Blocks and Procs

Chapter 10

This is definitely one of the coolest features of Ruby. Some other languages have this feature, though they may call it something else (like closures), but most of the more popular ones don't, and it's a shame.

So what is this cool new thing? It's the ability to take a block of code (code in between do and end), wrap it up in an object (called a proc), store it in a variable or pass it to a method, and run the code in the block whenever you feel like (more than once, if you want). So it's kind of like a method itself, except that it isn't bound to an object (it is an object), and you can store it or pass it around like you can with any object. I think it's example time:

```
toast = Proc.new do
  puts 'Cheers!'
end

toast.call
toast.call
toast.call
```

```
Cheers!
Cheers!
Cheers!
```

So I created a proc (which I think is supposed to be short for "procedure", but far more importantly, it rhymes with "block") which held the block of code, then I called the proc three times. As you can see, it's a lot like a method.

Actually, it's even more like a method than I have shown you, because blocks can take parameters:

```
doYouLike = Proc.new do |aGoodThing|
  puts 'I *really* like '+aGoodThing+'!'
end

doYouLike.call 'chocolate'
doYouLike.call 'ruby'
```

```
I *really* like chocolate!
I *really* like ruby!
```

Ok, so we see what blocks and procs are, and how to use them, but what's the point? Why not just use methods? Well, it's because there are some things you just can't do with methods. In particular, you can't pass methods into other methods (but you can pass procs into methods), and methods can't return other methods (but they can return procs). This is simply because procs are objects; methods aren't.

(By the way, is any of this looking familiar? Yep, you've seen blocks before... when you learned about iterators. But let's talk more about that in a bit.)

Methods Which Take Procs

When we pass a proc into a method, we can control how, if, or how many times we call the proc. For example, let's say there's something we want to do before and after some code is run:

```ruby
def doSelfImportantly someProc
  puts 'Everybody just HOLD ON!  I have something to do...'
  someProc.call
  puts 'Ok everyone, I\'m done.  Go on with what you were doing.'
end

sayHello = Proc.new do
  puts 'hello'
end

sayGoodbye = Proc.new do
  puts 'goodbye'
end

doSelfImportantly sayHello
doSelfImportantly sayGoodbye
```

```
Everybody just HOLD ON!  I have something to do...
hello
Ok everyone, I'm done.  Go on with what you were doing.
Everybody just HOLD ON!  I have something to do...
goodbye
Ok everyone, I'm done.  Go on with what you were doing.
```

Maybe that doesn't appear particulary fabulous... but it is. 😃 It's all too common in programming to have strict requirements about what must be done when. If you want to save a file, for example, you have to open the file, write out the information you want it to have, and then close the file. If you forget to close the file, Bad Things(tm) can happen. But each time you want to save or load a file, you have to do the same thing: open the file, do what you really want to do, then close the file. It's tedious and easy to forget. In Ruby, saving (or loading) files works similarly to the code above, so you don't have to worry about anything but what you actually want to save (or load). (In the next chapter I'll show you where to find out how to do things like save and load files.)

You can also write methods which will determine how many times, or even if to call a proc. Here's a method which will call the proc passed in about half of the time, and another which will call it twice:

```ruby
def maybeDo someProc
```

```
  if rand(2) == 0
    someProc.call
  end
end

def twiceDo someProc
  someProc.call
  someProc.call
end

wink = Proc.new do
  puts '<wink>'
end

glance = Proc.new do
  puts '<glance>'
end

maybeDo wink
maybeDo glance
twiceDo wink
twiceDo glance
```

```
<wink>
<wink>
<glance>
<glance>
```

These are some of the more common uses of procs which enable us to do things we simply could not have done using methods alone. Sure, you could write a method to wink twice, but you couldn't write one to just do something twice!

Before we move on, let's look at one last example. So far the procs we have passed in have been fairly similar to each other. This time they will be quite different, so you can see how much such a method depends on the procs passed into it. Our method will take some object and a proc, and will call the proc on that object. If the proc returns false, we quit; otherwise we call the proc with the returned object. We keep doing this until the proc returns false (which it had better do eventually, or the program will crash). The method will return the last non-false value returned by the proc.

```
def doUntilFalse firstInput, someProc
  input  = firstInput
  output = firstInput

  while output
    input  = output
    output = someProc.call input
  end

  input
end

buildArrayOfSquares = Proc.new do |array|
  lastNumber = array.last
  if lastNumber <= 0
    false
  else
    array.pop                         # Take off the last number...
    array.push lastNumber*lastNumber  # ...and replace it with its square...
    array.push lastNumber-1           # ...followed by the next smaller number.
  end
end

alwaysFalse = Proc.new do |justIgnoreMe|
  false
end

puts doUntilFalse([5], buildArrayOfSquares).inspect
puts doUntilFalse('I\'m writing this at 3:00 am; someone knock me out!',
alwaysFalse)
```

```
[25, 16, 9, 4, 1, 0]
I'm writing this at 3:00 am; someone knock me out!
```

Ok, so that was a pretty weird example, I'll admit. But it shows how differently our method acts when given very different procs.

The inspect method is a lot like to_s, except that the string it returns tries to show you the ruby code for building the object you passed it. Here it shows us the whole array returned by our first call to doUntilFalse. Also, you might notice that we never actually squared that 0 on the end of that array, but since 0 squared is still just 0, we didn't have to. And since alwaysFalse was, you know, always false, doUntilFalse didn't do anything at all the second time we called it; it just returned what was passed in.

Methods Which Return Procs

One of the other cool things you can do with procs is to create them in methods and return them. This allows all sorts of crazy programming power (things with impressive names, like lazy evaluation, infinite data structures, and currying), but the fact is that I almost never do this in practice, nor can I remember seeing anyone else do this in their code. I think it's the kind of thing you don't usually end up having to do in Ruby, or maybe Ruby just encourages you to find other solutions; I don't know. In any case, I will only touch on this briefly.

In this example, compose takes two procs and returns a new proc which, when called, calls the first proc and passes its result into the second proc.

```
def compose proc1, proc2
  Proc.new do |x|
    proc2.call(proc1.call(x))
  end
end

squareIt = Proc.new do |x|
  x * x
end

doubleIt = Proc.new do |x|
  x + x
end

doubleThenSquare = compose doubleIt, squareIt
squareThenDouble = compose squareIt, doubleIt

puts doubleThenSquare.call(5)
puts squareThenDouble.call(5)
```

```
100
50
```

Notice that the call to proc1 had to be inside the parentheses for proc2 in order for it to be done first.

Passing Blocks (Not Procs) into Methods

Ok, so this has been sort of academically interesting, but also sort of a hassle to use. A lot of the problem is that there are three steps you have to go through (defining the method, making the proc, and calling the method with the proc), when it sort of feels like there should only be two (defining the method, and passing the block right into the method, without using a proc at all), since most of the time you don't want to use the proc/block after you pass it into the method. Well, wouldn't you know, Ruby has it all figured out for us! In fact, you've already been doing it every time you use iterators.

I'll show you a quick example first, then we'll talk about it.

```
class Array
  def eachEven(&wasABlock_nowAProc)
    # We start with "true" because arrays start with 0, which is even.
    isEven = true

    self.each do |object|
      if isEven
        wasABlock_nowAProc.call object
      end

      isEven = (not isEven)  # Toggle from even to odd, or odd to even.
    end
  end
end

['apple', 'bad apple', 'cherry', 'durian'].eachEven do |fruit|
  puts 'Yum!  I just love '+fruit+' pies, don\'t you?'
end

# Remember, we are getting the even-numbered elements
# of the array, all of which happen to be odd numbers,
# just because I like to cause problems like that.
[1, 2, 3, 4, 5].eachEven do |oddBall|
  puts oddBall.to_s+' is NOT an even number!'
end
```

```
Yum!  I just love apple pies, don't you?
Yum!  I just love cherry pies, don't you?
1 is NOT an even number!
3 is NOT an even number!
5 is NOT an even number!
```

So to pass in a block to eachEven, all we had to do was stick the block after the method. You can pass a block into any method this way, though many methods will just ignore the block. In order to make your method not ignore the block, but grab it and turn it into a proc, put the name of the proc at the end of your method's parameter list, preceded by an ampersand (&). So that part is a little tricky, but not too bad, and you only have to do that once (when you define the method). Then you can use the method over and over again, just like the built-in methods which take blocks, like each and times. (Remember 5.times do...?)

If you get confused, just remember what eachEven is supposed to do: call the block passed in with every other element in the array. Once you've written it and it works, you don't need to think about what it's actually doing under the hood ("which block is called when??"); in fact, that's exactly why we write methods like this: so we never have to think about how they work again. We just use them.

I remember one time I wanted to be able to time how long different sections of a program were taking. (This is also known as profiling the code.) So I wrote a method which takes the time before running the code, then it runs it, then it takes the time again at the end and figures out the difference. I can't find the code right now, but I don't need it; it probably went something like this:

```
def profile descriptionOfBlock, &block
  startTime = Time.now

  block.call
  duration = Time.now - startTime

  puts descriptionOfBlock+':  '+duration.to_s+' seconds'
end

profile '25000 doublings' do
  number = 1

  25000.times do
    number = number + number
  end

  # Show the number of digits in this HUGE number.
  puts number.to_s.length.to_s+' digits'
end

profile 'count to a million' do
  number = 0

  1000000.times do
    number = number + 1
  end
end
```

```
7526 digits
25000 doublings:  0.109064 seconds
count to a million:  0.124997 seconds
```

How simple! How elegant! With that tiny method, I can now easily time any section of any program that I want to; I just throw the code in a block and send it to profile. What could be simpler? In most languages, I would have to explicitly add that timing code (the stuff in profile) around every section which I wanted to time. In Ruby, however, I get to keep it all in one place, and (more importantly) out of my way!

A Few Things to Try

Grandfather Clock. Write a method which takes a block and calls it once for each hour that has passed today. That way, if I were to pass in the block do puts 'DONG!' end, it would chime (sort of) like a grandfather clock. Test your method out with a few different blocks (including the one I just gave you). Hint: You can use Time.now.hour to get the current hour. However, this returns a number between 0 and 23, so you will have to alter those numbers in order to get ordinary clock-face numbers (1 to 12).

Program Logger. Write a method called log, which takes a string description of a block and, of course, a block. Similar to doSelfImportantly, it should puts a string telling that it has started the block, and another string at the end telling you that it has finished the block, and also telling you what the block returned. Test your method by sending it a code block. Inside the block, put another call to log, passing another block to it. (This is called nesting.) In other words, your output should look something like this:

```
Beginning "outer block"...
Beginning "some little block"...
..."some little block" finished, returning:  5
Beginning "yet another block"...
..."yet another block" finished, returning:  I like Thai food!
..."outer block" finished, returning:  false
```

Better Logger. The output from that last logger was kind of hard to read, and it would just get worse the more you used it. It would be so much easier to read if it indented the lines in the inner blocks. To do this, you'll need to keep track of how deeply nested you are every time the logger wants to write something. To do this, use a global variable, a variable you can see from anywhere in your code. To make a global variable, just precede your variable name with $, like these: $global, $nestingDepth, and $bigTopPeeWee. In the end, your logger should output code like this:

```
Beginning "outer block"...
  Beginning "some little block"...
    Beginning "teeny-tiny block"...
    ..."teeny-tiny block" finished, returning:  lots of love
  ..."some little block" finished, returning:  42
  Beginning "yet another block"...
  ..."yet another block" finished, returning:  I love Indian food!
..."outer block" finished, returning:  true
```

Well, that's about all you're going to learn from this tutorial. Congratulations! You've learned a lot! Maybe you don't feel like you remember everything, or you skipped over some parts... really, that's just fine. Programming isn't about what you know; it's about what you can figure out. As long as you know where to find out the things you forgot, you're doing just fine. I hope you don't think that I wrote all of this without looking things up every other minute! Because I did. I also got a lot of help with the code which runs all of the examples in this tutorial. But where was I looking stuff up, and who was I asking for help? Let me show you...Blocks and Procs