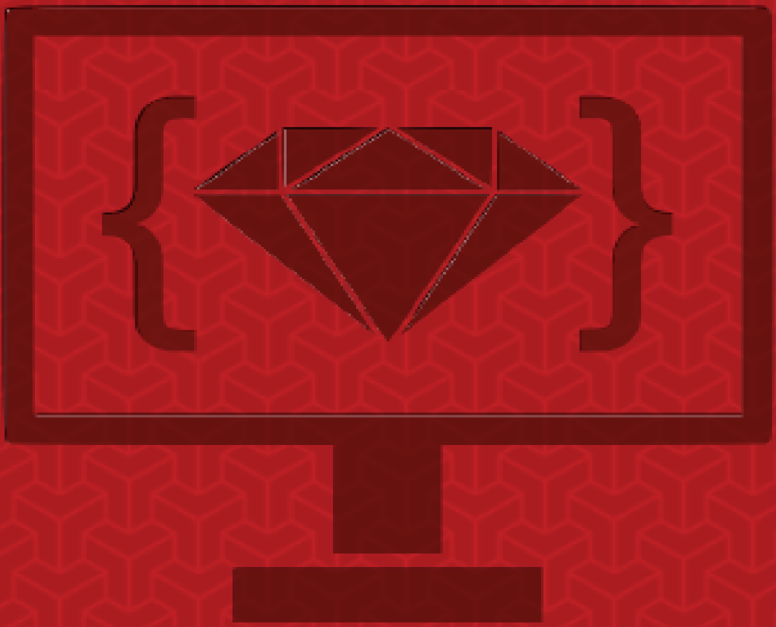
EVERYTHING YOU NEED TO *MASTER* BLOCKS, PROCS, AND LAMBDAS.



THE **RUBY CLOSURES** BOOK

BENJAMIN TAN WEI HAO

The Ruby Closures Book

Everything You Need to Master Blocks, Procs and Lambdas

Benjamin Tan Wei Hao

This book is for sale at http://leanpub.com/therubyclosuresbook

This version was published on 2016-08-08

To Hui Ling, the love of my life.

Contents

Welcome!

Welcome to the Ruby Closures book! You have taken a very important step into become a better Ruby developer.

Why this book?

I have been a Ruby programmer for quite a number of years, but until recently, I have been blissfully ignorant of one of Ruby's most powerful features. That is, until one day, I saw a co-worker casually write a method that took in a block. I realized immediately that I had a major gap in my Ruby knowledge.

That incident left me unsure of my Ruby skills and knowledge. I decided to learn as much as I could about Ruby's blocks. That pursuit also led me to procs and lambdas.

It didn't take long to discover that blocks, procs, and lambdas are Ruby's implementation of closures – hence the title of this book. The word "closure" sounds scary concept, and indeed, for most parts of my career, I avoided learning about it. Not this time. I got my hands on every book, conference video, screencast, and blog post that barely mentioned "closure" and dove in.

This book is the essence of everything I have learned.

Who is this book for?

This book is written for the Ruby (duh!) developer in mind. You are already comfortable with programming in Ruby but are by no means an expert.

This book will *not* go through the basics of Ruby, except when it comes it relates to the subject matter. There are many excellent books and resources that will do a much better job.

How to use this book

The overarching goal is to make you a better Ruby developer. To be a better developer, you have to learn how to read and write good code. This book is designed specifically to help you read and write good code that uses Ruby's closures.

I have strived to make this book as understandable as possible. I have used numerous diagrams to make the explanations more concrete. You will find that I repeat concepts, although slightly different each time. I find that I learn much better this way, and I hope that works for you too.

Obviously, I shouldn't be doing all the hard work. You will learn better when you try out the code samples on `irb`. More importantly, you will find exercises at the back of every chapter – **DO THEM**. Simply reading about code will not magically transform you into a better developer. Suggested solutions to all the exercises are included at the back of the book, so you will not be left in the dark if you are stuck.

One thing that have helped me improve as a developer is, as Gregory Brown from Practicing Ruby[1] puts it, build *cheap counterfeits*. That is, we will implement our own versions of built-in features and scaled down versions of popular libraries, purely for educational purposes.

Structure of the book

This book is broken into four parts.

The first part covers the basics of closures and sets the groundwork for the rest of the book.

The second part dives into blocks, where we start with the basics and build our very own `Enumerable` and `Enumerator` in Ruby. You will learn about the patterns that involve blocks, and how they are used in real-world code.

The third part is dedicated to *procs* and *lambdas*. You will learn how to create and use them. You will also appreciate the special relationship that blocks, procs and lambdas have with each other. Doing the exercises will change how you write Ruby code.

Finally, the fourth part is when we put what we have learned into good use. We begin with building our own test framework à la RSpec. This will be our first taste in building a domain specific language, or DSL for short. We then extend our enumerables example and attempt to build *lazy* enumerables. We then tackle something slightly more complex by building a minimal Promises library in Ruby. If all that has left you scratching your head, no worries. All will be explained in due course.

About Ruby versions

I used Ruby 2.2.X to develop the examples. Ruby 1.9.X and 2.X should have no problems. If you have Ruby 1.8, you are on your own. Earlier versions of Ruby have confusing behavior (with regard to procs) that was fixed in later versions. Also, I'm too lazy (and you are too smart) to show you how to install Ruby.

Let's do this!

I hope you are excited and ready to go! Warm up your brain and fire up your console, because we are diving straight into the fascinating world of closures.

[1]https://practicingruby.com/articles/patterns-for-building-excellent-examples

Proctology

In this chapter, we look at `procs` and `lambdas`, both of which come from the `Proc` class. We look at the `Procs` class first and explore multiple ways of calling Procs.

Ever wondered how `["o","h","a","i"].map(&:upcase)` expands to `["o","h","a","i"].map { |c| c.upcase) }`? We will learn how this is implemented in this chapter!

Next, we examine the difference between a `proc` and a `lambda`. If you were ever confused between these two, this section is for you.

We then look at currying, a functional programming concept. Although its practical uses (with respect to Ruby programming) are pretty limited, it is still a fun topic to explore.

Procs and blocks are closely related. In fact, be prepared to see some un-Ruby-like code! Procs and lambdas are ubiquitous in Ruby code. Here, we will see some real-world code that makes good use of `procs` and `lambdas`.

Hopefully by then `procs` and `lambdas` do not seem mysterious anymore, and a tool that you can readily use at your disposal.

How does Symbol#to_proc work?

`Symbol#to_proc` is one of the finest examples of the flexibility and beauty of Ruby. This syntax sugar allows us to take a statement such as:

```
words.map { |s| s.length }
```

and turn it into:

```
words.map &:length
```

Let’s unravel this syntactical sleight of hand, by figuring out how this works.

What does the &:symbol do?

How does Ruby even know that it has to call a `to_proc` method, and why is this only specific to the `Symbol` class?

When Ruby sees an `&` and an object – *any* object – **it will try to turn it into a block**. This is simply a form of type coercion.

Take `to_s` for example. We can do `2.to_s`, which returns the string representation of the integer `2`. Similarly, `to_proc` will attempt to turn an object – again, *any* object – into a proc.

Reimplementing Symbol#to_proc

Let's see what this means. Let's create an object, and plop it into `each`:

```
obj = Object.new => #<Object:0x007ff4218761b8>
[1,2,3].map &obj
TypeError: wrong argument type Object (expected Proc)
```

That's awesome! Our error message is telling us exactly what we need to know: it's saying that `obj` is well, an `Object` and not a `Proc`. The fix is simple: the `Object` class must have a `to_proc` method that *returns a proc*. Let's do the simplest thing possible:

```
class Object
  def to_proc
    proc {}
  end
end

obj = Object.new
[1, 2, 3].map &obj #=> [nil, nil, nil]
```

Now when we run this again, we get no errors. Almost there! How can we then access each element, and say, print it? We need to let out `proc` accept a parameter:

```
class Object
  def to_proc
    proc { |x| "Here's #{x}!" }
  end
end

obj = Object.new
[1,2,3].map &obj #=> ["Here's 1!", "Here's 2!", "Here's 3!"]
```

This hints at a possible implementation of `Symbol#to_proc`. Let's start with what we know, and *redefine* `to_proc`:

```
class Symbol
  def to_proc
    proc { |obj| obj }
  end
end
```

We know that in an expression such as

```
words.map &:length
```

is equivalent to

```
words.map { |w| w.length }
```

Here, the symbol *instance* is `:length`. This *value* of the symbol corresponds to the *name* of the method. We have previously found out how to access each yielded object – by making the proc return value in `to_proc` take in an argument.

We want to achieve this effect:

```
class Symbol
  def to_proc
    proc { |obj| obj.length }
  end
end
```

How can we use the name of the symbol to call a method on `obj`? `send` to the rescue! I hereby present you our own implementation of `Symbol#to_proc`:

```
class Symbol
  def to_proc
    proc { |obj| obj.send(self) }
  end
end
```

Here, `self` is the symbol object (`:length` in our example), which is exactly what `#send` expects.

Improving on our Symbol#to_proc

Our initial implementation of `Symbol#to_proc` is naïve. The reason is that we only consider the `obj` in the body of the `proc`, and totally ignore its arguments.

Recall that unlike lambdas, procs are more relaxed when it comes to the number of arguments it is given. We can therefore easily expose this limitation.

First, we return a lambda instead of a proc in `to_proc`. Recall that a lambda is a proc, so everything should work as per normal:

```
class Symbol
  def to_proc
    lambda { |obj| obj.send(self) }
  end
end

words = %w(underwear should be worn on the inside)
words.map &:length # => [9, 6, 2, 4, 2, 3, 6]
```

Since we know lambdas are picky when it comes to the number of arguments, is there a method that requires *two* arguments? Of course: `inject/reduce`. The usual way of writing `reduce` is:

```
[1, 2, 3].inject(0) { |result, element| result + element } # => 6
```

As you can see, the block in `inject` takes two arguments. Let's see how our implementation does, by using the `&:symbol` notation:

```
[1, 2, 3].inject(&:+)
```

Here's the error we get:

```
ArgumentError: wrong number of arguments (2 for 1)
from (irb):10:in `block in to_proc'
from (irb):14:in `each'
from (irb):14:in `inject'
...
```

We can now clearly see that we are missing an argument. The lambda currently accepts only 1 argument, but what it received was 2 arguments. We need to allow the lambda to take in arguments:

```
class Symbol
  def to_proc
    lambda { |obj, args| obj.send(self, *args) }
  end
end

[1, 2, 3].inject(&:+) # => 6
```

Now it works as expected! We use the splat operator (that's the `*` in `*args`) to support a variable number of arguments. We have one problem though. This doesn't work anymore:

```
words = %w(underwear should be worn on the inside)
words.map &:length # => [9, 6, 2, 4, 2, 3, 6]

ArgumentError: wrong number of arguments (1 for 2)
from (irb):3:in `block in to_proc'
from (irb):8:in `map'
...
```

There are two ways to fix this. First, we can give `args` a default value:

```
class Symbol
  def to_proc
    lambda { |obj, args=nil| obj.send(self, *args) }
  end
end

words = %w(underwear should be worn on the inside)
words.map &:length # => [9, 6, 2, 4, 2, 3, 6]

[1, 2, 3].inject(&:+) # => 6
```

Or, we can just make it a `Proc` again:

```
class Symbol
  def to_proc
    proc { |obj, args| obj.send(self, *args) }
  end
end

words = %w(underwear should be worn on the inside)
words.map &:length # => [9, 6, 2, 4, 2, 3, 6]

[1, 2, 3].inject(&:+) # => 6
```

This is one of the rare cases when being less picky about arity helps. Now that you know how `Symbol#to_proc` works, it's time to work on the exercises!

Exercises

1. Reimplement Symbol#to_proc: Now that you have seen how `Symbol#to_proc` is implemented, you should have a go at it yourself.
2. Class initialisation with #to_proc:

Consider this behavior:

```
class SpiceGirl
  def initialize(name, nick)
    @name = name
    @nick = nick
  end

  def inspect
    "#{@name} (#{@nick} Spice)"
  end
end

spice_girls = [["Mel B", "Scary"], ["Mel C", "Sporty"],
["Emma B", "Baby"], ["Geri H", "Ginger",], ["Vic B", "Posh"]]

p spice_girls.map(&SpiceGirl)
```

returns:

```
[Mel B (Scary Spice), Mel C (Sporty Spice),
Emma B (Baby Spice), Geri H (Ginger Spice), Vic B (Posh Spice)]
```

This example demonstrates how `to_proc` can be used to initialize a class. Implement this!

Solutions

1. The answer is exactly the one in the chapter:

```
class Symbol
  def to_proc
    proc { |obj, args| obj.send(self, *args) }
  end
end
```

1. Since we are interested in adding behavior to *object initialization* , it therefore makes sense to implement `to_proc` within the `Class` class. Here's a possible implementation:

```
class Class
  def to_proc
    proc { |args| new(*args) }
  end
end
```

Since we are creating objects with arrays, each array element is treated as a single object, therefore the `proc` takes a single argument.

Next, we use the splat operator to convert the array into method arguments, and pass it into `new`, which then calls the initializer.