CURING YOUR PRIMITIVE OBSESSION

WITH HELP FROM ERIC ROBERTS | @eroberts

PRIMITIVE OBSESSION

WHAT IS A PRIMITIVE?

In computing, language primitives are the simplest elements available in a programming language. A primitive is the smallest 'unit of processing' available to a programmer of a particular machine, or can be an atomic element of an expression in a language.

– Wikipedia

IN RUBY

```
boolean = true
number  = 1
array   = [1,2,3,4,5]
range   = 1..3
hash    = { foo: :bar, baz: :qux }
```

OBSESSION

"the state of being obsessed with someone or something."

REALLY, GOOGLE?

"an idea or thought that continually preoccupies or intrudes on a person's mind."

PRIMITIVE OBSESSION

EVERY ARRAY YOU MAKE

Primitive obsession is the practice of using primitives where specialized objects would be more appropriate. For example, using a string to represent a URL or Postal Code.

"N1H 7H8"

```
"N1H 7H8".valid?          #=> Uh... what?
```

```ruby
class PostalCode < Struct.new(:code)
  def valid?
    # some code that ensures validity
  end

  def to_s
    code
  end

  def postal_district
    code[0]
  end

  def forward_sortation_area
    code[0..3]
  end

  def local_delivery_unit
    code[3..6]
  end
end
```

```
postal_code = PostalCode.new("N1H 7H8")
postal_code.to_s                    #=> "N1H 7H8"
postal_code.postal_district         #=> "N"
postal_code.forward_sortation_area  #=> "N1H"
postal_code.local_delivery_unit     #=> "7H8"
```

SO, WHAT ABOUT THESE PERCENT THINGS?

```ruby
# Taken from ActionView::Helpers::NumberHelper

number_to_percentage(100)                                    #=> 100.000%
number_to_percentage("98")                                   #=> 98.000%
number_to_percentage(100, precision: 0)                      #=> 100%
number_to_percentage(1000, delimiter: '.', separator: ',')   #=> 1.000,000%
number_to_percentage(302.24398923423, precision: 5)          #=> 302.24399%
number_to_percentage(1000, locale: :fr)                      #=> 1 000,000%
number_to_percentage("98a")                                  #=> 98a%
number_to_percentage(100, format: "%n  %")                   #=> 100  %

number_to_percentage("98a", raise: true)                     #=> InvalidNumberError
```

THIS HELPS WITH FORMATTING
BUT NOT MUCH ELSE.

WHAT ELSE COULD YOU DO?

```ruby
# Something like...
50.percent(10)    #=> 5
```

MONKEY PATCHING TO THE RESCUE!

```ruby
class Numeric
  def percent(p)
    p.to_f / self.to_f * 100.0
  end
end

100.percent(10)     #=> 10
```

OR NOT...

SO, WHAT DO WE DO?

WE NEED AN OBJECT

```ruby
class Percent
  def initialize(value)
    @value = value
  end
end
```

WHAT SHOULD IT DO?

```ruby
percent = Percent.new(50)
percent.value   #=> 50.0
percent.to_s    #=> '50%'
percent.to_f    #=> 0.5
percent == 50   #=> false
percent == 0.5  #=> true
```

```
attr_reader :value
```

```ruby
def to_s
  '%g%%' % value
end

percent = Percent.new(50)
percent.to_s #=> "50%"
```

```
def to_f
  value/100
end

percent = Percent.new(85)
percent.to_f #=> 0
```

WAIT, WHAT?

```
85/100   = 0
85.0/100 = 0.85
```

```ruby
def initialize(value)
  @value = value.to_f
end

percent = Percent.new(85)
percent.to_f #=> 0.85
```

```
def == other
  (other.class == class && other.value == value) ||
    other == to_f
end

Percent.new(20) == Percent.new(20)    #=> true
Percent.new(20) == 0.2                #=> true
```

```ruby
def eql? other
  self == other
end
```

```ruby
percent = Percent.new(50)
percent.value   #=> 50.0
percent.to_s    #=> '50%'
percent.to_f    #=> 0.5
percent == 50   #=> false
percent == 0.5  #=> true
```

```
bigger = Percent.new(90)
smaller = Percent.new(10)

bigger > smaller  #=> true
smaller < bigger  #=> true
```

```ruby
def <=> other
  to_f <=> other.to_f
end
```

```
percent = Percent.new(10)
percent + percent    #=> Percent.new(20)
percent - percent    #=> Percent.new(0)
percent * percent    #=> Percent.new(1)
percent / percent    #=> 1
```

```ruby
def + other
  self.class.new(value + other.value)
end

percent = Percent.new(10)
percent + percent #=> Percent.new(20)
```

```ruby
def - other
  self.class.new(value - other.value)
end

percent = Percent.new(10)
percent - percent #=> Percent.new(0)
```

```ruby
def * other
  self.class.new(to_f * other.value)
end

percent = Percent.new(10)
percent * percent #=> Percent.new(1)
```

```ruby
def / other
  self.class.new(value / other.value)
end

percent = Percent.new(10)
percent / percent #=> 1
```

OK, BUT THAT'S NOT ALL THAT INTERESTING

```
percent = Percent.new(50)
percent + 10     #=> Percent.new(60)
percent - 10     #=> Percent.new(40)
percent * 10     #=> Percent.new(500)
percent / 10     #=> Percent.new(5)
```

WE'RE GOING TO FOCUS ON JUST ONE METHOD FOR NOW

```ruby
def * other
  case other
  when Percent
    self.class.new(to_f * other.value)
  when Numeric
    self.class.new(value * other)
  end
end

percent = Percent.new(50)
percent * percent #=> Percent.new(25)
percent * 10      #=> Percent.new(500)
```

WHAT ABOUT THE OTHER WAY AROUND?

```
percent = Percent.new(50)
10 + percent     #=> 15
10 - percent     #=> 5
10 * percent     #=> 5
10 / percent     #=> 20
```

```
percent = Percent.new(50)
10 * percent    #=> 5
```

NOW THINGS START TO GET INTERESTING

#COERCE

```
percent = Percent.new(50)
1 * percent
```

ANY GUESSES AS TO WHAT HAPPENS HERE?

```ruby
1 * percent

# percent will receive the coerce message
# with the number we are trying to multiply by

percent.coerce(1)
```

```
TypeError: Percent can't be coerced into Fixnum
```

```ruby
class Percent
  def coerce other
    [other, to_f]
  end
end
```

```ruby
percent = Percent.new(50)

10 * percent
percent.coerce(1)  #=> [10, 0.5]
10 * 0.5            #=> 5
```

LET'S REVIEW...

```ruby
class Percent
  [...]

  def * other
    case other
    when Percent
      self.class.new(to_f * other.value)
    when Numeric
      self.class.new(value * other)
    end
  end

  def coerce other
    [other, to_f]
  end
end
```

WHAT IF "OTHER" IS NOT NUMERIC?

```ruby
percent = Percent.new(50)
money = Money.new(100)

# What we want to happen
percent * money  #=> Money.new(50)
```

```
# What actually happens
percent * money  #=> nil
```

```ruby
def * other
  case other
  when Percent
    self.class.new(to_f * other.value)
  when Numeric
    self.class.new(value * other)
  end
end
```

```ruby
def * other
  case other
  when Percent
    self.class.new(to_f * other.value)
  when Numeric
    self.class.new(value * other)
  when Money
    other * to_f
  end
end
```

#COERCE TO THE RESCUE

```ruby
def * other
  if other.is_a? Percent
    self.class.new(to_f * other.value)
  elsif other.respond_to? :coerce
    a, b = other.coerce(self)
    a * b
  else
    raise TypeError, "#{other.class} can't be coerced into Percent."
  end
end
```

```
percent = Percent.new(50)
money = Money.new(100)

percent * money
```

```
percent * money
money.coerce(percent)

# Money receive :coerce with the percent,
# and returns the same things in opposite order
[money, percent]

# Then we try the operation again
money * percent
percent.coerce(money)

# Now, percent receives coerce, with money,
# and returns two more things, this time with
# the percent changed to float
[money, float]

# Finally, we can perform this operation
# without more coercion
money * float

Money.new(100) * 0.5  #=> Money.new(50)
```

```ruby
class Money
  def coerce(other)
    [self, other]
  end
end
```

```
percent * money
money.coerce(percent)

# Money receive :coerce with the percent,
# and returns the same things in opposite order
[money, percent]

# Then we try the operation again
money * percent
percent.coerce(money)

# Now, percent receives coerce, with money,
# and returns two more things, this time with
# the percent changed to float
[money, float]

# Finally, we can perform this operation
# without more coercion
money * float

Money.new(100) * 0.5  #=> Money.new(50)
```

```ruby
class Percent
  def coerce other
    [other, to_f]
  end
end
```

```
percent * money
money.coerce(percent)

# Money receive :coerce with the percent,
# and returns the same things in opposite order
[money, percent]

# Then we try the operation again
money * percent
percent.coerce(money)

# Now, percent receives coerce, with money,
# and returns two more things, this time with
# the percent changed to float
[money, float]

# Finally, we can perform this operation
# without more coercion
money * float

Money.new(100) * 0.5  #=> Money.new(50)
```

PERCENT KNOWS NOTHING ABOUT MONEY, AND MONEY KNOWS NOTHING ABOUT PERCENT, BUT IT ALL WORKS!

MORE COMPLICATED COERCIONS

```
percent = Percent.new(50)
10 + percent     #=> 15
10 - percent     #=> 5
10 * percent     #=> 5
10 / percent     #=> 20
```

```ruby
# Our current coerce method
def coerce other
  [other, to_f]
end
```

```ruby
percent = Percent.new(50)

# expected
10 + percent    #=> 15

#actual
10 + percent    #=> 10.5
```

```ruby
percent = Percent.new(50)
10 + percent

# percent receives coerce, and returns itself
# as a float in the second spot
percent.coerce(10)    #=> [10, 0.5]

# Now it adds those two together
10 + 0.5                  #=> 10.5
```

```ruby
# But it works as expected for multiplication!

percent = Percent.new(50)
10 * percent

percent.coerce(10)     #=> [10, 0.5]
10 * 0.5               #=> 5
```

COERCE DOESN'T TELL US WHAT METHOD WAS CALLED, SO WHAT DO WE DO?

INVESTIGATE THE CALL STACK!

```
[
  "(irb):74:in `*'",
  "(irb):79:in `irb_binding'",
  "/Users/Eric/.rvm/rubies/ruby-2.1.2/lib/ruby/2.1.0/irb/workspace.rb:86:in `eval'",
  "/Users/Eric/.rvm/rubies/ruby-2.1.2/lib/ruby/2.1.0/irb/workspace.rb:86:in `evaluate'",
  "/Users/Eric/.rvm/rubies/ruby-2.1.2/lib/ruby/2.1.0/irb/context.rb:380:in `evaluate'",
  "/Users/Eric/.rvm/rubies/ruby-2.1.2/lib/ruby/2.1.0/irb.rb:492:in `block (2 levels) in eval_input'",
  "/Users/Eric/.rvm/rubies/ruby-2.1.2/lib/ruby/2.1.0/irb.rb:624:in `signal_status'",
  "/Users/Eric/.rvm/rubies/ruby-2.1.2/lib/ruby/2.1.0/irb.rb:489:in `block in eval_input'",
  "/Users/Eric/.rvm/rubies/ruby-2.1.2/lib/ruby/2.1.0/irb/ruby-lex.rb:247:in `block (2 levels) in each_top_level_statement'",
  "/Users/Eric/.rvm/rubies/ruby-2.1.2/lib/ruby/2.1.0/irb/ruby-lex.rb:233:in `loop'",
  "/Users/Eric/.rvm/rubies/ruby-2.1.2/lib/ruby/2.1.0/irb/ruby-lex.rb:233:in `block in each_top_level_statement'",
  "/Users/Eric/.rvm/rubies/ruby-2.1.2/lib/ruby/2.1.0/irb/ruby-lex.rb:232:in `catch'",
  "/Users/Eric/.rvm/rubies/ruby-2.1.2/lib/ruby/2.1.0/irb/ruby-lex.rb:232:in `each_top_level_statement'",
  "/Users/Eric/.rvm/rubies/ruby-2.1.2/lib/ruby/2.1.0/irb.rb:488:in `eval_input'",
  "/Users/Eric/.rvm/rubies/ruby-2.1.2/lib/ruby/2.1.0/irb.rb:397:in `block in start'",
  "/Users/Eric/.rvm/rubies/ruby-2.1.2/lib/ruby/2.1.0/irb.rb:396:in `catch'",
  "/Users/Eric/.rvm/rubies/ruby-2.1.2/lib/ruby/2.1.0/irb.rb:396:in `start'",
  "/Users/Eric/.rvm/rubies/ruby-2.1.2/bin/irb:11:in `<main>'"
]
```

```
"(irb):74:in `*'",

caller[0].match("`(.+)'")[1].to_sym     #=> :*
```

```ruby
def coerce other
  method = caller[0].match("`(.+)'")[1].to_sym

  case other
  when Numeric
    case method
    when :+
      [to_f * other, other]
    else
      [other, to_f]
    end
  else
    fail TypeError, "#{self.class} can't be coerced into #{other.class}"
  end
end
```

```ruby
percent = Percent.new(50)

# Multiplication
10 * percent
percent.coerce(10)      #=> [10, 0.5]
10 * 0.5                #=> 5

# Addition
10 + percent
percent.coerce(10)      #=> [10, 5]
10 + 5                  #=> 15
```

DOESN'T REALLY FEEL RIGHT THOUGH, DOES IT?

```
10.plus(percent)
percent.coerce(10)      #=> [something, something_else]
something.plus(something_else)
```

IF IT WALKS LIKE A DUCK AND QUACKS LIKE A DUCK, IT'S PROBABLY RUBY

WHAT WE NEED IS AN OBJECT THAT RESPONDS TO #PLUS

```ruby
def coerce other
  [CoercedPercent.new(self), other]
end
```

```ruby
class CoercedPercent
  attr_reader :percent

  def initialize(percent)
    @percent = percent
  end

  def * other
    other * percent.to_f
  end

  def + other
    other + self * other
  end
end
```

```ruby
percent = Percent.new(50)

# Multiplication
10 * percent
percent.coerce(10)                    #=> [CoercedPercent.new(percent), 10]
CoercedPercent.new(percent) * 10      #=> 5

# Addition
10 + percent
percent.coerce(10)                    #=> [CoercedPercent.new(percent), 10]
CoercedPercent.new(percent) + 10      #=> 15
```

```ruby
# We can do the same thing for division and subtraction

class CoercedPercent
  [...]

  def / other
    other / percent.to_f
  end

  def - other
    other - self * other
  end
end
```

NOW, LET'S CLEAN SOME THINGS UP

BRING ON THE MONKEY PATCHING!

```ruby
module Percentable
  module Numeric
    def to_percent
      Percentable::Percent.new(self)
    end
  end
end

class Numeric
  include Percentable::Numeric
end

10.to_percent     #=> Percent.new(10)
```

WHAT ABOUT RAILS?

```ruby
module Percentable
  module Percentize
    def percentize *args
      options = args.pop if args.last.is_a? Hash

      args.each do |method_name|
        define_method(method_name) do |args=[]|
          Percent.new(super(*args) || options[:default])
        end
      end
    end
  end
end
```

```ruby
class Thingamajig < ActiveRecord::Base
  include Percentable::Percentize

  percentize :taxes, default: 10
end

t = Thingamajig.new(taxes: 20)
t.taxes     #=> Percent.new(20)
```

That's all, folks!

FURTHER READING

- Percentable by Eric Roberts
- Ruby Tapas Episode 206: Coercion by Avdi Grimm
- Class Coercion in Ruby by Zach Church
- On Obsession, Primitive and Otherwise by Colin Jones
- Monkeypatching is Destroying Ruby by Avdi Grimm