Ruby's Eigenclasses Demystified

by Andrea Singh | June 24, 2011

In Ruby, classes and objects are intricately connected in ways that might not be immediately obvious. Owing to their special use-case, it is easy to lose sight of the fact that classes are themselves objects in Ruby. What makes them different, vis-à-vis regular objects, is that they serve as a blueprint for object creation and also that they are part of a hierarchy of classes. That is, classes can have instances (objects), superclasses (parents) and subclasses (children).

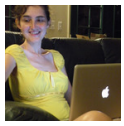
If classes are also objects it follows that they must have a class of their own. In Ruby the class of all class objects is the class Class:

```
# one way to instantiate a new class
Dog = Class.new

# a more conventional way of class construction
class Dog
    # some doggie behavior here
end

Dog.class
=> Class
```

The Method Lookup Path

The relationship between classes and objects becomes important in the context of the **method lookup path**. When you call a method on an object, Ruby first looks for that method in the object's class. Failing to find it there, it next checks the superclass of the object's class and keeps on going up through the class hierarchy until it hits the topmost class which as of Ruby 1.9 is BasicObject. The diagram below illustrates the way Ruby traverses the class hierarchy in search of a given method:

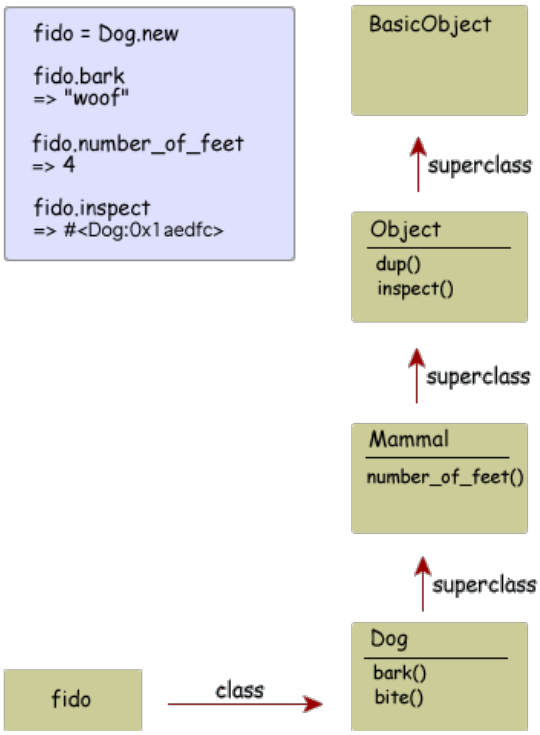
Andrea Singh



Welcome to Made by DNA. A hodgepodge of tips, tutorials and tidbits on Ruby, Rails, Javascript and other software technologies.

I am currently working on an exciting biological data integration project at UC Berkeley.

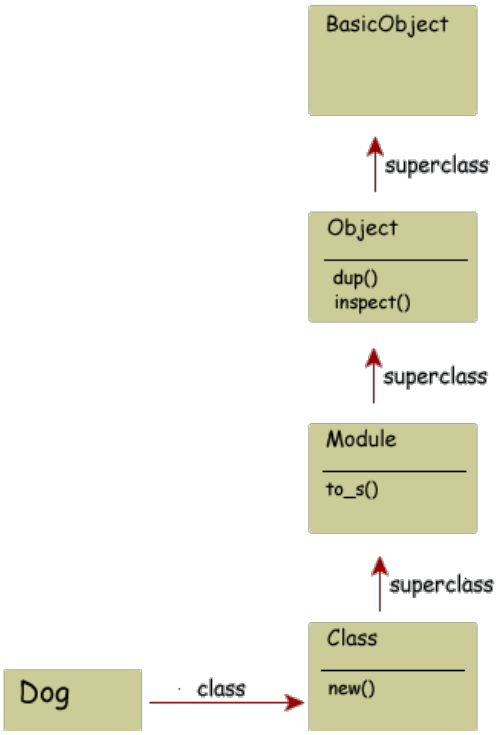
If you'd like to contact me, write to info@madebydna.com.



The method lookup path for classes works in a similar fashion. When you call a method on a class object, it will first look in its class, which as we have established is Class, and then continue on to its superclass, Module, all the way up to BasicObject:

```
Dog.class
=> Class
Class.superclass
=> Module
Module.superclass
=> Object
Object.superclass
=> BasicObject
```

And again visually:



As we can see `BasicObject` and `Object` lie at the root of the class hierarchy, so this means that all objects, whether they be regular objects or class objects, will eventually inherit the instance methods defined in those classes.

Singleton Methods

The primary function of classes is to define the behavior of their instances. For example, the behavior of dog instances in general is placed in the Dog class. However, in Ruby we can also confer unique behavior to individual objects. Meaning that we can create methods for a specific dog object that are *not* shared by other instances of the Dog class. These methods are often referred to as **singleton methods** as they belong to a single object only.

To re-iterate, singleton methods are defined on the object itself instead of in its class:

```
snoopy = Dog.new
def snoopy.alter_ego
  "Red Baron"
end

snoopy.alter_ego
# => "Red Baron"

fido.alter_ego
# => NoMethodError: undefined method `alter_ego' for #<Dog:0x00000
```

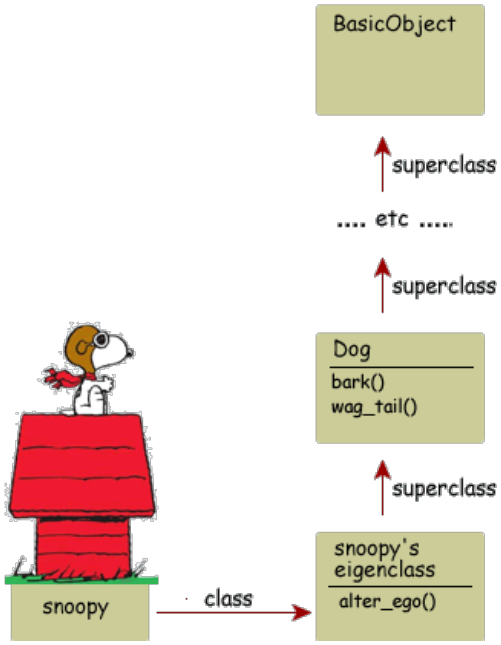
Eigenclasses

As noted above, when a method gets called on an object, Ruby first looks in the object's class for the method definition. In the case of singleton methods, the method is obviously *not* added to the object's class, since that method is not available to other instances of that class. So where the heck is it? And, how does Ruby manage to find both the object's singleton methods and also the general instance methods without disrupting the normal method lookup path?

As it turns out, Ruby accomplishes all this in typical neat fashion. First a new anonymous class is created to hold the object's singleton methods. Then this anonymous class assumes the role of the object's class and the original class is

re-designated as the superclass of that anonymous class. As such, the normal method lookup pattern is unaltered - both the singleton methods (in the anonymous class) and the instance methods (in the superclass) will be found along the method lookup path when the object receives a method call (see diagram below).

There are many alternate names for this dynamically created anonymous class that Ruby inserts into the method lookup path: metaclasses, ghost classes, singleton classes and eigenclasses. The word "eigen" comes from the German, roughly connoting something like "one's very own". Eigenclass seems to have found favor in the Ruby community, so I'll stick to that.



Note: In addition to defining singleton methods using the object name (e.g. `def snoopy.alter_ego`), you can also add methods to eigenclasses using this special syntax:

```
class << snoopy
  def alter_ego
    "Red Baron"
  end
end
```

The `class <<` syntax opens the eigenclass of whatever object you pass to it and lets you define a method straight in there.

Class Methods

Being objects, classes too can have singleton methods. In fact, what we commonly think of as **class methods** are nothing more than singleton methods for classes - methods owned by a singular object that incidentally happens to be a class. As with all singleton methods, class methods are also housed in an eigenclass.

Class singleton methods can be created in a variety of ways:

```
class Dog
  def self.closest_relative
    "wolf"
  end
end
```

```ruby
class Dog
  class << self
    def closest_relative
      "wolf"
    end
  end
end
```

```ruby
def Dog.closest_relative
  "wolf"
end
```

```ruby
class << Dog
  def closest_relative
    "wolf"
  end
end
```

All the examples above are identical in that they all open the eigenclass of the object Dog and define a (class) method directly in there.

Eigenclasses Revealed

Eigenclasses are not only anonymous; under normal circumstances they are hidden from view. This, despite the fact that they have been designated as the first stop in the method lookup path:

```ruby
class << Dog
  def closest_relative
   "wolf"
  end
end

Dog.class
# => Class
```

In the code above, the Dog class is still reporting its class as the class Class even though we have added a class method, which should have caused the class to be changed to be the eigenclass. Nonetheless, there is a way we can get objects to reveal their eigenclass:

```ruby
class Object
  def eigenclass
    class << self
      self
    end
  end
end
```

This code might appear a bit puzzling at first glance, so let's see if we can figure out what's going on here.

First off, we know that the Object class is one of the ultimate ancestors of all classes in Ruby, which is why it is a good place to define methods that you want to be universally available.

To understand this code we have to keep track of where and how the value of `self` changes. Immediately inside the `eigenclass()` instance method, `self` is the object that the method was originally called on. We then open the eigenclass of this receiver

object with the `class << self` syntax. Now that we are in the scope of the eigenclass, the value of `self` changes and now refers to the eigenclass object itself. By returning the value of `self` from within this eigenclass, we trigger Ruby's built-in `to_s` object identifier - allowing us get a glimpse of the elusive eigenclass.

Here's how we can make actual use of this `eigenclass()` instance method to bring eigenclasses out into the open.

```
class Dog
end

snoopy = Dog.new
=> #<Dog:0x00000001c4a170>

snoopy.eigenclass
=> #<Class:#<Dog:0x00000001c4a170>>

snoopy.eigenclass.superclass
=> Dog
```

Ruby's to_s Method

The `to_s` method is defined in the Object class and returns a string representing the object it was called on. This string is a composite of the object's class name followed by a unique object ID. For example, `#<Mouse:0x00000001c4a170>`.

Class objects (instances of the class Class) such as Class, Object or String override the `to_s` method and simply return their name instead.

```
Dog.to_s
=> "Dog"
```

In actuality, the overriding of the `to_s` method for classes is done in Module which is the superclass of Class. Here is how the Ruby documentation describes the Module `to_s` method:

> Returns a string representing the module or class. For basic classes and modules this is the name. For singletons, we show information on the thing we are attached to as well.

Note the bit about singletons including information about the attached object as well. This explains why when we call `to_s` on the eigenclass of snoopy we get back the anonymously named "Class" followed by the identifier for its attached object: `#<Class:#<Dog:0x00000001c4a170>>`. Calling `to_s` on the superclass of the eigenclass, however, references the original (non-singleton) class Dog and so simply returns the name "Dog".

Eigenclasses and Class Inheritance

Just as an object has access to instance methods defined in the superclasses of its class, so do classes have access to the class methods defined in their ancestor classes. Take a look at the code below:

```
class Mammal
  def self.warm_blooded?
    true
  end
end

class Dog < Mammal
  def self.closest_relative
    "wolf"
```

```
  end
end

Dog.closest_relative
# => "wolf"
Dog.warm_blooded?
# => true
```

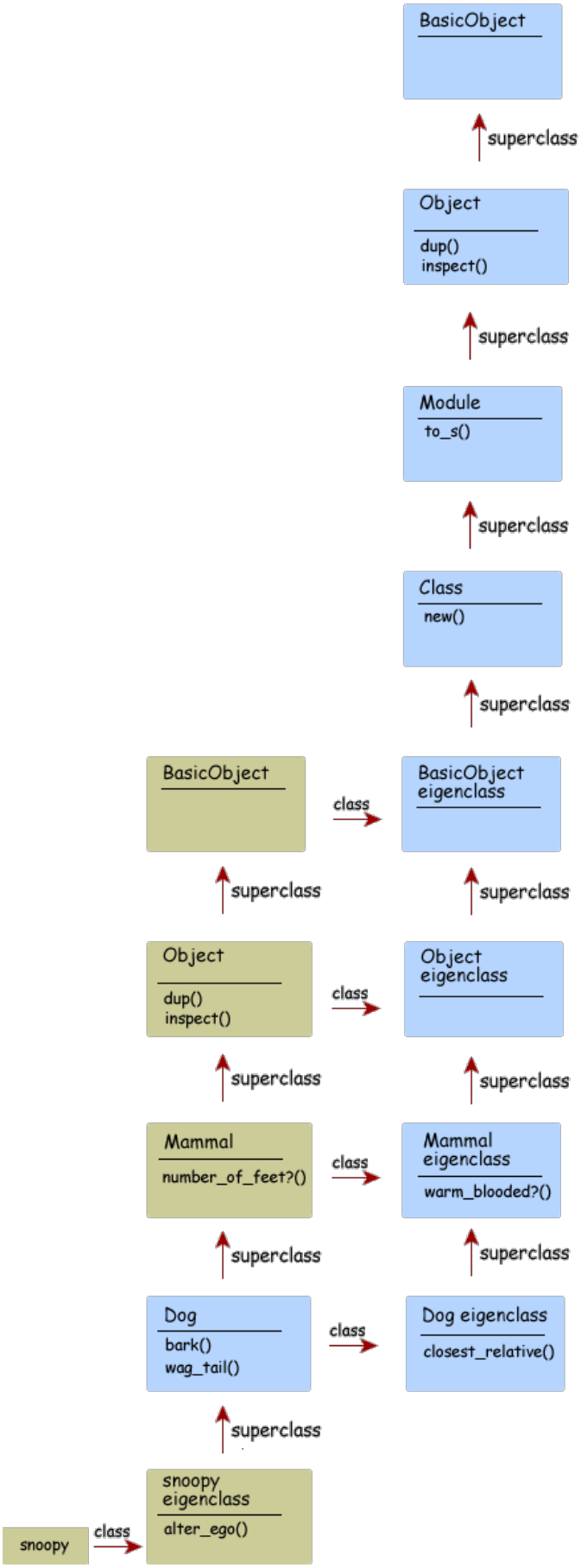
This is another puzzler. How is it that classes manage to inherit the class methods of their superclasses? After all, the superclass of a class object is in the method lookup path of its instances but *not* in its *own* method lookup path:

Method lookup path of a Dog instance: fido --> Dog --> Mammal --> Object ...

Method lookup path of a Dog Object: Dog --> Class --> Module --> Object...

Here again some fancy footwork is done to ensure that the method lookup path pattern remains unaltered and yet allow access to the class methods of the superclass.

First off, when a class (aka singleton) method is defined on Dog, an eigenclass is created. For all intents and purposes, this eigenclass becomes the class of the Dog object in lieu of Class. Class gets "pushed up" the lookup chain and becomes a superclass. Now, when a class method is defined on the superclass of Dog (e.g, Mammal), the resulting eigenclass is designated as the immediate superclass of Dog's own eigenclass and Class gets pushed up even higher. As this description is probably as clear as mud, hopefully the following diagram will shed some light on the matter:


BasicObject
superclass
Object
dup()
inspect()
superclass
Module
to_s()
superclass
Class
new()
superclass
BasicObject
class
BasicObject eigenclass
superclass
superclass
Object
dup()
inspect()
class
Object eigenclass
superclass
superclass
Mammal
number_of_feet?()
class
Mammal eigenclass
warm_blooded?()
superclass
superclass
Dog
bark()
wag_tail()
class
Dog eigenclass
closest_relative()
superclass
snoopy eigenclass
alter_ego()
snoopy
class

So there you have it. The Ruby object method lookup path in all its glory. Well almost. We haven't looked at where module mixins fit into the picture. Hopefully I'll have the time to cover that in a later blogpost.

Resources

1. The Ruby Object Model and Metaprogramming, Screencasts by Dave Thomas
2. Metaprogramming Ruby by Paolo Perrotta

16 Comments Made by DNA 1 Login

Recommend 18 Share Sort by Best

Join the discussion…

banisterfiend · 4 years ago

Very good and very readable! But just a few things :) 1. In MRI class singletons are not created lazily, they're actually created when the class is created. 2. You leave out the classes of the eigenclasses (perhaps intentionally? :). It's probably overkill, but if you're interested in seeing a 100% complete Method lookup diagram for ruby 1.8.7 check here: http://banisterfiend.wordpress...

I didn't bother to update it for ruby 1.9 as it's pretty much the same, except for the addition of BasicObject.

Nice work on the post, very clear.

4 · Reply · Share ›

john smith ↗ banisterfiend · a year ago

banisterfiend, I'm confused. Dave Thomas, during a talk he gave in 2009 [1] (at about 29:30), says that singleton classes are created lazily. And after some quick google-fu it would seem others concur [2].

Have I misunderstood your comment in some way, or is there something I'm not taking into consideration?

Thanks!

[1] http://scotland-on-rails.s3.am...

[2] https://charlie.bz/blog/why-do...

· Reply · Share ›

banisterfiend ↗ john smith · a year ago

i said 'class singletons', that is the singletons specifically for classes, not standard objects. 'class singletons' are not created lazily, they're built at class boot time :)

1 · Reply · Share ›

Prasanna N · 3 years ago

now I know the 'class <<' syntax better! Thanks.

1 · Reply · Share ›

mitya777 · 3 months ago

I took a shot at making a visualization to get a better grasp of the ruby object model and class inheritance hierarchy: https://mitya777.github.io/dev...

· Reply · Share ›

Ron Jeffries · 6 months ago

Lovely article! Makes a difficult subject clear.

· Reply · Share ›