Ruby Procs And Lambdas (And The Difference Between Them)

May 28, 2010 By Alan Skorkin 21 Comments

As you know, I am a big believer in knowing the basics. I find it is especially valuable to go back to the fundamentals of whatever skill you're trying to master periodically, as you gain more experience. Somehow you're always able to extract something valuable from the experience due to the extra insight you've acquired along the way. Recently I've been looking more closely at some Ruby concepts that I may have glossed over when I was learning the language initially, so I will likely be writing about this in some of my upcoming posts. I thought I would begin with one of the most powerful features of Ruby – **procs** (*and lambdas :)*).

What's So Good About Procs?

You know how everything in Ruby is an object, well, as it turns out that's not quite true. Ruby blocks are not objects! We can discuss the implications of that, but let's refrain, accept it as fact and move on. So, blocks are not objects, but you can turn them into objects without too much trouble. We do this by wrapping our block in an instance of the Proc class (*more on this shortly*). This is really great since it turns our block into a first class function, which in turn allows Ruby to support closures and **once a language has closures, you can do all sorts of interesting things** like making use of various functional concepts. That however is a story for another time, before we dive deep into those areas, let us really understand how Procs work.

Procs And Lambdas In Detail

There are three ways to create a Proc object in Ruby. Well, actually there are four ways, but the last one is implicit. Here they are:

1. **Using Proc.new**. This is the standard way to create any object, you simply need to pass in a block and you will get back a Proc object which will run the code in the block when you invoke its call method.

```
proc_object = Proc.new {puts "I am a proc object"}
proc_object.call
```

```
alan@alan-ubuntu-vm:~/tmp$ ruby a.rb
I am a proc object
```

2. **Using the proc method in the Kernel module**. Being in the Kernel module, this method is globally available. It is basically equivalent to *Proc.new* in Ruby

Receive Rss Updates

RSS - Posts

Receive Email Updates

Email Address

SUBSCRIBE

Popular Posts

- 10 Awesome Fantasy Series That Are Not Potter or LoTR
- The Difference Between A Developer, A Programmer And A Computer Scientist
- How A Ruby Case Statement Works And What You Can Do With It
- Here Are Some Words That Rhyme With Orange!
- You Don't Need Math Skills To Be A Good Developer But You Do Need Them To Be A Great One
- Method Arguments In Ruby
- Bash Shortcuts For Maximum Productivity
- Why Developers Never Use State Machines
- Serializing (And Deserializing) Objects With Ruby
- How To Quickly Generate A Large File On The Command Line (With Linux)

1.9, but not in Ruby 1.8. In Ruby 1.8 the *proc* method is equivalent to *lambda*. As you may have guessed there is a difference between procs and lambdas (*see below*), so you need to be aware of whether you're using Ruby 1.8 or 1.9. If you're using 1.9 there is a way to find out if you're dealing with a proc or a lambda. The Proc object *lambda?* method:

```
proc_object = proc {puts "Hello from inside the proc"}
proc_object.call
puts "Is proc_object a lambda - #{proc_object.lambda?}"
```

```
alan@alan-ubuntu-vm:~/tmp$ rvm use 1.9.1
Using ruby 1.9.1 p378
alan@alan-ubuntu-vm:~/tmp$ ruby a.rb
Hello from inside the proc
Is proc_object a lambda - false
```

If you're in 1.8 this method will not exist at all. By the way, did you notice how I had to switch Rubies and was able to do so painlessly through the awesome power of RVM, hope you're already using it. As far as I am concerned, if you need to switch between different versions of Ruby, don't bother with the *proc* method, just use *Proc.new* and you will never have an issue.

3. **Using the Kernel lambda method**. Once again, this is globally available being in the Kernel module, using this method will create a Proc object, but it will be a lambda as opposed to a proc (*if you really need to know what the differences are at this point you can skip to the explanation :)*).

```
proc_object = lambda {puts "Hello from inside the proc"}
proc_object.call
puts "Is proc_object a lambda - #{proc_object.lambda?}"
```

```
alan@alan-ubuntu-vm:~/tmp$ ruby a.rb
Hello from inside the proc
Is proc_object a lambda - true
```

4. **The implicit way**. When you write a method which will accept a block, there are two ways you can go about it. The first is to simply use the *yield* keyword inside your method. If there is a *yield* in a method, but there is no block you will get an error.

```
def my_method
  puts "hello method"
  yield
end

my_method {puts "hello block"}
my_method
```

```
alan@alan-ubuntu-vm:~/tmp$ ruby a.rb
hello method
hello block
hello method
a.rb:7:in `my_method': no block given (yield) (LocalJumpE
    from a.rb:11:in `<main>'
```

The other way is to use a special parameter in your method parameter list. This special parameter can have the same name but you will need to precede it with an ampersand. The block you pass into this method will then be associated with this parameter, you will infact no longer have a block – it will be a Proc object. You will simply need to call the Proc *call* method to execute the code in the block you passed in. The difference is, you can now return your block (*or rather the Proc object*) as the return value of the method, thereby getting your hands on a Proc object without having to use any of the above three methods explicitly.

```
def my_method(&my_block)
  puts "hello method"
  my_block.call
  my_block
end

block_var = my_method {puts "hello block"}
block_var.call
```

```
alan@alan-ubuntu-vm:~/tmp$ ruby a.rb
hello method
hello block
hello block
```

Those are all the different ways to get a Proc object in Ruby (*either a proc or a lambda*).

Fun And Interesting Trivia About Procs And Lambdas

If you have a method to which you pass a block, but instead of calling *yield*, you call *Proc.new* without giving it a block of its own, it will return a proc that represents the block you passed in to the method e.g.:

```
def some_method
  Proc.new
end

my_proc = some_method{puts "I am a happy proc"}
my_proc.call
```

```
alan@alan-ubuntu-vm:~/tmp$ ruby a.rb
I am a happy proc
```

In Ruby 1.9, there is actually another way to create lambdas. It isn't really a different way, it is simply some syntactic sugar, but it is crazy looking, so worth knowing :). The normal way to create a lambda is using the *lambda* keyword, and if you want your lambda to take parameters, you simply pass them in the normal block way e.g.:

```
normal_lambda = lambda {|param1, param2| puts "para
normal_lambda.call(10, 20)
```

```
alan@alan-ubuntu-vm:~/tmp$ ruby a.rb
param1: 10 and param2: 20
```

With the new syntax it is slightly different. We replace the lambda method with the -> method (*as in minus followed by greater than sign*). The parameters move out of the block, and move into some brackets right after the -> method (*so they look like method parameters, but in reality are still block parameters*). Everything else is the same, so our previous example would look like:

```
normal_lambda = ->(param1, param2) {puts "param1: #
```

```
normal_lambda.call(10, 20)
```

```
alan@alan-ubuntu-vm:~/tmp$ ruby a.rb
param1: 10 and param2: 20
```

Everything still works fine (*in Ruby 1.9, don't try this in 1.8*), but if you're using Netbeans, this syntax is way too crazy for it and it will barf all over itself. This syntax is an interesting curiosity, but **I recommend sticking to plain old lambda for your own sanity** (*especially if you're using Netbeans*) and that of everyone else who has to read your code later.

Procs Are First Class (Functions That Is)

Once we have created our first class function (*in Proc form*), there are a few things we may want to do with it, such as calling it or comparing it to another Proc. As usual, Ruby makes all of this interesting. Let's look at Proc equality first. You can compare procs or lambdas to each other, using the == method (*just like with any other object*). Unlike many other objects though, defining exactly the same proc twice will not make them equal. For example:

```
string1 = "blah"
string2 = "blah"

puts "#{string1 == string2}"

proc1 = Proc.new{"blah"}
proc2 = Proc.new{"blah"}

puts "#{proc1 == proc2}"
```

```
alan@alan-ubuntu-vm:~/tmp$ ruby a.rb
String are equal? - true
Procs are equal? - false
```

There are only two ways that two procs or lambdas will be equal to each other. One way is to define two really simple procs that have the same object in them e.g.:

```
proc1 = Proc.new{string1}
proc2 = Proc.new{string1}

puts "Procs are equal? - #{proc1 == proc2}"
```

```
alan@alan-ubuntu-vm:~/tmp$ ruby a.rb
Procs are equal? - true
```

This is not really very useful. The other way that two procs will ever be equal is if they are clones of each other (*one was created by using the clone or dup method on the other*).

```
proc1 = Proc.new{|x|"blah1"*x}
proc2 = proc1.dup

puts "Procs are equal? - #{proc1 == proc2}"
```

```
alan@alan-ubuntu-vm:~/tmp$ ruby a.rb
Procs are equal? - true
```

Let us now look at executing procs. We have already seen the call method; this is the most common way to execute a proc or lambda.

```
puts Proc.new{|x|"blah1"*x}.call(2)
```

```
alan@alan-ubuntu-vm:~/tmp$ ruby a.rb
blah1blah1
```

But, there are actually two other ways. The first is to use the array syntax, this works in both Ruby 1.8 and 1.9:

```
puts Proc.new{|x|"blah1"*x}[2]
```

```
alan@alan-ubuntu-vm:~/tmp$ ruby a.rb
blah1blah1
```

The other is the dot syntax, which only works in 1.9.

```
puts Proc.new{|x|"blah1"*x}.(2)
```

```
alan@alan-ubuntu-vm:~/tmp$ ruby a.rb
blah1blah1
```

This syntax being 1.9 only will once again cause Netbeans to barf all over itself :). Both of those ways are nice and succinct, but **I still recommend you use *call*; it is not that much longer and is much more readable, both for yourself and for others**. The array syntax could be especially confusing, consider:

```
some_var[2,3]
```

That could refer to an array, a string, a proc – without context it's a bit too ambiguous for my tastes, stick to *call*.

Even More Trivia About Procs And Lambdas

You may not know this, but procs and lambdas have an *arity* method which allows you to find out how many arguments a Proc objects expects to receive (*Method objects also have this method*). For example:

```
my_proc = Proc.new{|x|"blah1"*x}
puts "I need #{my_proc.arity} arguments"
```

```
alan@alan-ubuntu-vm:~/tmp$ ruby a.rb
I need 1 arguments
```

Pretty simple – the fun bit starts when the block you use to create the proc expects an arbitrary number of arguments (*or requires some, but sponges up the rest using a * prefixed final parameter*) e.g.:

```
my_proc = Proc.new{|x, *rest|"blah1 #{x} - #{rest}"
puts "I need #{my_proc.arity} arguments"
```

```
alan@alan-ubuntu-vm:~/tmp$ ruby a.rb
I need -2 arguments
```

What the hell? **When you get a negative number for you arity, it means there is potentially an arbitrary number of arguments**

involved. In this case, all you can do is find out how many arguments are required (*the rest will be optional*). You do this by using the ~ operator on the return value of the arity method:

```
my_proc = Proc.new{|x, *rest|"blah1 #{x} - #{rest}"
puts "I actually require #{~my_proc.arity} argument
```

```
alan@alan-ubuntu-vm:~/tmp$ ruby a.rb
I actually require 1 arguments, the rest are option
```

I reckon that's interesting stuff to know, it won't save your life on a daily basis, but there will be that one time when knowing this will make you look like an absolute superstar :).

So What IS The Difference Between Procs And Lambdas?

It took a while, but we finally got here. Long story short, **the real difference between procs and lambdas has everything to do with control flow keywords**. I am talking about return, raise, break, redo, retry etc. – those control words. Let's say you have a return statement in a proc. When you call your proc, it will not only dump you out of it, but will also return from the enclosing method e.g.:

```
def my_method
  puts "before proc"
  my_proc = Proc.new do
    puts "inside proc"
    return
  end
  my_proc.call
  puts "after proc"
end

my_method
```

```
alan@alan-ubuntu-vm:~/tmp$ ruby a.rb
before proc
inside proc
```

The final *puts* in the method, was never executed, since when we called our proc, the return within it dumped us out of the method. If, however, we convert our proc to a lambda, we get the following:

```
def my_method
  puts "before proc"
  my_proc = lambda do
    puts "inside proc"
    return
  end
  my_proc.call
  puts "after proc"
end

my_method
```

```
alan@alan-ubuntu-vm:~/tmp$ ruby a.rb
before proc
inside proc
```

```
after proc
```

The return within the lambda only dumps us out of the lambda itself and the enclosing method continues executing. The way control flow keywords are treated within procs and lambdas is the main difference between them. Or rather, the way that the *return* keyword and the *break* keyword are treated is the difference since most of the others are treated in exactly the same way in both procs and lambdas. Let's have a look at *break*. If you have a break keyword within a lambda it is treated just like a return:

```
def my_method
  puts "before proc"
  my_proc = lambda do
    puts "inside proc"
    break
  end
  my_proc.call
  puts "after proc"
end

my_method
```

```
alan@alan-ubuntu-vm:~/tmp$ ruby a.rb
before proc
inside proc
after proc
```

However, if we change the lambda to a proc:

```
def my_method
  puts "before proc"
  my_proc = proc do
    puts "inside proc"
    break
  end
  my_proc.call
  puts "after proc"
end

my_method
```

```
alan@alan-ubuntu-vm:~/tmp$ ruby a.rb
before proc
inside proc
a.rb:64:in `block in my_method': break from proc-closure (Lo
    from a.rb:66:in `call'
    from a.rb:66:in `my_method'
    from a.rb:70:in `<main>'
```

We get a LocalJumpError, what happened? Well, the break keyword is used to break out of iteration, and we didn't have an iterator around our proc, so we couldn't break out of anything, so Ruby tries to punish us with an error. If we put an iterator around our proc, everything would be fine:

```
def my_method
  puts "before proc"
  my_proc = proc do
    puts "inside proc"
    break
  end
  [1,2,3].each {my_proc.call}
  puts "after proc"
end

my_method
```

```
alan@alan-ubuntu-vm:~/tmp$ ruby a.rb
before proc
inside proc
after proc
```

So, how come the lambda didn't have the same error, we didn't have an iterator there either. Well, this is the other difference between procs and lambdas. Having a *break* keyword within a lambda will cause us to break out of the lambda itself, but within a proc we get an error if there is no iterator around the proc and we break out of the iterator if we do have one.

There you go, the difference between Ruby procs and lambdas. There are other control flow keywords (*next, redo, retry, raise*), but they behave the same in both procs and lambdas, that behaviour, however, is quite interesting, I urge you to have a play with it. More interesting Ruby fundamentals soon.

Images by rofi and Richard Moross

Share this:

Twitter 59 Facebook 14 Google Reddit More

21 Comments Skorks 1 Login

Recommend 2 Share Sort by Best

Join the discussion…

Adam · a year ago
Great great great. Thank you for taking the time to write such a readable lesson.
· Reply · Share ›

aash dhariya · a year ago
Nice post. really helpful. Thank You
· Reply · Share ›

Evan R. Murphy · 2 years ago
Very helpful, thank you!
· Reply · Share ›

kishore cheruku · 2 years ago
Thank you so much for valuable information
· Reply · Share ›

kcolclasure · 2 years ago
Very useful article... Thanks, I'll be checking back again!
· Reply · Share ›

kranthi · 3 years ago
Superb!!! Waiting for some more nice articles..keep going!! :)
· Reply · Share ›

JoeJack · 4 years ago
Great article, but cumbersome to read with all of the errant HTML symbols.
· Reply · Share ›

pcreux · 5 years ago
Great article! I look forward new articles to know more about... the basics!
· Reply · Share ›

sunaku · 5 years ago
In the future, it would be helpful if you could run your `a.rb` examples with `ruby -v` so that we know exactly which version of Ruby is being used:

ruby -v a.rb

Thanks for the article.
· Reply · Share ›

Alan Skorkin → sunaku · 5 years ago
Hi Suraj,

Fair point, I almost always use 1.9.1 these days.
· Reply · Share ›

pechorin-andrey · 5 years ago
Hello Alan.
Thanks for article, also sorry for my bad english, i'm from russia.

It is very good what people can find intersting articles about ruby. Many peoples uses procs/iterations but doesn't understand this things right.

I start to learn C now, because i want to have deep understanding about ruby lang :) I'm really love this lang.

You can write me message, it's very intersting to have some discuss with you.

Russia,Moscow :)
· Reply · Share ›