**Procs**

Now that we have reviewed the basics of using methods that accept blocks, how can we write our own custom methods that utilize blocks? We'll need knowledge of **Procs**. Procs and blocks go hand in hand. A proc is an object that contains a block. We need procs because they allow us to save blocks to variables so we can manipulate them in our code.

**Creating a Proc**

We cannot pass a block into a method, but we can turn the block into a proc and pass that proc into a method to be executed. Think of a proc as a wrapper around a block! Let's take a look at how we can create a proc manually. We'll need to use Proc.new and give it the block to wrap up:

doubler = Proc.new { |num| num \* 2 }

p doubler # #<Proc:0x00007f9a8b36b0c8>

Printing the proc gives a somewhat cryptic output,"<Proc:0x00007f9a8b36b0c8>". This is the visual representation of the proc object, but it's not too important. Since a proc is a normal ruby object, we are free to save the proc to a variable, doubler. This is an important fact about procs. In comparison, we *cannot* save a plain block to a variable:

# incorrect

doubler = { |num| num \* 2 } # SyntaxError: unexpected '}'

**Calling a Proc**

Now that we have a proc, let's execute the code it contains. To do this, we need to use the Proc#call method:

doubler = Proc.new { |num| num \* 2 }

p doubler.call(5) # => 10

p doubler.call(7) # => 14

When calling the proc, we can pass in any arguments the block expects. We can also call the proc as many times as we please! The Proc#call method will evaluate to the last line of code executed within the block. Let's take a look at this with a multiline block:

is\_even = Proc.new do |num|

if num % 2 == 0

true

else

false

end

end

p is\_even.call(12) # => true

When we do is\_even.call(12), the result is true because the last executed expression in the block is true.

**Passing a Proc to a Method**

Since a proc has an advantage of being an object, we can pass this proc object into a method. Let's say we had this method:

def add\_and\_proc(num\_1, num\_2, prc)

sum = num\_1 + num\_2

p prc.call(sum)

end

The add\_and\_proc method will take in two numbers and a proc. It will call the proc, giving it the sum of the two numbers, and finally print the result of the proc. Let's see it in action. To use this method, we'll also need a proc to pass in:

def add\_and\_proc(num\_1, num\_2, prc)

sum = num\_1 + num\_2

p prc.call(sum)

end

doubler = Proc.new { |num| num \* 2 }

add\_and\_proc(1, 4, doubler) # => 10

square = Proc.new { |num| num \* num }

add\_and\_proc(3, 6, square) # => 81

negate = Proc.new { |num| -1 \* num }

add\_and\_proc(3, 6, negate) # => -9

Notice that we can pass different blocks/procs into the method to really vary its behavior. Now our add\_and\_proc method is pretty versatile. The only knock against this code is that we have to repeatedly wrap each block in a proc using Proc.new. Fret not! Ruby affords us a way to automatically convert a block into a proc when passed into method. Let's compare the two ways, side by side:

# Version 1: manual, proc accepting method

def add\_and\_proc(num\_1, num\_2, prc)

sum = num\_1 + num\_2

p prc.call(sum)

end

doubler = Proc.new { |num| num \* 2 }

add\_and\_proc(1, 4, doubler) # => 10

# Version 2: automatic conversion from block to proc

def add\_and\_proc(num\_1, num\_2, &prc)

sum = num\_1 + num\_2

p prc.call(sum)

end

add\_and\_proc(1, 4) { |num| num \* 2 } # => 10

Take a moment to compare the two methods and how we call them. In version 2, it seems that we only pass two number arguments to the method, but the definition lists 3 arguments. This is because the third argument, prc, will refer to the block we pass! By using the & operator on the third parameter, ruby knows to automatically convert the block into proc for us.

Because of the &prc parameter we must always pass a block into add\_and\_proc, we can no longer pass in a proc because a conversion from block to proc must take place.

def add\_and\_proc(num\_1, num\_2, &prc)

sum = num\_1 + num\_2

p prc.call(sum)

end

doubler = Proc.new { |num| num \* 2 }

add\_and\_proc(1, 4, doubler) # ArgumentError: wrong number of arguments (given 3, expected 2)

Here are two general tips that you can use to reason out whether a method expects a proc or a block.

def my\_method(prc)

# ...

end

# By looking at the parameter `prc` we know that my\_method must be passed a proc:

my\_proc = Proc.new { "I'm a block" }

my\_method(my\_proc)

def my\_method(&prc)

# ...

end

# By looking at the parameter `&prc` we know that my\_method must be passed a block,

# because & denotes conversion from block to proc here:

my\_method { "I'm a block" }

**Using &**

We already saw how & can be used to convert a *block into a proc*. But it can also be used for the opposite, that is, convert a *proc into a block*. We know, we know, that sounds hopelessly confusing. Let's show it off using our last example. We already established that this code can only accept blocks now, in this context &prc is converting a block to a proc. If we try to pass our method the doubler proc, we will get an error. This is because doubler is a proc, not a block!

def add\_and\_proc(num\_1, num\_2, &prc)

sum = num\_1 + num\_2

p prc.call(sum)

end

doubler = Proc.new { |num| num \* 2 }

add\_and\_proc(1, 4, doubler) # Error

However, we can use & again to convert a proc to a block. In other words, if doubler is a proc, then &doubler is a block:

def add\_and\_proc(num\_1, num\_2, &prc)

sum = num\_1 + num\_2

p prc.call(sum)

end

doubler = Proc.new { |num| num \* 2 }

add\_and\_proc(1, 4, &doubler) # => 10

Since & either turns a *block into a proc* or *proc into a block*, here's a rule you can use to identify what is happening. It all depends on context: when we see & in the parameters for a method definition we know it will convert a block to a proc and when we see & in the arguments for a method call we know it will convert a proc to a block. Another give away is that doubler is most certainly already a proc since we used Proc.new, so &doubler converts that proc into a block.

**Another Example**

The dual function of & is the biggest point of confusion for blocks and procs so let's step through another example in familiar territory. We know that map is a built-in method that *must be given a block*:

[1,2,3].map { |num| num \* 2 } # => [2, 4, 6]

However, if we have a proc and want to use it with map, we can use & to convert it to a block:

doubler = Proc.new { |num| num \* 2 }

[1,2,3].map(&doubler) # => [2, 4, 6]