Eigenclasses, Singletons, What?

Christopher Mark Gore

cgore.com

Monday, June 8, AD 2015

Everything is a Thing

```ruby
12.class # Fixnum
"Hi there".class # String
2.71828.class # Float
[1, 2, 3].class # Array
{age: 36, name: "Chris"}.class # Hash
:foo.class # Symbol
```

We Can Insist on the "Right" Thing

We'll explain the difference between the concepts "instance of," "is a," and "kind of" in a few slides.

```
1  if not k.class == Fixnum
2    raise ILoveOnlyNumbers
3  end
4  if not m.instance_of? Fixnum
5    raise ILoveOnlyNumbers
6  end
7  if not n.is_a? Fixnum
8    raise ILoveOnlyNumbers
9  end
10 if not o.kind_of? Fixnum
11   raise ILoveOnlyNumbers
12 end
```

We Can Do Different Things to Different Things

```
1 if n.class == Fixnum
2   puts "Whole␣Numbers␣FTW!!!"
3 elsif n.class == Float
4   puts "More␣Accuracy␣is␣MORE!!!"
5 else
6   raise ILoveOnlyNumbers
7 end
```

Even Classes are Things!

This is one of the more unique aspects of Ruby. In a lot of other object-oriented programming languages, classes are "special" somehow, and you can't treat them the same way as everything else.

```
1 :foo.class # Symbol
2 :foo.class.class # Class
3 :foo.class.class.class # Class
```

We'll see how that's actually really nice and useful in a bit.

We Can Make Our Own Classes

```ruby
class Cat
  def initialize color
    @color = color
    @lives = 9
  end
  def die
    @lives = @lives - 1
  end
  def alive?
    @lives >= 1
  end
  def eat_tuna
    puts "Purrr!"
  end
end
lucky = Cat.new "black"
lucky.alive? # true
```

Classes Have Methods

And those methods let our class instances do things.

```
1  class ATM
2    def initialize amount
3      @amount = amount
4    end
5    def deposit account, amount
6      @amount = @amount + amount
7      account.deposit amount
8    end
9    def withdraw accout, amount
10     raise GoAwayPeasant if amount > account.amount
11     raise RunOnTheBank if amount > @amount
12     account.withdraw amount
13     @amount = @amount - amount
14   end
15 end
```

Instance Variables

Instance variables are written with an at-sign like `@foo`, and are per-instance.

```
1  class Person
2    attr_reader :age, name
3    def initialize(age, name)
4      @age = age
5      @name = name
6    end
7  end
8  john = Person.new 12, "John␣Jacobson"
9  fred = Person.new 44, "Fred␣Fellows"
10 john.age # 12
11 fred.age # 44
12 john == fred # false
```

Getters and Setters

```ruby
class Car
  attr_accessor :color
  attr_reader :mileage
  def initialize(color, mileage)
    @color, @mileage = color, mileage
  end
  def drive miles
    @mileage = @mileage + miles
  end
end
c = Car.new 4, :blue
c.drive 92_000
c.mileage # 92004
c.color = :green
c.mileage = 123 # NoMethodError: undefined method `mileage='
```

Class Variables

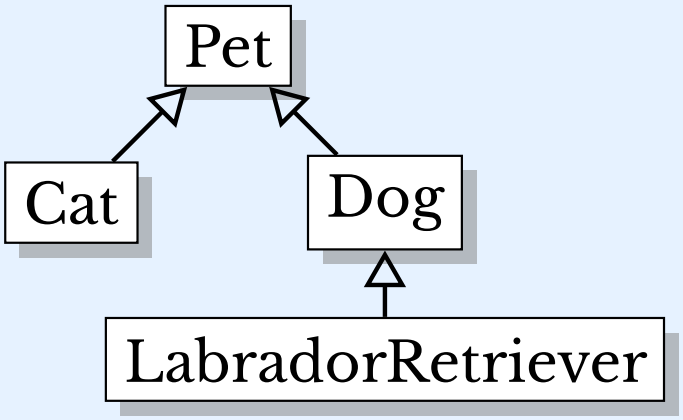
Class variables are written with two at-signs like `@@foo`, and are per-class. They are almost never what you actually want.

```
1  class People
2    @@population = 0
3    attr_reader :name, :age
4    def initialize(name, age)
5      @@population = @@population + 1
6      @alive = true
7      @name, @age = name, age
8    end
9    def die
10     @alive = false
11     @@population = @@population - 1
12   end
13 end
```

Class Methods

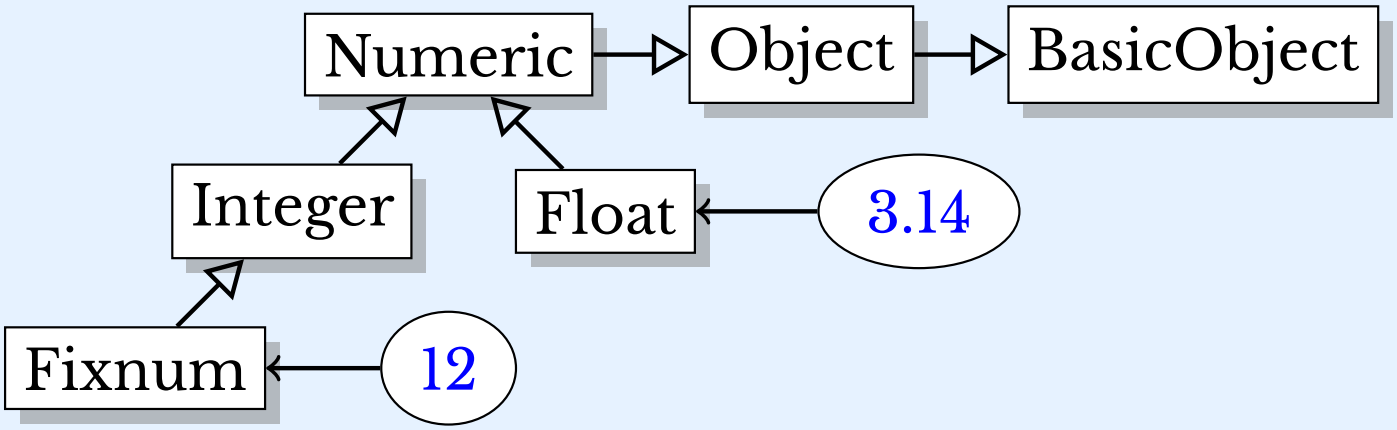
```
1  class Person
2    attr_reader :name, :born, :died
3    def initialize(name, born, died)
4      @name, @born, @died = name, born, died
5    end
6    class << self
7      def from_s s # String like "John Doe (1912 - 1990)"
8        name, born, died = s.scan(/(.*)\((\d+) - (\d+)\)/)[0]
9        Person.new name, born.to_i, died.to_i
10     end
11   end
12 end
13 george = Person.from_s "George␣Washington␣(1732␣-␣1739)"
14 ben = Person.from_s "Benjamin␣Franklin␣(1706␣-␣1790)"
```

Inheritance



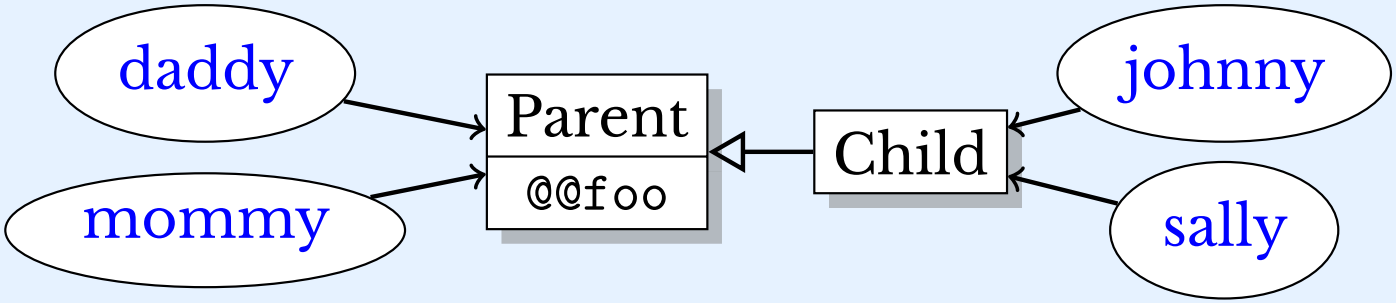

```
1 class Pet
2 end
3 class Cat < Pet
4 end
5 class Dog < Pet
6 end
7 class LabradorRetriever < Dog
8 end
```

Kind-Of, Is-A, and Instance-Of




```
1 12.class # Fixnum
2 3.14.class # Float
3 12.is_a? Numeric # true
4 12.is_a? Integer # true
5 12.is_a? Fixnum # true
6 12.kind_of? Numeric # true
7 12.instance_of? Numeric # false
8 12.instance_of? Integer # false
9 12.instance_of? Fixnum # true
```

Class Variables Are Shared With Subclasses




```
1  class Parent
2    @@foo = []
3    def foo
4      @@foo
5    end
6    def self.foo
7      @@foo
8    end
9  end
10
11 class Child < Parent
12 end
```

Class Variables Are Shared With Subclasses

```
1  daddy, mommy = Parent.new, Parent.new
2  johnny, sally = Child.new, Child.new
3  Parent.foo << 1
4  Child.foo << 2
5  daddy << 3
6  johnny << 4
7  Parent.foo # [1, 2, 3, 4]
8  Child.foo # [1, 2, 3, 4]
9  daddy.foo # [1, 2, 3, 4]
10 mommy.foo # [1, 2, 3, 4]
11 johnny.foo # [1, 2, 3, 4]
12 sally.foo # [1, 2, 3, 4]
```

Class Instance Variables Are Not Shared

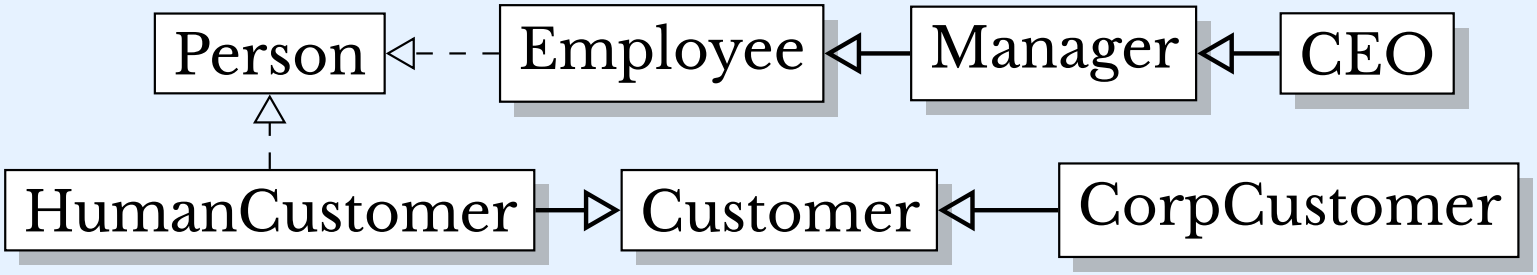
```
1  class A
2    class << self
3      attr_accessor :x
4    end
5  end
6  class B < A
7  end
8  A.x = :aay
9  B.x = :buzz
10 A.x # :aay
11 B.x # :buzz
12 a = A.new
13 b = B.new
14 a.class.x # :aay
15 b.class.x # :buzz
```

Modules

Modules can't be instantiated. They're like classes, but more for libraries of functions.

```
1  module MathStuff
2    BETTER_PI = 3.15
3    class << self
4      def addition x,y # I love typing
5        x+y
6      end
7      def subtraction x,y
8        x-y
9      end
10   end
11 end
12
13 MathStuff.addition 12,44 # 56
14 m = MathStuff.new # NoMethodError: undefined method `new'
```

Mixins Are Modules Stuffed Into Classes




```
1  module Person
2    attr_accessor :name, :age, :favorite_color
3  end
4  class Employee
5    include Person
6  end
7  class Manager < Employee
8  end
9  class Customer
10   include Person
11 end
```

Singletons

You almost never actually want one of these.

```
1  require "singleton"
2
3  class ThereCanBeOnlyOne
4    include Singleton
5    attr_accessor :name
6  end
7
8  connor = ThereCanBeOnlyOne.instance # You can't call new.
9  connor.name = "Connor␣MacLeod"
10 duncan = ThereCanBeOnlyOne.instance
11 duncan.name = "Duncan␣MacLeod"
12 connor.name # "Duncan MacLeod"
13 connor == duncan # true
```

Eigenclasses

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

class A
end

a, b = A.new, A.new
a.eigenclass != b.eigenclass # true
a.eigenclass.class # Class
a.eigenclass.superclass # A
```

Instance Methods

```
1  class C
2    def initialize x
3      @x = x
4    end
5    def f
6      puts @x
7    end
8  end
9
10 c = C.new "default␣stuff"
11 d = C.new "wat?"
12 def d.f
13   puts @x.reverse
14 end
15 c.f # "default stuff"
16 d.f # "?taw"
```

Metaclasses

Instance methods *actually* live in the metaclass.

```
1  class C
2    def initialize x
3      @x = x
4    end
5    def f
6      puts @x
7    end
8  end
9  c, d = C.new(13), C.new(13)
10 def c.f
11   "lucky"
12 end
13 c.to_s # "lucky"
14 d.to_s # "13"
15 metaclass = class << c; self; end
```

Questions?