Extracted from:

Mastering Ruby Closures

A Guide to Blocks, Procs, and Lambdas

The Pragmatic Bookshelf

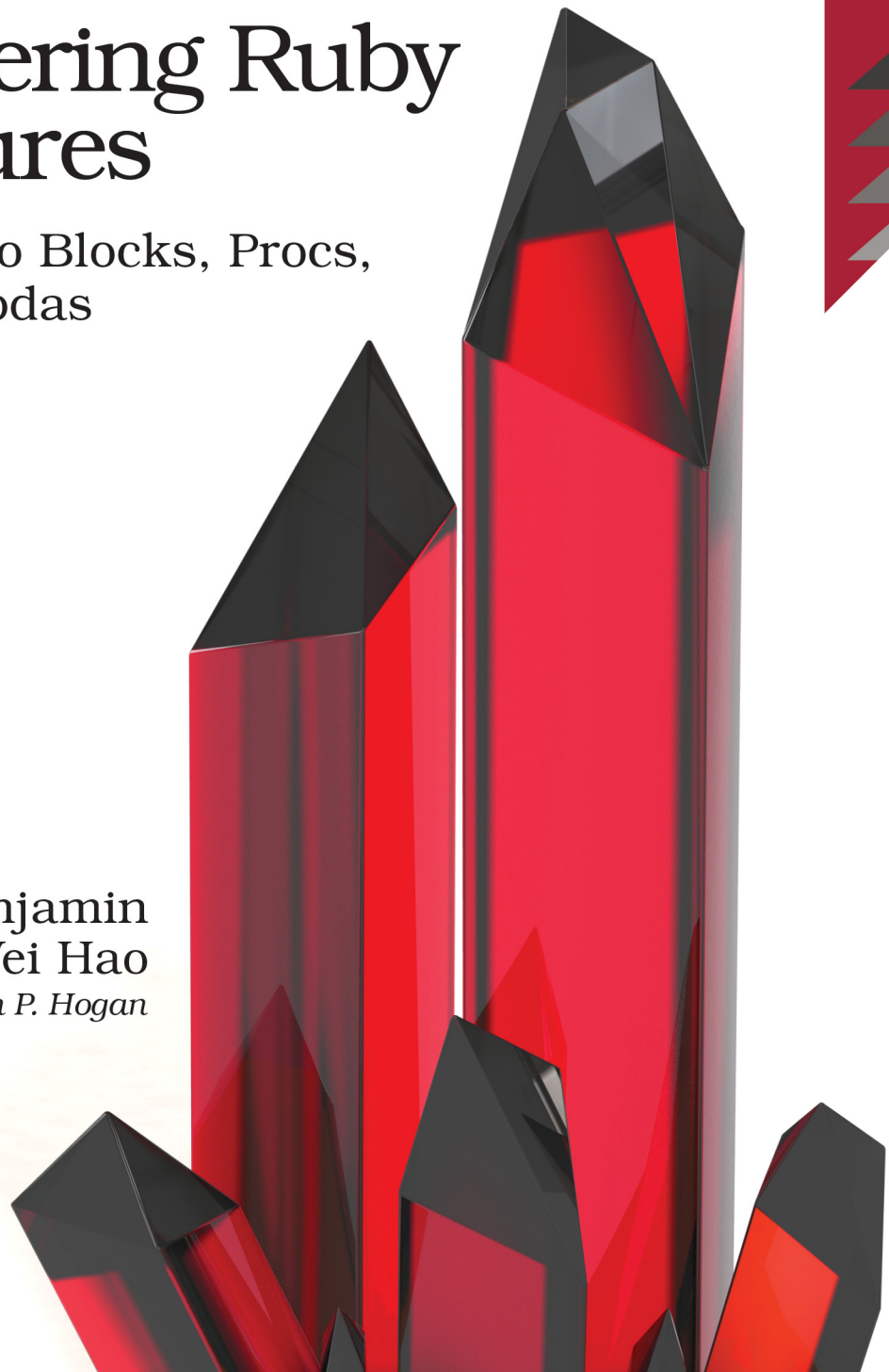
Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic Press

Mastering Ruby Closures

A Guide to Blocks, Procs, and Lambdas



Benjamin Tan Wei Hao

edited by Brian P. Hogan

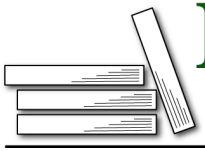
Mastering Ruby Closures

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

Blocks as Closures and Block Local Variables

In Ruby, blocks act like anonymous functions. After all, blocks carry a bunch of code, to be called only when yielded. A block also carries around the context in which it was defined:

```
def chalkboard_gag(line, repetition)
  repetition.times { |x| puts "#{x}: #{line}" }
end

chalkboard_gag("I will not surprise the incontinent", 3)
```

This returns:

```
0: I will not surprise the incontinent
1: I will not surprise the incontinent
2: I will not surprise the incontinent
```

What's the free variable here? It is line. That's because line is *not* a *block local variable*. Instead, it needs access to the *outer* scope until it reaches the arguments of chalkboard_gag.

The behavior of the preceding code shouldn't be too surprising, because it seems rather intuitive. Imagine now if Ruby *didn't* have closures. The block then wouldn't be able to access the arguments. You can simulate this by declaring line to be a block local variable by preceding it with a semicolon:

```
def chalkboard_gag(line, repetition)
  repetition.times { |x; line| puts "#{x}: #{line}" }
end
```

Block local variables are declared after the semicolon. Now line in the block no longer refers to the arguments of chalkboard_gag:

```
0:
1:
2:
```

Block local variables are a way to ensure that the variables within a block don't override another *outer* variable of the same name. This essentially circumvents the variable capturing behavior of a closure.

Here's another example:

```
x = "outside x"
1.times { x = "modified from the outside block" }

puts x # => "modified from the outside block"
```

In this example, the outer x is modified by the block, because the block *closes over* the outer x, and therefore has a reference to it. If we want to prevent this behavior, we could do this:

```
x = "outside x"
1.times { |;x| x = "modified from the outside block" }

puts x # => "outside x"
```

That covers most of what there is to know about block variables. In the next section, we take a look at different block patterns that are often seen in Ruby code. These patterns cover enumeration, resource management, and object initialization.

Next, let's look at some patterns that use blocks, starting with enumeration.

Block Pattern #1: Enumeration

You may have fallen in love with Ruby because of the way it does enumeration:

```
>> %w(look ma no for loops).each do |x|
>>   puts x
>> end
look
ma
no
for
loops
=> ["look", "ma", "no", "for", "loops"]
```

Besides being very expressive, enumeration using blocks is more concise and less error-prone. It is concise because the block captures exactly *what* we want to do with each element (printing it out to the console). It is less error-prone compared to traditional for loops because it does away with indices that are prone to the infamous off-by-one error.

You should be familiar with this way of iterating over a collection, such as an Array. What's interesting is *how* these methods are *implemented* under the hood.

Going through the process of building your own implementation will give you a much deeper understanding of how methods and blocks work.

Implementing Fixnum#times

While it's not surprising that Ruby is an object-oriented language, the extent of "object-orientedness" often surprises newcomers to Ruby. For example,

most wouldn't associate a number with the notion of an object. However, Ruby begs to differ by making code like this possible:

```
>> 3.times { puts "Eat my shorts!" }
Eat my shorts!
Eat my shorts!
Eat my shorts!
=> 3
```

How is this possible? The answer is two-fold. First, 3 is an object of the Fixnum class. Second, the Fixnum#times() method is what makes the preceding code possible.

What can we say about the Fixnum#times() method? Well, it executes the block exactly *three* times. This information is taken from the *instance* of the Fixnum, 3. This detail is important, as you will soon see.

What can we say about the parameters of the block? Well, not much, since the block doesn't take any parameters. Let's implement Fixnum#times(). Additionally, we will assume that each() doesn't exist.

Create a file called fixnum_times.rb. Fill in an initial implementation like so:

```
class Fixnum
  def times
    puts "This does nothing yet!"
  end
end
```

Thanks to Ruby's open classes, we have now just overridden the default version of Fixnum#times() and replaced it with our own (currently non-working) one. Load the file in irb using the following command:

```
$ irb -r ./fixnum_times.rb
```

Let's try this out:

```
>> 3.times { puts "Eat my shorts!" }
puts "This does nothing yet!"
=> nil
```

For now, nothing happens since we have overridden the default Fixnum#times() method with our empty implementation. Remember that we imposed the constraint that we cannot use Array#each()? The reason is that would make things too easy for us. We can fall back to a while loop:

blocks/fixnum_times.rb

```
class Fixnum
  def times
    x = 0
```

```
    while x < self
      x += 1
      yield
    end
    self
  end
end
```

Now, redo the steps with the updated code:

```
% irb -r ./fixnum_times.rb
>> 3.times { puts "Eat my shorts!" }
Eat my shorts!
Eat my shorts!
Eat my shorts!
=> 3
```

Again, self is the Fixnum instance, also known as 3 in our example. In other words, it is using the value of the number to perform the same number of iterations. Pretty nifty, if you ask me.

The most important part of the code here is yield. In this example, yield is called without any arguments, which is exactly what the original implementation expects. The return value of the times() method is the number itself, hence self is returned at the end of the method.

Let's keep up the momentum and implement Array#each().