Demystifying Ruby Singleton Classes

March 15, 2018

One of the most mysterious aspects of the Ruby object model is the existence of singleton classes. Maybe you also read about metaclasses or eigenclasses, they are actually all referring to the same thing. Still, the "official" name being singleton classes, I will use this term through this post.

You actually don't need to understand much of singleton classes (or to even know them at all) to write decent Ruby code. Though, as they form an essential part of the Ruby object model, it is an interesting knowledge to have.

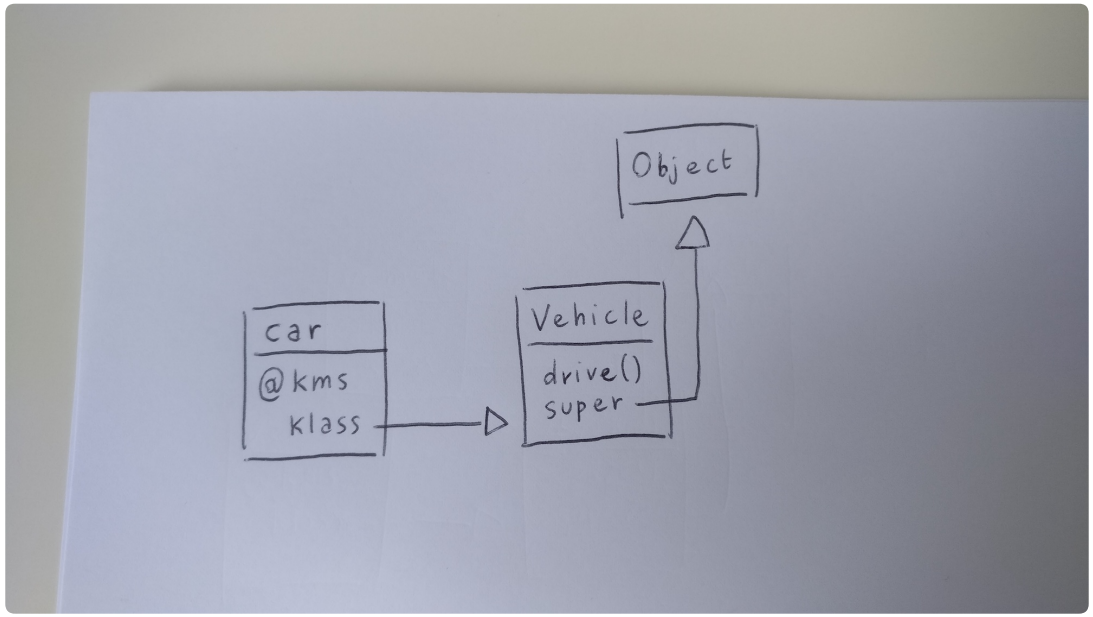
The method dispatch problem

First, to fully understand the purpose of such classes, we have to look to another essential piece of Ruby: method dispatching. Consider the following code:

```ruby
class Vehicle
  def initialize(kms)
    @kms = kms
  end
  def drive
    puts "let's go!"
  end
end
car = Vehicle.new(20_000)
car.drive
```

Nothing fancy here. We define a class Vehicle with a method `drive` and then create a new instance of it. Let's dig into what these few lines of code actually do behind the scenes.

Internally, here is how Ruby is representing the class `Vehicle` and its instance `car` in memory:



The `car` object is represented by Ruby using a C struct, holding the list of its instance variables and a pointer `klass` to the its class. The class `Vehicle` itself holds its list of methods in and a `super` attribute pointing at its superclass (in our case, `Object`, the default superclass for all classes).

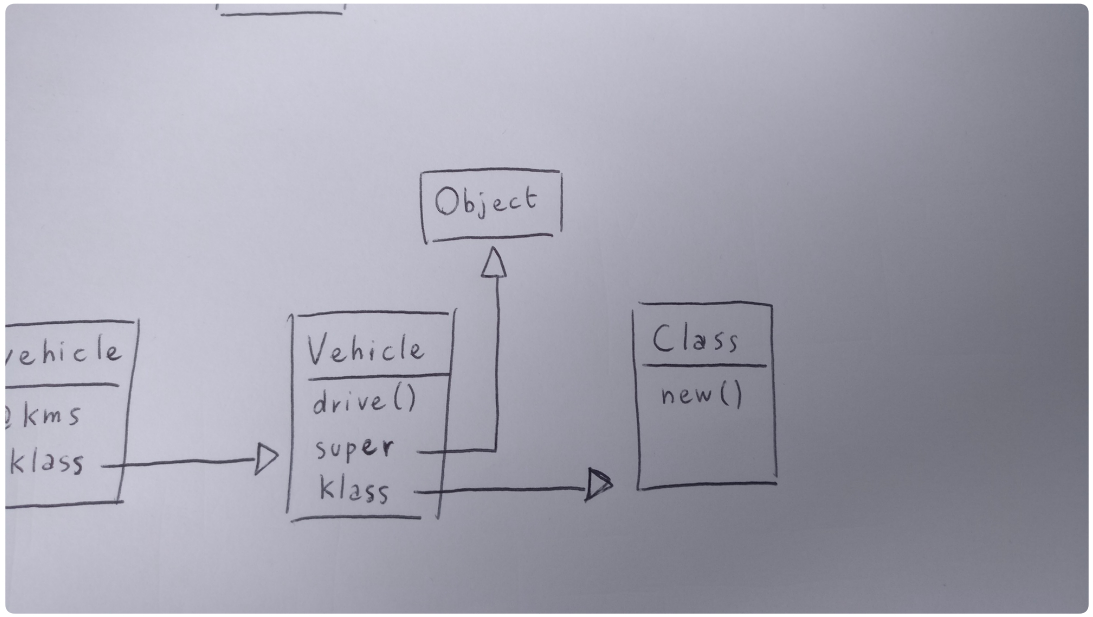
As we can see, the `car` object has actually no idea that it has a method named `drive`. All it knows is that it owns an instance variable named `kms`. When we call the method `drive` on the object, Ruby will actually follow the `klass` pointer of the underlying C struct to find the object's class. In the method list of this class, it will find the method `drive` and then call it on the current value of `self`, which is `car`.

As classes are also objects in Ruby, a similar strategy is used to resolve a method called on a class. Let's go back to the previous example and add some code:

```ruby
class Vehicle
  @@registry = []

  def self.register(vehicle)
    @@registry << vehicle
  end

  # ...
end

car = Vehicle.new(20_000)
Vehicle.register(car)
```

We just defined the class method `register` on our `Vehicle` class to manage a simple list of registered vehicles. Good. As we said, method dispatch works in the exact same way for classes than for objects. That means, calling `Vehicle.register` should work as we saw before, but with the `Vehicle` class instead of its instance.

Here is a schema similar to the one we saw earlier, showing the `Vehicle` class:



So, our class `Vehicle` owns an instance variable `registry` and its `klass` pointer is referencing the class of all classes, `Class` (which defines the class method `new` for example).

But then, where is the class method `register`? It can't be into the method list `Vehicle`. As we saw before, due to the way the Ruby object model works, it can only contains instance methods.

So, since method dispatch must work the same as for all other objects, Ruby should follow the `klass` pointer of `Vehicle` to find the method. However, this would mean that the `register` method is defined as a method of `Class`, which would mean *all* of our

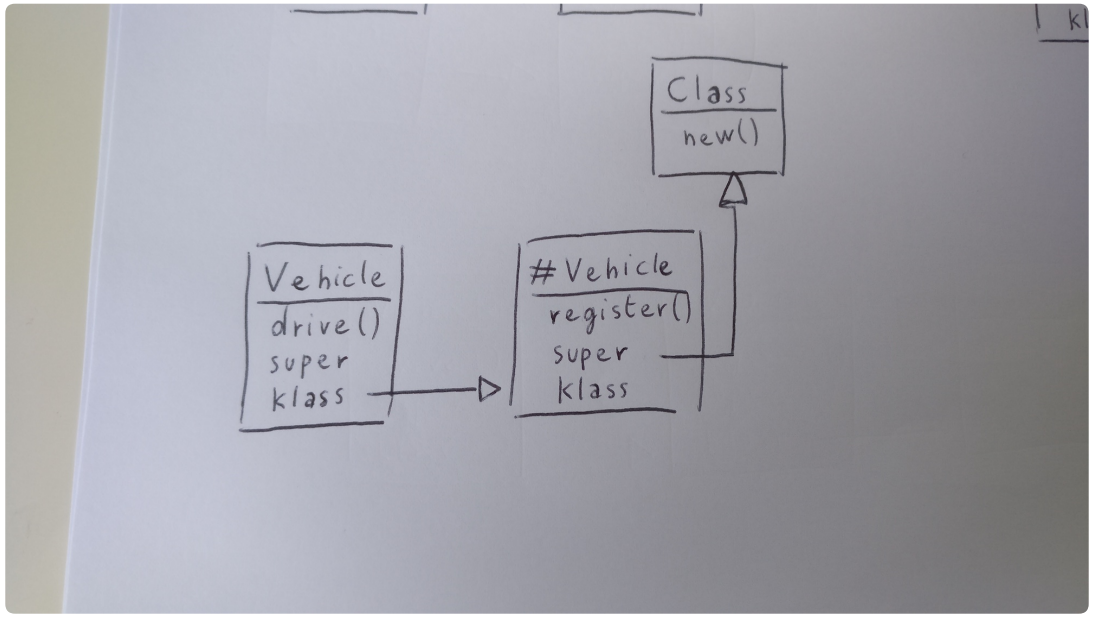
classes would get this method. So, neither of these solutions can work.

Ruby solves this problem using singleton classes.

Discovering singleton classes

A singleton class of an object (or a class) is a class created by Ruby only for this specific object. This class is somehow "hidden" to us, but it is there. When calling a method on this object, Ruby will look first into its singleton class, if there is one, to find that method.

For the rest of this post, we will refer to the singleton class of a class `A` as `#A` (and to the one of an object `a` as `#a`). So, taking back our example of the `Vehicle` class, here is how things look when we define the `register` class method:



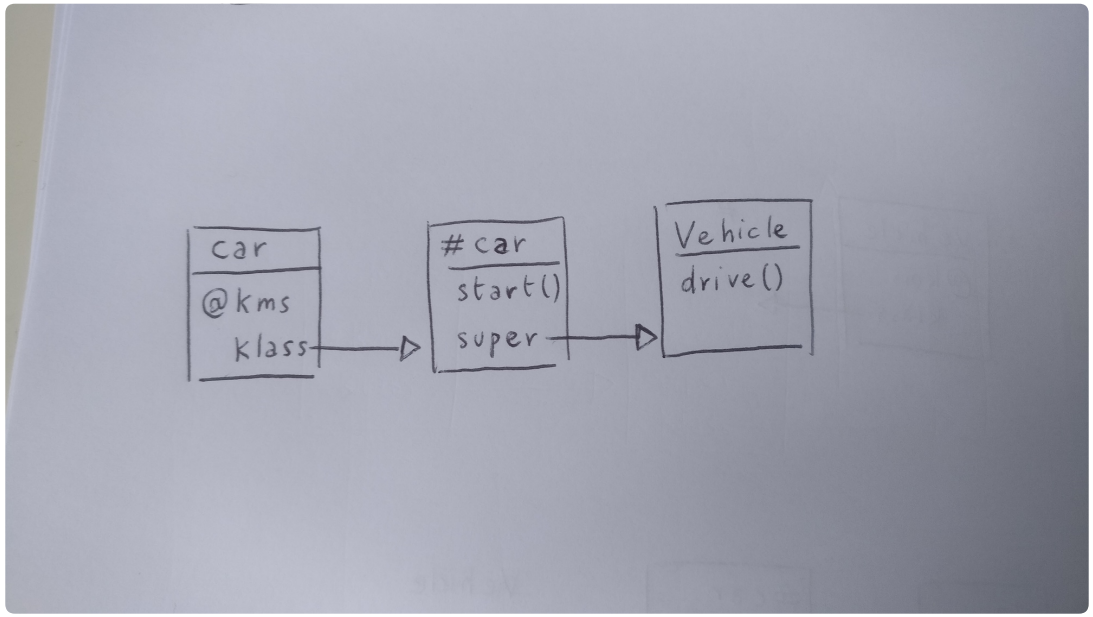
The schema above is actually slightly simplified, as we will see in a few minutes. But for now let's focus of what is happening here. Writing `def self.register`, we actually tell Ruby to open the singleton class of `self` (which is then pointing to the `Vehicle` class) and define the `register` method on it.

Now, when we call `Vehicle.register`, Ruby will look first into `#Vehicle` instead of `Class`, to find the `register` method, calling it with the value of `self` being the class `Vehicle`. Since the `super` of `#Vehicle` is pointing to `Class`, Ruby can still follow the chain to find any other class method like `new`. We are back on our feet.

Singleton classes can be defined in the exact same way for "regular" objects. We could then define a method only on a specific `Vehicle` instance:

```ruby
car = Vehicle.new(20_000)

def car.start
  puts "starting..."
end
```

We open the singleton class of `car` and define a `start` method on it. This method is then only available for this specific object. Again, we can draw a similar schema showing the relation between `car`, its singleton class and the `Vehicle` class.



Move along, no class here

Remember when we said that singleton classes where hidden to us. For example, calling `class` on the object will not return its singleton class `#car` as we could expect, but its class `Vehicle`. That is because internally, Ruby uses a special flag on the singleton class. Also, it cannot be instantiated like a regular class.

However, as the `super` attribute of `#car` is pointing to the original class `Vehicle`, it is possible to override methods defined by `Vehicle` calling `super`:

```ruby
car = Vehicle.new(1000)
bus = Vehicle.new(3000)
```

```ruby
def car.drive
  print "I'm driving a car! "
  super
end

car.drive # "I'm driving a car! let's go!"
bus.drive # "let's go!"
```

We can have access to the singleton class by calling the `singleton_class` method on the `car` object. By inspecting the singleton class, we can see Ruby names it using the following pattern: `#<Class:<original_object>>`

```ruby
> car.inspect
=> "#<Vehicle:0x007fd4442c38f8>"
> car.singleton_class.inspect
=> "#<Class:#<Vehicle:0x007fd4442c38f8>>"
```

This is also true for the singleton class of a class:

```ruby
> Vehicle.inspect
=> "Vehicle"
> Vehicle.singleton_class.inspect
=> "#<Class:Vehicle>"
```

The truth about Singleton Classes of Classes

So far, we learned that the class method `register` is actually defined on the singleton class of `Vehicle`, `#Vehicle`. But now let's take a slightly more complicated example:

```ruby
class Vehicle
  @@registry = []

  def self.register(vehicle)
    @@registry << vehicle
  end
end

class Car < Vehicle
  def self.register(car)
    # register car plate
    super
  end
end

some_car = Car.new
Car.register(some_car)
```

We created the class `Car` which is a subclass of `Vehicle`. We define as well a class method `register` on `Car` that does some car-specific work and calls the `register` method on the parent class `Vehicle` using `super`. If we take the time to draw again a schema using what we know, here how it looks:

There is something that we miss here. The `Car` class has a class method `register` on its singleton class `#Car` that clearly calls the `register` method of its superclass, `Vehicle`. But we saw that this method is itself defined on `#Vehicle`. If `#Car` and `#Vehicle` have no relationship, how can Ruby get back on his feet to find the `register` method of the superclass?

That's where the magic of the Ruby object model reveals itself. Here is the ultimate schema where we can see what are the real relationships between all the elements we looked at so far:

Take some time to analyze and understand this one, because for sure, it can be quite complicated and weird to get the first time. What happens is that, for the singleton class `#Car` to be able to call the superclass's method `register` defined on the singleton class `#Vehicle`, Ruby actually make the `super` pointer of `#Car` points to `#Vehicle` instead of class. That's why we are able to call a superclass class method from a subclass class method. In the end, even the `klass` pointer of the `Object` class points to `#Object` (that's why can also define class methods on the `Object` class).

Finally, `#Object`'s `super` pointer goes back to `Class`. So, as all Ruby classes inherit from `Object`, all of their singleton classes inherits from `#Object` and ultimately from `Class`.

From these last learnings, we can infer two rules that are part of the core principles of the entire Ruby object model:

> The superclass of the singleton class of an object is the object's class.

And

> The superclass of the singleton class of a class is the singleton class of the class's superclass.

Yep. Read it again a second time and a third time, just in case.

Alternative syntaxes

There are other ways to define methods on the singleton class of an object. We can use the special `class <<` syntax to directly open the singleton class and define methods on it:

```ruby
car = Vehicle.new(2000)

class << car
  def start
    "starting..."
  end
end
```

Again, this is also true for classes, so you can also use this syntax to define class methods (you can see it a lot in some frameworks and libraries like Rails):

```ruby
class Vehicle
  class << self
    def register(vehicle)
      @@vehicles << vehicle
    end
  end
end
```

Another way of defining methods on the singleton class is to use a module and the `extend` method. You may have already used `extend`. What it actually does is opening the singleton class of the receiver object and add methods from the module passed in argument.

The Next Programming Language
You Should Learn ⇢