.class
=> Class
```

```
>> m.metaclass.superclass
=> #<Class:MailTruck>
>> m.metaclass.instance_methods
=> [..., "to_yaml_properties", ...]
>> m.singleton_methods
=> ["to_yaml_properties"]
```

When you use the `class << m` syntax, you're opening up a metaclass. Ruby calls these *virtual classes*. Notice the result of `m.metaclass`. A class attached to an object: `#<Class:#<MailTruck:0x81cfb94>>`.

When an object finds methods in an attached metaclass, these methods are referred to as the object's *singleton methods* rather than the *object's metaclass' instance methods* (if you get my drift.) And since there can only be a single metaclass attached to an object, it's called a _single_ton.

It's much easier to see metaclasses when you're using the `metaclass` method. Normally, you would need to use `( class << self; self; end )` wherever you wanted to root out a metaclass. But this makes it much simpler.

Do Metaclasses Need Metaclasses?

```
>> m.metaclass.metaclass
=> #<Class:#<Class:#<MailTruck:0x81cfb94>>>
>> m.metaclass.metaclass.metaclass
=> #<Class:#<Class:#<Class:#<MailTruck:0x81cfb94>>>>
```

Check out those frivolous metaclasses we're creating. So what can we do with a metaclass of a metaclass?

Well, the same thing we do with a normal metaclass. A normal metaclass holds methods for an object. So a metaclass of a metaclass holds methods for that metaclass—which is just an object, of course!

The problem with a metaclass of a metaclass is that there's not much practical use for them. You can only use the methods if you're deep inside the chain and we don't really want to spend much time down there.

```
>> m.meta_eval do
>>   self.meta_eval do
>>     self.meta_eval do
>>       def ribbit; "*ribbit*"; end
>>     end
>>   end
>> end

>> m.metaclass.metaclass.metaclass.singleton_methods
=> ["class_def", "metaclass", "constants", "meta_def",
    "attr_test", "nesting", "ribbit"]
```

Metaclasses are only really useful one level deep. You want to have give methods to an object. Or, as you will see, you might want a specific class to have a metaclass. Beyond that, you're just storing methods in these obscure metaclasses that no one can really get at. Which you might need to do sometime. Who knows.

The important thing to know at this point is: **metaclasses don't go *up*, they go *out*.** Yes, when you create a metaclass for an object, it happens to intercept method calls before the object's inheritance chain. But that doesn't mean inheritance is affected by further metaclasses. When you create a metaclass of a metaclass, it has *no* affect on the object referred to by the original metaclass.

Metaclasses Have One More Funky Trick For Classes and It's The Crucial Trick In The Metaprogrammer's Handbook

One more point and I believe this one is the juiciest. If you read the rest of this essay and quit before this section, you've come away without the most important lesson. You may know some nice things about objects and metaclasses, but it all pales.

I'm going to reiterate two previous statements about classes and build on them.

1. Class are objects. This means they can hold *instance variables*.
2. Metaclasses hold *instance methods*. When attached to an object, these methods become *singleton methods*. These methods intercept calls before they trickle up the chain of inheritance.

Have you ever used instance variables in a class before? I don't mean in a class method. I mean in the class itself.

```
class MailTruck
  @trucks = []
  def MailTruck.add( truck )
    @trucks << truck
  end
end
```

Why not just use a class variable?

```
class MailTruck
  @@trucks = []
  def MailTruck.add( truck )
    @@trucks << truck
  end
end
```

They work exactly the same, right? I mean it doesn't matter, does it?

Here are two reasons you've probably been using class variables rather than class instance variables:

1. Class variables are clearly class variables. They have two at-symbols. Less confusion.
2. Class variables can be referenced in instance methods, if needed.

See, this works properly:

```
class MailTruck
  @@trucks = []
  def MailTruck.add( truck )
    @@trucks << truck
  end
  def say_hi
    puts "Hi, I'm one of #{@@trucks.length} trucks!"
  end
end
```

But this does not:

```
class MailTruck
  @trucks = []
  def MailTruck.add( truck )
    @trucks << truck
  end
  def say_hi
```

```
    puts "Hi, I'm one of #{@trucks.length} trucks!"
  end
end
```

So what are instance variables good for? What a waste of space! I'm never using them again! (Yes, please stick to class variables in situations like the above.)

Let me also point out that metaclasses are again showing up above, since **every class method is stored in a metaclass**. That's simply just how it works.

Which is why you can also use `self`:

```
class MailTruck
  def self.add( truck )
    @@trucks << truck
  end
end
```

Or the singleton syntax:

```
class MailTruck
  class << self
    def add( truck )
      @@trucks << truck
    end
  end
end
```

Class instance variables and metaclass instance methods are really pretty pointless in a plain old class. But when inheritance enters the mix, the party comes alive. Writhing bodies and drunken madness, believe me.

```
class MailTruck
  def self.company( name )
    meta_def :company do; name; end
  end
end
```

The above method is remarkably simple, but excavates a beachhead worth of possibilities. A new `company` class method is added to MailTruck that can be used in a class definition.

```
class HappyTruck < MailTruck
  company "Happy's -- We Bring the Mail, and That's It!"
end
```

Okay, so the `company` class method gets executed with the Happy’s company name and slogan. What does `meta_def` do with it??

Well, the meat of meta arrives here. The `meta_def` adds a new method called `company` to the `HappyTruck` metaclass. The beauty of this is that the method is **not added to the `MailTruck` metaclass, but to the derived class `HappyTruck`.**

This may seem simple, but it’s very powerful. You can write simple class methods which will add class methods to a derived class. This is the secret to Rails and Ruby/X11 and so many other examples of metaprogramming in Ruby.

Dwemthy’s Array

I discovered most of this while building Dwemthy’s Array for my cartoon Ruby book. I was able to simplify the `Creature` code (which gives a readability to the RPG) down to this fragment:

```
class Creature
  def self.traits( *arr )
    return @traits if arr.empty?
    attr_accessor *arr
    arr.each do |trait|
      meta_def trait do |val|
        @traits ||= {}
        @traits[trait] = val
      end
    end
    class_def :initialize do
      self.class.traits.each do |k,v|
        instance_variable_set( "@#{k}", v )
      end
    end
  end
end
```

The `meta_def` and `class_def` help make the metaprogramming a bit more clear. Pay close attention to the use of instance variables in the **`meta_def`**. If you want to understand just why class variables won't work in this situation, then try changing the instance variables to class variables above.

Then, start creating monsters as described on the Dwemthy's Array page and you'll watch them step all over each other.

11:51 PM