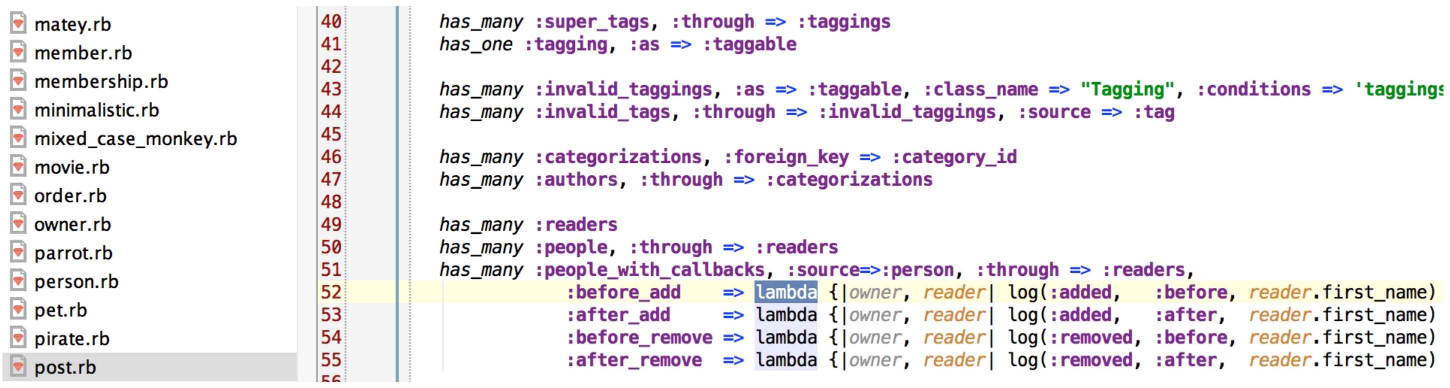
CONTACT@REACTIVE.IO (MAILTO:CONTACT@REACTIVE.IO) | (646) 820-4411 (TEL:646-820-4411)

The Official Blog of Reactive.IO



```
has_many :super_tags, :through => :taggings
has_one :tagging, :as => :taggable

has_many :invalid_taggings, :as => :taggable, :class_name => "Tagging", :conditions => 'taggings
has_many :invalid_tags, :through => :invalid_taggings, :source => :tag

has_many :categorizations, :foreign_key => :category_id
has_many :authors, :through => :categorizations

has_many :readers
has_many :people, :through => :readers
has_many :people_with_callbacks, :source=>:person, :through => :readers,
            :before_add    => lambda {|owner, reader| log(:added,   :before, reader.first_name)
            :after_add     => lambda {|owner, reader| log(:added,   :after,  reader.first_name)
            :before_remove => lambda {|owner, reader| log(:removed, :before, reader.first_name)
            :after_remove  => lambda {|owner, reader| log(:removed, :after,  reader.first_name)
```

Understanding Ruby Blocks, Procs and Lambdas (/tips/2008/12/21/understanding-ruby-blocks-procs-and-lambdas)

December 21, 2008 Robert Sosinski Ruby

Blocks, Procs and lambdas (referred to as closures (http://en.wikipedia.org/wiki/Closure_%28computer_science%29) in Computer Science) are one of the most powerful aspects of Ruby, and also one of the most misunderstood. This is probably because Ruby handles closures in a rather unique way. Making things more complicated is that Ruby has four different ways of using closures, each of which is a tad bit different, and sometimes nonsensical. There are quite a few sites with some very good information about how closures work within Ruby. But I have yet to find a good, definitive guide out there. Hopefully, this tutorial becomes just that.

First Up, Blocks

The most common, easiest and arguably most “Ruby like” way to use closures in Ruby is with blocks. They have the following familiar syntax:

```
array = [1, 2, 3, 4]

array.collect! do |n|
  n ** 2
end

puts array.inspect

# => [1, 4, 9, 16]
```

So, what is going on here?

1. First, we send the `collect!` method to an Array with a block of code.
2. The code block interacts with a variable used within the `collect!` method (`n` in this case) and squares it.
3. Each element inside the array is now squared.

Using a block with the `collect!` method is pretty easy, we just have to think that `collect!` will use the code provided within the block on each element in the array. However, what if we want to make our own `collect!` method? What will it look like? Well, lets build a method called `iterate!` and see.

```
class Array
  def iterate!
    self.each_with_index do |n, i|
      self[i] = yield(n)
    end
  end
end

array = [1, 2, 3, 4]

array.iterate! do |n|
  n ** 2
end

puts array.inspect

# => [1, 4, 9, 16]
```

To start off, we re-opened the Array class and put our `iterate!` method inside. We will keep with Ruby conventions and put a bang at the end, letting our users know to watch out, as this method might be dangerous! We then use our `iterate!` method just like Ruby's built in `collect!` method. The neat stuff however, is right in the middle of our `iterate!` method definition.

Unlike attributes, you do not need to specify the name of blocks within your methods. Instead, you can use the yield keyword. Calling this keyword will execute the code within the block provided to the method. Also, notice how we are passing `n` (the integer that the `each_with_index` method is currently working with) to yield. The attributes passed to yield corresponds to the variable specified in the piped list of the block. That value is now available to the block and returned by the yield call. So to recap what is happening:

1. Send `iterate!` to the Array of numbers.
2. When yield is called with the number `n` (first time is `1` , second time is `2` , etc…), pass the number to the block of code given.
3. The block has the number available (also called `n`) and squares it. As it is the last value handled by the block, it is returned automatically.
4. Yield outputs the value returned by the block, and rewrites the value in the array.
5. This continues for each element in the array.

What we now have is a flexible way to interact with our method. Think of blocks as giving your method an API (http://en.wikipedia.org/wiki/API), where you can determine to square each value of the array, cube them or convert each number to a string and print them to the screen. The options are infinite, making your method very flexible, and as such, very powerful.

However, that is just the beginning. Using yield is one way to use your block of code, however there is another. By calling it as a Proc. Take a look.

```
class Array
  def iterate!(&code)
    self.each_with_index do |n, i|
      self[i] = code.call(n)
    end
  end
end

array = [1, 2, 3, 4]

array.iterate! do |n|
  n ** 2
end

puts array.inspect

# => [1, 4, 9, 16]
```

Looks very similar to our previous example, however there are two differences. First, we are passing an ampersand argument called `&code`. This argument is, conveniently enough, our block. The second is in the middle of our `iterate!` method definition, where instead of using `yield`, we send `call` to our block of code. The result is exactly the same. However, if this is so, why even have this difference in syntax? Well, it lets us learn a bit more about what blocks really are. Take a look:

```
def what_am_i(&block)
  block.class
end

puts what_am_i {}

# => Proc
```

A block is just a Proc! That being said, what is a Proc?

Procedures, AKA, Procs

Blocks are very handy and syntactically simple, however we may want to have many different blocks at our disposal and use them multiple times. As such, passing the same block again and again would require us to repeat ourself. However, as Ruby is fully object-oriented, this can be handled quite cleanly by saving reusable code as an object itself. This reusable code is called a Proc (short for procedure). The only difference between blocks and Procs is that a block is a Proc that cannot be saved, and as such, is a one time use solution. By working with Procs, we can start doing the following:

```
class Array
  def iterate!(code)
    self.each_with_index do |n, i|
      self[i] = code.call(n)
    end
  end
end

array_1 = [1, 2, 3, 4]
array_2 = [2, 3, 4, 5]

square = Proc.new do |n|
  n ** 2
end

array_1.iterate!(square)
array_2.iterate!(square)

puts array_1.inspect
puts array_2.inspect

# => [1, 4, 9, 16]
# => [4, 9, 16, 25]
```

Why lowercase block and uppercase Proc?

I always write Proc as uppercase as it is a proper class within Ruby. However, blocks do not have a class of their own (they are just Procs after all) and are just a type of syntax within Ruby. As such, I write block in lowercase. Later in this tutorial, you will see me also writing lambda in lowercase. I do so for the same reason.

Notice how we do not prepend an ampersand to the code attribute in our `iterate!` method. This is because passing Procs is no different then passing any other data type. As Procs are treated just like any other object, we can start having some fun and push Ruby's interpreter to do some interesting things. Give this a try:

```
class Array
  def iterate!(code)
    self.each_with_index do |n, i|
      self[i] = code.call(n)
    end
  end
end

array = [1, 2, 3, 4]

array.iterate!(Proc.new do |n|
  n ** 2
end)

puts array.inspect

# => [1, 4, 9, 16]
```

The above is how most languages handle closures and is exactly the same as sending a block. However, if you said this does not look “Ruby like”, I would have to agree. The above reason is exactly why Ruby has blocks to begin with, and that is to stay within its familiar `end` concluding syntax.

If this is the case, why just not use blocks exclusively? Well, the answer is simple, what if we want to pass two or more closures to a method? If this is the case, blocks quickly become too limiting. By having Procs however, we can do something like this:

```
def callbacks(procs)
  procs[:starting].call

  puts "Still going"

  procs[:finishing].call
end

callbacks(:starting => Proc.new { puts "Starting" },
          :finishing => Proc.new { puts "Finishing" })

# => Starting
# => Still going
# => Finishing
```

So, when should you use blocks over Procs? My logic is as follows:

1. Block: Your method is breaking an object down into smaller pieces, and you want to let your users interact with these pieces.
2. Block: You want to run multiple expressions atomically, like a database migration.
3. Proc: You want to reuse a block of code multiple times.
4. Proc: Your method will have one or more callbacks.

Lambdas

So far, you have used Procs in two ways, passing them directly as an attribute and saving them as a variable. These Procs act very similar to what other languages call anonymous functions, or lambdas. To make things more interesting, lambdas are available within Ruby too. Take a look:

```
class Array
  def iterate!(code)
    self.each_with_index do |n, i|
      self[i] = code.call(n)
    end
  end
end

array = [1, 2, 3, 4]

array.iterate!(lambda { |n| n ** 2 })

puts array.inspect

# => [1, 4, 9, 16]
```

On first look, lambdas seem to be exactly the same as Procs. However, there are two subtle differences. The first difference is that, unlike Procs, lambdas check the number of arguments passed.

```
def args(code)
  one, two = 1, 2
  code.call(one, two)
end

args(Proc.new{|a, b, c| puts "Give me a #{a} and a #{b} and a #{c.class}"})

args(lambda{|a, b, c| puts "Give me a #{a} and a #{b} and a #{c.class}"})

# => Give me a 1 and a 2 and a NilClass
# *.rb:8: ArgumentError: wrong number of arguments (2 for 3) (ArgumentError)
```

We see with the Proc example, extra variables are set to `nil`. However with lambdas, Ruby throws an error instead.

The second difference is that lambdas have diminutive returns. What this means is that while a Proc return will stop a method and return the value provided, lambdas will return their value to the method and let the method continue on. Confused? Lets take a look at an example.

```
def proc_return
  Proc.new { return "Proc.new"}.call
  return "proc_return method finished"
end

def lambda_return
  lambda { return "lambda" }.call
  return "lambda_return method finished"
end

puts proc_return
puts lambda_return

# => Proc.new
# => lambda_return method finished
```

In `proc_return` , our method hits a return keyword, stops processing the rest of the method and returns the string `Proc.new` . On the other hand, our `lambda_return` method hits our lambda, which returns the string `lambda` , keeps going and hits the next return and outputs `lambda_return method finished` . Why the difference?

The answer is in the conceptual differences between procedures and methods. Procs in Ruby are drop in code snippets, not methods. Because of this, the Proc return is the `proc_return` method's return, and acts accordingly. Lambdas however act just like methods, as they check the number of arguments and do not override the calling methods return. For this reason, it is best to think of lambdas as another way to write methods, an anonymous way at that.

So, when should you write an anonymous method (lambda) instead of a Proc? The following code shows one such case.

```
def generic_return(code)
  code.call
  return "generic_return method finished"
end

puts generic_return(Proc.new { return "Proc.new" })
puts generic_return(lambda { return "lambda" })

# => *.rb:6: unexpected return (LocalJumpError)
# => generic_return method finished
```

Part of Ruby's syntax is that arguments (a Proc in this example) cannot have a return keyword in it. However, a lambda acts just like a method, which can have a literal return, and thus sneaks by this requirement unscathed! This different in semantics shows up in situations like the following example.

```
def generic_return(code)
  one, two    = 1, 2
  three, four = code.call(one, two)
  return "Give me a #{three} and a #{four}"
end

puts generic_return(lambda { |x, y| return x + 2, y + 2 })

puts generic_return(Proc.new { |x, y| return x + 2, y + 2 })

puts generic_return(Proc.new { |x, y| x + 2; y + 2 })

puts generic_return(Proc.new { |x, y| [x + 2, y + 2] })

# => Give me a 3 and a 4
# => *.rb:9: unexpected return (LocalJumpError)
# => Give me a 4 and a
# => Give me a 3 and a 4
```

Here, our method `generic_return` is expecting the closure to return two values. Doing this without the return keyword becomes dicy though. With a lambda, everything is easy. However with a Proc, we ultimately have to take advantage how Ruby interprets Arrays with assignment.

So, when to use Proc over lambdas and vice versa? Honestly, besides argument checking, the difference is just in how you see closures. If you want to stay in the mindset of passing blocks of code, keep with Proc. If sending a method to another method that can return a method makes sense to you, use lambdas. But, if lambdas are just methods in object form, can we store existing methods and pass them just like Procs? For that, Ruby has the something pretty tricky up its sleeve.

Method Objects

So, you already have a method that works, but you want to pass it to another method as a closure and keep your code DRY (http://en.wikipedia.org/wiki/Don%27t_repeat_yourself). To do this, you can take advantage of Ruby's `method` method.

```
class Array
  def iterate!(code)
    self.each_with_index do |n, i|
      self[i] = code.call(n)
    end
  end
end

def square(n)
  n ** 2
end

array = [1, 2, 3, 4]

array.iterate!(method(:square))

puts array.inspect

# => [1, 4, 9, 16]
```

In this example, we already have a method called `square` that would work out just fine for the task at hand. As such, we can reuse it as a parameter by converting it into a Method object and passing it to our `iterate!` method. But, what is this new object type?

```
def square(n)
  n ** 2
end

puts method(:square).class

# => Method
```

Just as you guessed, `square` is not a Proc, but a Method. The neat thing is that this Method object will act just like a lambda, because the concept is the same. This method however, is a named method (called `square`) while lambdas are anonymous methods.

Conclusion

So to recap, we went through Ruby's four closure types, blocks, Procs, lambdas and Methods. We also know that blocks and Procs act like drop-in code snippets, while lambdas and Methods act just like methods. Finally, through a slew of code examples, you were able to see when to use which and how to use each effectively. Now, you should be able to start using this expressive and interesting feature of Ruby in your own code, and start offering flexible and powerful methods to other developers you work with.

About Reactive.IO

Based in NYC, Reactive.IO builds your mission-critical software right the first time.

Get Reactive.TODAY

Subscribe to Reactive.TODAY, the newsletter that keeps you ahead of the competition.

Email Address Subscribe

New Reactive.TIPS

Understanding Optionals in Swift (/tips/2015/03/02/understanding-optionals-in-swift)

Introduction to Distributed Messaging with Elixir (/tips/2015/02/07/introduction-to-distributed-messaging-with-elixir)

Introduction to Parallel Computing with Elixir (/tips/2015/02/03/introduction-to-parallel-computing-with-elixir)

Get Connected

222 Broadway
New York, NY 10038
(https://www.google.com/maps/place/222+Broadway/@40.710968,-74.0084713,18z
/data=!3m1!4b1!4m2!3m1!1s0x89c25a18456b1dc7:0xa07b4b73fc1baa2d)

(646) 820-4411 (tel:646-820-4411)
contact@reactive.io (mailto:contact@reactive.io)

Stay Connected

in (https://www.linkedin.com/company/reactive-io)

(https://github.com/reactive-io)

(https://angel.co/reactive-io) </>
(https://coderwall.com/team/reactive-io)
(https://twitter.com/reactive_io)