Practicing Ruby

I decided to start off this newsletter with one of the most basic but essential pieces of knowledge you can have about Ruby's object model: the way it looks up methods. Let's do a little exploration by working through a few examples.

Below we have a simple report class tasked with performing some basic data manipulations and then producing some text output.

```
class Report
  def initialize(ledger)
    @balance          = ledger.inject(0) { |sum, (k,v)| sum + v }
    @credits, @debits = ledger.partition { |k,v| v > 0 }
  end

  attr_reader :credits, :debits, :balance

  def formatted_output
    "Current Balance: #{balance}\n\n" +
    "Credits:\n\n#{formatted_line_items(credits)}\n\n" +
    "Debits:\n\n#{formatted_line_items(debits)}"
  end

  def formatted_line_items(items)
    items.map { |k, v| "#{k}: #{'%.2f' % v.abs}" }.join("\n")
  end
end
```

The following example demonstrates how we'd make use of this class.

```
ledger = [ ["Deposit Check #123", 500.15],
           ["Fancy Shoes",       -200.25],
           ["Fancy Hat",          -54.40],
           ["ATM Deposit",       1200.00],
           ["Kitteh Litteh",       -5.00] ]

report = Report.new(ledger)
puts report.formatted_output
```

And for those who don't want to take the time to copy and paste this code and run it locally, the actual output is shown below.

```
Current Balance: 1440.5

Credits:

Deposit Check #123: 500.15
ATM Deposit: 1200.00

Debits:

Fancy Shoes: 200.25
Fancy Hat: 54.40
Kitteh Litteh: 5.00
```

While not particularly pretty, this report is mostly what we'd expect to see. You can probably imagine how this information might be embedded within another report, such as an email-based form letter with some header and footer information. One possible way to do this would be through class inheritance, as in the example below.

```
require "date"

class EmailReport < Report
  def header
    "Dear Valued Customer,\n\n"+
    "This report shows your account activity as of #{Date.today}\n"
  end

  def banner
    "\n.....................................................\n"
  end

  def formatted_output
    header + banner + super + banner + footer
  end

  def footer
    "\nWith Much Love,\nYour Faceless Banking Institution"
  end
end
```

We only need to make a minor change to our calling code to make use of this new class.

```
ledger = [ ["Deposit Check #123", 500.15],
           ["Fancy Shoes",       -200.25],
           ["Fancy Hat",          -54.40],
           ["ATM Deposit",       1200.00],
```

```
           ["Kitteh Litteh",       -5.00] ]

report = EmailReport.new(ledger)
puts report.formatted_output
```

Below you can see what the new output ends up looking like.

```
Dear Valued Customer,

The following report shows your account activity as of 2010-11-09

.....................................................
Current Balance: 1440.5

Credits:

Deposit Check #123: 500.15
ATM Deposit: 1200.00

Debits:

Fancy Shoes: 200.25
Fancy Hat: 54.40
Kitteh Litteh: 5.00
.....................................................

With Much Love,
Your Faceless Banking Institution
```

Looking back at the EmailReport code, it's easy to see what we've done to produce this new output. We've defined a new formatted_output method which adds the headers and footers, and combined this new behavior with the original behavior of our Report class by calling super. This is the same extension by inheritance pattern that you'll learn in any basic computer science course or encounter in any of the reasonably traditional object oriented languages out there.

But before you go asking for a refund and start telling your friends that this newsletter is painfully dull, consider this: While many languages have a method lookup path which is based on inheritance alone, that isn't even close to being true about Ruby.

Because Ruby allows for module mixins and per-object behavior, the superkeyword takes on a whole new life in which an object's superclass is the last stop on a five part journey through Ruby's object model. The following example proves the point by composing a simple string which demonstrates the order in which methods are resolved in Ruby.

```
module W
```

```
  def foo
    "- Mixed in method defined by W\n" + super
  end
end

module X
  def foo
    "- Mixed in method defined by X\n" + super
  end
end

module Y
  def foo
    "- Mixed in method defined by Y\n" + super
  end
end

module Z
  def foo
    "- Mixed in method defined by Z\n" + super
  end
end

class A
  def foo
    "- Instance method defined by A\n"
  end
end

class B < A
  include W
  include X

  def foo
    "- Instance method defined by B\n" + super
  end
end

object = B.new
object.extend(Y)
object.extend(Z)

def object.foo
  "- Method defined directly on an instance of B\n" + super
end
```

```
puts object.foo
```

When we run this code, we see the following output, which traces the supercalls all the way up from the method defined directly on our object to its superclass.

```
- Method defined directly on an instance of B
- Mixed in method defined by Z
- Mixed in method defined by Y
- Instance method defined by B
- Mixed in method defined by X
- Mixed in method defined by W
- Instance method defined by A
```

As promised, it's a five step journey. Particularly, the above is a demonstration that Ruby methods are looked up in the following order:

1. Methods defined in the object's singleton class (i.e. the object itself)
2. Modules mixed into the singleton class in reverse order of inclusion
3. Methods defined by the object's class
4. Modules included into the object's class in reverse order of inclusion
5. Methods defined by the object's superclass.

This process is then repeated all the way up the inheritance chain until BasicObjectis reached. Now that we know the basic order, we should stop and consider a few questions about what we've discussed so far.

Open Questions / Things To Explore {#open-questions--things-to-explore}

- Why would we want or need five distinct places to define methods? Do these other options really gain us anything over ordinary inheritance?
- Does this change the way that classic object oriented design principles apply to Ruby? For example, how well do you think direct translations of design patterns map to Ruby?
- Think of each place you can define a method in Ruby, and consider which ones are important for every day use, and which ones are edge cases. Is per-object behavior really that useful?
- It is rare to use all of these options at once, and the only reason it was done in this exercise was for demonstration purposes. But taken individually, can you think of a practical use for each way of defining Ruby methods?
- What are some disadvantages for each technique shown here?

I will address these points and also go over some practical applications in the next issue, but please share your own thoughts in the comments section below.

> **NOTE:**This article has also been published on the Ruby Best Practices blog. There may be additional commentaryover there worth taking a look at.