Ruby Explained: Blocks, Procs, and Lambdas, aka "Closures"

This post will explain blocks and Procs, with which you're probably at least somewhat familiar, but also some lesser known Ruby closures like Lambdas and Methods

One of the most confusing parts of learning basic Ruby (until your AHA! moment) is understanding what blocks are and how they work, mostly because it's something you probably haven't ever seen before trying out Ruby. It shouldn't be, because they're actually pretty simple. You HAVE already seen them before... they are commonly used as inputs to some of the iterators you've no doubt worked with like `each` or `map`.

Here, you'll learn more about blocks and also about their lessor known cousins, Procs, lambdas and Methods. By the end, you should be comfortable working with blocks and writing your own methods that take them. You should understand when you may need to use a Proc instead and the basics of the other two options -- lambdas and Methods.

Blocks are just chunks of code that you can pick up and drop into another method as an input. They're often called "anonymous functions" because they have no name but behave much like functions. They're like little helper functions... you don't find blocks just hanging around without some method (like `#each`) using them.

You **declare a block** using squiggly braces `{}` if it's on one line or `do ... end` if it's on multiple lines (by convention... you can use either one if you really want):

```
> [1,2,3].each { |num| print "#{num}! " }
1! 2! 3! =>[1,2,3]
> [1,2,3].each do |num|
>    print "#{num}!"
> end
1! 2! 3! =>[1,2,3]         # Identical to the first case.
```

Just like methods, some blocks take inputs, others do not. Some return important information, others do not. Blocks let you use the implicit **return** (whatever's on the last line) but NOT `return`, since that will return you from whatever method actually called the block.

Blocks are used as arguments to other methods (like `#each`), just like the normal arguments that you see between the parentheses... they just happen to always be listed last and on their own because they tend to take up multiple lines. Don't think of them as anything too special. The `#each` method isn't special either, it's just built to accept a block as an argument.

How does `#each` take a block then? Through the magic of **the `yield` statement**, which basically says "run the block right here". When you write your own methods, you don't even need to specially declare that you'd like to accept a block. It will just be there waiting for you when you call `yield` inside your method.

`yield` can **pass parameters to your block** as well. See this made-up version of the `#each` method to get an idea of what's happening under the hood. We'll put this method into the Array class so you can call it directly on an array (like `[1,2,3].my_each`) instead of having to take the array as an argument like `my_each([1,2,3])`:

```
class Array
```

```
  def my_each
    i = 0
    while i < self.size
        yield(self[i])
        i+=1
    end
    self
  end
end
```

As you can see, we iterate over the array that our `#my_each` method was called on (which can be grabbed using `self`). Then we call the block that got passed to `#my_each` and pipe in whatever member of the original array we are currently on. Last, we just return the original array because that's what `#each` does. We would run it just the same way as `#each`:

```
> [1,2,3].my_each { |num| print "#{num}!" }
1! 2! 3! => [1,2,3]
```

Which operates in that case just like all these lines:

```
class Array
  def my_each
    i = 0
    while i < self.size
        print "#{self[i]}!"   # Our block got "subbed in" here
        i+=1
    end
    self
  end
end
```

So one reason blocks are great is because you can write a sort of generic method like `#each` which wraps your block in code that says what to do with it. Another use case is when creating methods where you want to optionally be able to override how they "work" internally by supplying your own block -- `#sort` lets you supply your own block to determine how to actually order the items of the array if you want to!

If you want to **ask whether a block was passed** at all (to only yield in that case), use `#block_given?`, or rather: `yield if block_given?`

A lot of beginners just blindly take it on faith that `#each` and `#map` and `#select` etc. all work the way they do. You're more skeptical than that, which is good. They're really quite simple and you'll get a chance to build your own soon enough.

What if you want to pass TWO blocks to your function? What if you want to save your block to a variable so you can use it again later? That's a job for **Procs**, aka Procedures! Actually, a block *is* a Proc (which is the class name for a block) and they rhyme just to confuse you. The block is sort of like a stripped-down and temporary version of a Proc that Ruby included just to make it really easy to use things like those `#each` iterators.

A Proc is just a block that you save to a variable, thereby giving it a bit more permanence:

```
> my_proc = Proc.new { |arg1| print "#{arg1}! " }
```

Use that block of code (now called a Proc) as an input to a function by prepending it with an apersand `&`:

```
> [1,2,3].each(&my_proc)
1! 2! 3! =>[1,2,3]
```

It's the same as passing the block like you did before!

When you create your own function to accept procs, the guts need to change a little bit because you'll need to use `#call` instead of `yield` inside (because which proc would `yield` run if you had more than one?). `#call` literally just runs the Proc that is called on. You can give it arguments as well to pass on to the Proc:

```
> my_proc.call("howdy ")  # edit: note that this is the same `m
howdy => nil
```

Most of the time, using a block is more than sufficient, especially in your early projects. Once you start seeing the need for using a Proc (like passing multiple arguments or saving it for later as a callback), you'll have Procs there waiting for you.

Blocks and Procs are both a type of "closure". A closure is basically a formal, computer-science-y way of saying "a chunk of code that you can pass around but which hangs onto the variables that you gave it when you first called it". It's the blanket term used to refer to blocks and Procs and...

There are two other similar closures to be aware of but about which you certainly don't need to be an expert because they're used in less typical applications. The first of these is a **lambda**. If Procs are sort of a more-fleshed-out version of blocks, then lambdas are sort of a more-fleshed-out version of Procs. They are one step closer to being actual methods themselves, but still technically count as anonymous functions. If you're coming from Javascript, anonymous functions shouldn't be anything new to you.

Just to focus on the differences between lambdas and Procs, a lambda acts more like a real method. What does that mean?

- A lambda gives you more flexibility with what it returns (like if you want to return multiple values at once) because you can safely use the explicit `return` statement inside of one. With lambdas, `return` will only return from the lambda itself and not the enclosing method, which is what happens if you use `return` inside a block or Proc.
- Lambdas are also much stricter than Procs about you passing them the correct number of arguments (you'll get an error if you pass the wrong number).

Here's a simple example to show you the syntax of a lambda (btw, there's nothing special to lambdas about placing the `#call` after the `end`, if you hadn't seen that done before, it's just like method chaining):

```
 > lambda do |word|
>   puts word
>   return word            # you can do this in lambdas not Pro
> end.call("howdy ")
howdy => "howdy "        # not nil because we gave it a return
```

The second additional closure is called a **Method** because, well, it's the closest of the four (blocks, Procs, lambdas, and Methods) to an actual method. Which it is. "Method"'s (capitalized because they're actually a class of their own) are really just a convenient way to pass a normal method to another normal method by wrapping the symbol of its name in the word `method()` So what? To use the same example as we have been so far:

Edit: Note that `#my_each` has been modified for this example to now take an argument, which the standard `#each` does not. We're using `#my_each` below.

```
class Array
  def my_each(some_method)
    i = 0
    while i < self.size
      some_method.call(self[i])
      i+=1
    end
  end
  self
end

def print_stuff(word)
  print "#{word}! "
end

> [1,2,3].my_each(method(:print_stuff))    # symbolize the name
```

```
1! 2! 3! => nil
```

TADA! We've now gotten our worthless array (and a "howdy") printed out 4 different ways, each seemingly less useful than the last! Fear not. Learn your blocks cold and have a good handle on Procs (which should be easy since they're basically the same thing) but just keep lambdas and Methods in the back of your head for much later.

So...

- **Blocks** are unnamed little code chunks you can drop into other methods. Used all the time.
- **Procs** are identical to blocks but you can store them in variables, which lets you pass them into functions as explicit arguments and save them for later. Used explicitly sometimes.
- **Lambdas** are really full methods that just haven't been named. Used rarely.
- **Methods** are a way of taking actual named methods and passing them around as arguments to or returns from other methods in your code. Used rarely.
- **Closure** is just the umbrella term for all four of those things, which all somehow involve passing around chunks of code.

The "Ruby Explained" posts are designed to be a sort of "In-Plain-English" version of key Ruby concepts which are usually covered in other introductory texts but rarely for free and often incompletely. When I'm learning a new thing, I usually want someone to explain it to me like I'm a five year old because that's the best way to make sure nothing gets missed. This is my attempt to pass that same sentiment on to you. Let me know if there's anything I can improve.

If you're just getting interested in this stuff, check out The Odin Project for a free curriculum to learn web development.

Tags: *Ruby Explained*

Recent Posts

Startup Addiction

A Quick Preview of the Viking Code School

Entrepreneurial Epiphanies #4: Do Nothing Quietly

TheOdinProject.com Redesign Part II: Turning Pretty Mockups Into Code

---

Erik Trautman is the founder of Viking Code School, the premier program for learning web development online
He writes about startups, web design, development, education, and creativity