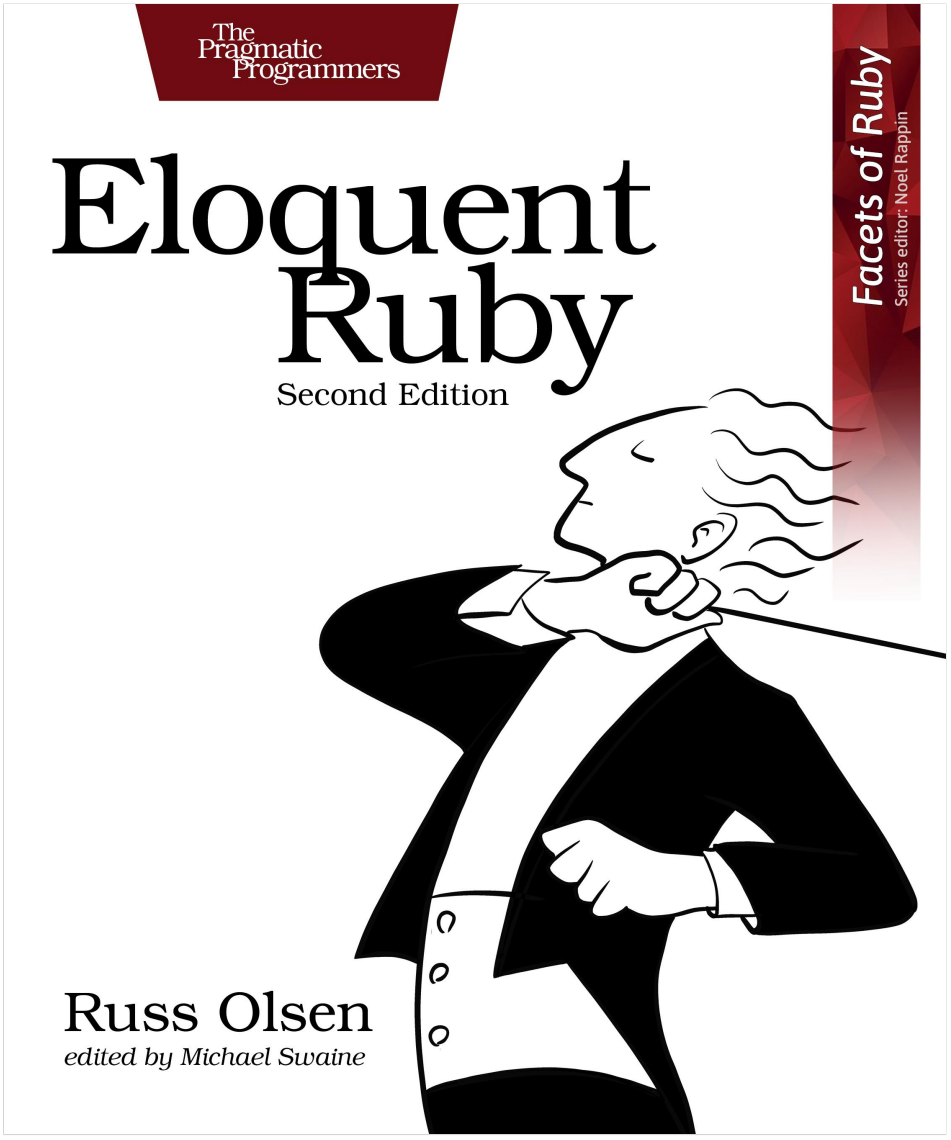

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

In the history of programming languages, operator overloading—the ability to put your own code behind built-in operators like + and *—has had a somewhat checkered career: The very minimalistic C programming language had no room for programmer-defined operators. By contrast, C++ embraced operator overloading as does Python. Programmer-defined operators vanished again with Java and JavaScript, only to rematerialize with Ruby.

All of this to-ing and fro-ing betrays a certain ambivalence about programmer-defined operators on the part of language designers. In this chapter we will look at how you define operators for your Ruby classes and how they might be useful. Along the way we will look into some of the deep pits that wait for you on the road to building your own operators and perhaps gain a little insight into why this is one of those features that falls in and out of vogue. Most importantly, we will talk at some length about how you can dodge those black pits by knowing when *not* to define an operator.

Defining Operators in Ruby

One of the nice things about Ruby is that the language keeps very few secrets from its programmers. Many of the tools used to construct the basic workings of the Ruby programming language are available to the ordinary Joe Programmer. Operators are a good example of this: If you were so inclined, you could implement your own `Float` class and—at least as far as operators like +, -, *, and / are concerned—your handcrafted `Float` would be indistinguishable from the `Float` class that comes with Ruby.

The Ruby mechanism for defining your own operators is straightforward and based on the fact that Ruby translates every expression involving programmer-definable operators into an equivalent expression where the operators are replaced with method calls. So when you say this:

```
sum = first + second
```

What you are really saying is:

```
sum = first.+(second)
```

The second expression sets the variable `sum` to the result of calling the + method on `first`, passing in `second` as an argument. Other than + being a strange-looking method name (it is, however, a perfectly good Ruby method name), the second expression is a simple assignment involving a method call. It is also exactly equivalent to the first expression. The Ruby interpreter is clever about the operator-to-method translation process and will make sure that

the translated expression respects operator precedence and parentheses, so that this:

```
result = first + second * (third - fourth)
```

Will smoothly translate into:

```
result = first.+(second.*(third.-fourth))
```

What this means is that creating a class that supports operators boils down to defining a bunch of instance methods, methods with names like +, -, and *.

To make all of this a little more concrete, let's add an operator to our Document class. Documents aren't the most operator friendly of objects, but we might think of adding two documents together to produce a bigger document:

operators/doc_with_operators.rb

```
class Document
  # Most of the class omitted...
  #
  def +(other)
    Document.new(
      title: title,
      author: author,
      content: "#{content} #{other.content}")
  end
end
```

With this code we can now sum up our documents, so that if we run:

```
doc1 = Document.new(
  title: "Tag Line1",
  author: "Kirk",
  content: "These are the voyages")

doc2 = Document.new(
  title: "Tag Line2",
  author: "Kirk",
  content: "of the star ship ...")

total_document = doc1 + doc2
puts total_document.content
```

We will see the famous tag line:

```
These are the voyages of the star ship ...
```

A Sampling of Operators

Of course, + is not the only operator you can overload. Ruby allows you to define more than twenty five operators for your classes. Among these are the

other familiar arithmetic operators of subtraction (-), division (/), and multiplication (*), along with the modulo operator (%). You can also define your own version of the bit-oriented and (&) or (|), as well as the exclusive or (^) operator.

Another widely defined operator is the bitwise left shift operator, <<. This operator is not popular because Ruby programmers do a lot of bit fiddling; it's popular because it has taken on a second meaning as the concatenation, or "add another one," operator:

```
names = []
names << "Rob"      # names.size is now 1
names << "Denise"   # names.size is now 2
```

Along with binary operators like << and *—which do their thing on a pair of objects—Ruby also lets you define single object, or *unary*, operators. One such unary operator is the ! operator. Here's a somewhat silly unary operator definition for the ! operator:

operators/doc_with_operators.rb

```
class Document
  # Stuff omitted...

  def !
    Document.new(
      title: title,
      author: author,
      content: "It is not true: #{content}")
  end
end
```

This code enables us to have a tongue-in-cheek argument with ourselves. Start with this:

```
favorite = Document.new(
  title: "Favorite",
  author: "Russ",
  content: "Chocolate is best")
```

And !favorite will have a content of:

```
It is not true: Chocolate is best
```

The + and - operators are interesting in that they can be both binary and unary.For example, in the expression -(2+6), the minus sign is a unary operator that changes the sign of the final result while the plus sign is a binary operator that adds the numbers together. But rewrite the expression as +(2-6) and the operator roles are reversed. We saw earlier that defining the + method on your class defines the binary addition operator. To create the unary operator, you need to define a method with the special (and rather arbitrary) name +@.

The same pattern applies to -: The plain old - method defines the binary operator while -@ defines the unary one. Here, for example, are some silly unary operator definitions for our Document class:

```
class Document
  # Most of the class taking a break...

  def +@
    Document.new(
      title: title,
      author: author,
      content: "I am sure that #{@content}")
  end

  def -@
    Document.new(
      title: title,
      author: author,
      content: "I doubt that #{@content}")
  end
end
```

Which lets us do this:

```
unsure = -(+favorite)
```

So we end up with a document containing this wonderful statement of dietary angst:

```
I doubt that I am sure that Chocolate is best
```

Ruby programmers can also define a couple of methods that will make their objects look like arrays or hashes: [] and []=. Although technically these bracketed methods are not operators, the Ruby parser sprinkles some very operator-like syntactic sugar on them: When you say foo[4] you are really calling the [] method on foo, passing in four as an argument. Similarly, when you say foo[4] = 99, you are actually calling the []= method on foo, passing in four and ninety-nine.

You might, for example, define a [] method, a method that will make Document instances look like an arrays of words:

```
class Document
  # Most of the class omitted...
  def [](index)
    words[index]
  end
end
```

If you do add the bracket methods to your object, you will probably also want to put in a size method too, otherwise your users won't be able to tell when they are running off the end of the pseudo-array.