Extracted from:

Mastering Ruby Closures

A Guide to Blocks, Procs, and Lambdas

The Pragmatic Bookshelf

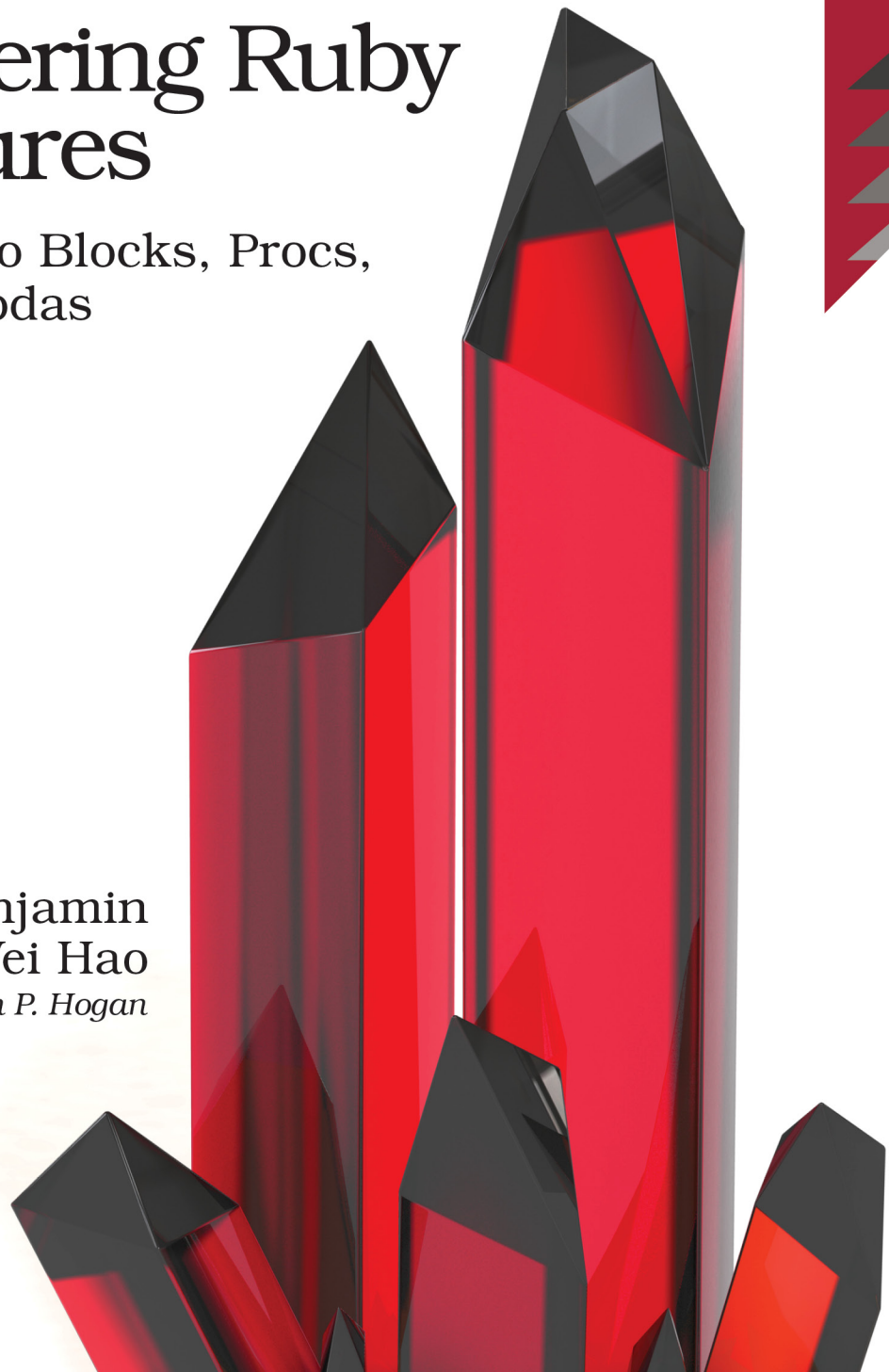
Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic Press

Mastering Ruby Closures

A Guide to Blocks, Procs, and Lambdas



Benjamin Tan Wei Hao

edited by Brian P. Hogan

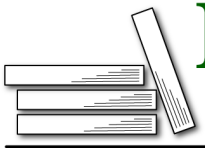
Mastering Ruby Closures

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 2

Beautiful Blocks

Blocks are effectively a type of closure. Blocks capture pieces of code that can be passed into methods to be executed later. In a sense, they act like anonymous functions.

Blocks are ubiquitous in Ruby and are one of the defining characteristics of Ruby—you can immediately tell that it's Ruby code once you see the familiar `do ... end` or curly braces. It's virtually impossible to write any meaningful Ruby program without using blocks. In order to understand and appreciate real-world Ruby code, it's imperative that you understand how blocks work and how to use them.

Rubyists often wax lyrical about blocks—and for good reason. They are a powerful language construct that leads to beautiful and succinct code. They allow you to modify specific behavior without changing the general pieces of your code. This means that you get to do more with less code.

Blocks are also instrumental in crafting domain-specific languages, or DSLs. This ability has been exploited to great effect, especially in tools such as Rake and Rails.

There are two main objectives in this chapter. The first is to make sure you understand how blocks are used. In order to do that, you will learn about the `yield` keyword and the `block_given?()` method by writing your own methods that take blocks as input. You will also learn what block variables are, and their relationship to blocks acting as closures.

The second objective is to get you well acquainted with the various ways that blocks are used in Ruby—*block patterns*, if you will. You will write code that enumerates a collection such as an array or a hash. Having the skills to use blocks in conjunction with the classes in the Ruby Standard Library will save

you precious time, especially when you start to realize how blocks can make methods extremely versatile.

However, blocks have a lot more to offer than going through the elements of a collection. Other block patterns that are pervasive in real-world Ruby code include resource management, object initialization, and the abstraction of pre- and post-processing. You will be writing code that explores each of these patterns in the sections that follow.

Along the way, you will get to work with some meta-programming goodness and learn the secret to creating Ruby DSLs.

By the end of this chapter, you will gain a deeper appreciation of blocks and understand how to use them effectively in your own code. You will be confident in writing your own code that uses blocks. You will also have an understanding of how DSLs are built in Ruby, and you won't be intimidated when you look at a foreign-looking DSL.

Separating the General from the Specific

The ability to encapsulate behavior into blocks and pass it into methods is an extremely useful programming technique. This lets you separate the general and specific pieces of your code. Open irb and let's explore what this separation of concerns looks like.

Suppose you have a range of numbers from 1 to 20, and you're interested in only getting the even numbers. In Ruby, this is how you can do it:

```
>> Array(1..20).select { |x| x.even? }
=> [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Later, you decide that the list is too big and you want to add another condition: the even numbers must also be greater than 10:

```
>> Array(1..20).select { |x| x.even? and x > 10 }
=> [12, 14, 16, 18, 20]
```

Notice that the only code that you had to change was contained within the blocks. That is, the actual "business logic" piece. You didn't have to implement your own special version of Array#select() in order to cope with a change in requirements. This also comes up pretty often with sorting.

Imagine that you're working on an e-commerce site that sells sports shoes, and you want to display a selection of the products on the main page:

```
>> require 'ostruct'

>> catalog = []
```

```
>> catalog << OpenStruct.new(name: 'Nike', qty: 20, price: 99.00)
>> catalog << OpenStruct.new(name: 'Adidas', qty: 10, price: 109.00)
>> catalog << OpenStruct.new(name: 'New Balance', qty: 2, price: 89.00)
```

It's plain to see that we have a pretty wide selection of footwear. Now, the boss wants to display the products by the lowest priced first:

```
>> catalog.sort_by { |x| x.price }
=> [#<OpenStruct name="New Balance", qty=2, price=89.0>,
    #<OpenStruct name="Nike", qty=20, price=99.0>,
    #<OpenStruct name="Adidas", qty=10, price=109.0>]
```

What if now she wants the products with the highest quantity to be displayed first?

```
>> catalog.sort_by { |x| x.qty }.reverse
=> [#<OpenStruct name="Nike", qty=20, price=99.0>,
    #<OpenStruct name="Adidas", qty=10, price=109.0>,
    #<OpenStruct name="New Balance", qty=2, price=89.0>]
```

In both instances, all you had to change was the code in the block. In fact, you didn't have to change the implementation of Enumerable#sort_by(). You were able to trust that the method would do its job provided you gave it a reasonable sorting criteria to work with.

So how is this possible? With yield.