Extracted from:

Mastering Ruby Closures

A Guide to Blocks, Procs, and Lambdas

The Pragmatic Bookshelf

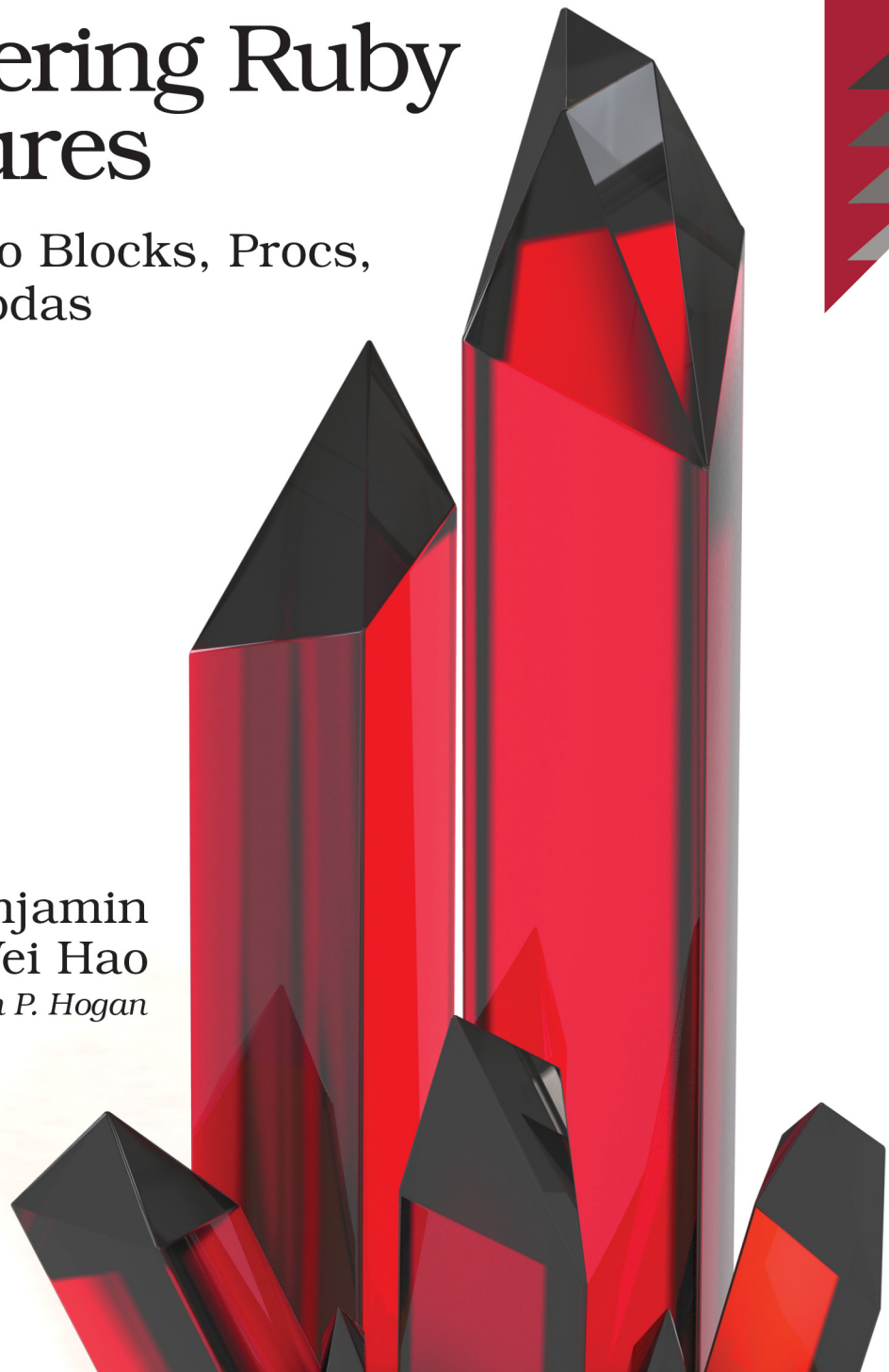
Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic Press

Mastering Ruby Closures

A Guide to Blocks, Procs, and Lambdas



Benjamin Tan Wei Hao

edited by Brian P. Hogan

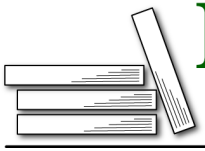
Mastering Ruby Closures

A Guide to Blocks, Procs, and Lambdas

Benjamin Tan Wei Hao

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Susannah Davidson Pfalzer
Development Editor: Brian P. Hogan
Copy Editor: Nicole Abramowitz
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-261-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—August 2017

CHAPTER 3

The Power of Procs and Lambdas

Recall that blocks by themselves are not objects—they cannot exist by themselves. In order to do anything interesting with a block, you need to pass a block into a method.

Procs have no such restrictions, because they *are* objects. They allow you to represent a block of code (anything between a `do ... end`) as an object. Some languages call these anonymous functions, and indeed, they do play the part.

Procs are ubiquitous in real-world Ruby code, although chances are, you might not be using them that much. Through the examples, you'll learn how to use them effectively in your own code.

Ruby also uses Procs to perform some really nifty tricks. For example, have you ever wondered how `["o","h","a","i"].map(&:upcase)` expands to `["o","h","a","i"].map { |c| c.upcase) }`? By the end of this chapter, you'll understand the mechanics of how Ruby performs this sleight of hand.

Procs also assume another form: lambdas. While they serve similar functions (pun intended!), it is also important to learn about their differences, so that you will know when to use which at the right time.

One technique that Procs enable but hasn't seen very wide use is currying, a functional programming concept. Although its practical uses (with respect to Ruby programming) are pretty limited, it's still a fun topic to explore.

By the end of this chapter, you should be comfortable enough to use Procs and lambdas in your own code and make them an indispensable part of your Ruby toolbox.

Procs and the Four Ways of Calling Them

Unlike the language named after a certain serpent, Ruby embraces TMTOWTDI (pronounced as *Tim Toady*), or *There's more than one way to do it*. The calling of Procs is a wonderful example. In fact, Ruby gives you *four* different ways:

1. Proc#call(args)
2. .(args)()
3. Threequals
4. Lambdas

Fire up irb. Let's begin by creating a very simple Proc:

```
>> p = proc { |x, y| x + y }
=> #<Proc:0x007ffb12907940@(irb):1>
```

There are two things to notice here. First, the return value tells you that a Proc has been created. Second, Ruby provides a shorthand to create Procs. This is really a method in the Kernel class:

```
>> p = Kernel.proc { |x, y| x + y }
=> #<Proc:0x007ffb12907940@(irb):1>
```

Of course, since Proc is just like any other class, you can create an instance of it the usual way:

```
>> p = Proc.new { |x, y| x + y }
=> #<Proc:0x007ffb12907940@(irb):1>
```

Now you know how to create a Proc. Time to make it do some work. The first way is to use Proc#call(args)():

```
>> p = proc { |x,y| x + y }

>> p.call("oh", "ai")
=> "ohai"

>> p.call(4, 2)
=> 6
```

In fact, this is my preferred way of invoking Procs because it conveys the intent of invocation much better than the alternatives, which are presented next.

Ruby provides a shorthand for the call(args)() method: .(args)(). Therefore, the previous example could have been rewritten as such:

```
>> p = proc { |x,y| x + y }
>> p.("oh", "ai")
```

```
>> p.(4, 2)
```

Here's an interesting Ruby tidbit. Turns out, the .() syntax works across *any* class that implements the call() method. For example, here's a class with only the call() method:

```
class Carly
  def call(who)
    "call #{who}, maybe"
  end
end

c = Carly.new
c.("me") # => "call me, maybe"
```

You should avoid using .() if you can, because this could potentially confuse other people who might not be familiar with the syntax.

Ruby has an even quirkier syntax for invoking Procs:

```
p = proc { |x,y| x + y }
p === ["oh", "ai"]
```

The === operator is also known as the *threequals* operator. This operator makes it possible to use a Proc in a case statement. Look at the following code:

```
even = proc { |x| x % 2 == 0 }
case 11
  when even
    "number is even"
  else
    "number is odd"
end
```

Here, even, when given a number, returns true or false depending on the case statement. For example:

```
>> even = proc { |x| x % 2 == 0 }
>> even === 11
=> false
>> even === 10
=> true
```

Note that invoking a Proc that expects a single argument this way is incorrect and results in a confusing error message:

```
>> even = proc { |x| x % 2 == 0 }
>> even === [11]
NoMethodError: undefined method `%' for [11]:Array
        from (irb):1:in `block in irb_binding'
```

Next, let's look at lambdas and how they relate to Procs.