# CHAPTER 11

# Define Operators Respectfully

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, the very expansive C++ embraced operator overloading and included some fairly elaborate facilities to support it. Programmer-defined operators vanished again when Java entered the scene, only to rematerialize again 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 hand-crafted `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:

```
class Document
  # Most of the class omitted...

  def +(other)
    Document.new( title, author, "#{content} #{other.content}" )
```

```
    end
  end
```

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

```
doc1 = Document.new('Tag Line1', 'Kirk', "These are the voyages")
doc2 = Document.new('Tag Line2', 'Kirk', "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 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:[1]

1. This will not work in versions earlier than Ruby 1.9, which greatly expanded the number of operators you can define.

```
class Document
  # Stuff omitted...

  def !
    Document.new( title, author, "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( 'Favorite', 'Russ', 'Chocolate is best')
```

And `!favorite` will have a content of:

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

One interesting aspect of the `!` operator is that it sits right on the cusp between the operator-defining facilities available to the Ruby programmer and what is built into the language. Although you can override `!` and make it do anything you want, you can't override the nearly synonymous `not`. The `not` operator, along with `and`, `or`, `||`, and `&&`, are built in to Ruby, and their behavior is fixed.

The `+` and `–` operators are interesting in a different way: They can be both binary and unary. It's easy to see the dual role of `+` and `-` with numeric expressions. In the expression `-(2+6)`, the minus sign is a unary operator that simply 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, author, "I am sure that #{@content}" )
  end
```

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

Which lets us do this:

```
favorite = Document.new('Favorite', 'Russ', 'Chocolate is best')
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 on the `Document` class, 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.

## Operating Across Classes

One nice thing about the unary operators is that you only need to deal with one object—and one class—at a time. Coping with two objects doesn't present much of a challenge if you are dealing with two objects of the same class, but binary operators that work across classes can be one of those "seems simple until you try it" kind of jobs. Take our `Document` addition method:

```
def +(other)
  Document.new( title, author, "#{content} #{other.content}" )
end
```

Methods don't get much simpler than this: It just does the Ruby thing and assumes that the other operand is a `Document`—or at least an object with a `content` method that returns some text—and lets it go at that.[2] It would be nice, however, if we could add a string to a document, so that if we did this:

```
doc = Document.new( 'hi', 'russ', 'hello')
new_doc = doc + 'out there!'
```

We would end up with a document containing `'hello out there!'` There's not much to making this happen:

```
def +(other)
  if other.kind_of?(String)
    return Document.new( title, author, "#{content} #{other}")
  end
  Document.new( title, author, "#{content} #{other.content}" )
end
```

Great! Now we can add a document and a string together to get ever larger documents. Unfortunately, we missed something. If we reverse the expression and add a document to a string:

2. You could argue that this method is a bit too simple. After all, we just throw away the author and title of the second document. This is fine for an example, but it is perhaps an issue in real life.

```
'I say to you, ' + doc
```

We end up with a very unpleasant error:

```
#<TypeError: can't convert Document into String>
```

The trouble is that this expression calls the + method on the *String* class, which blows up in our face because `String` doesn't know about the `Document` class. The bottom line is that if you want to define binary operators that work across classes, you need to either make sure that both classes understand their responsibilities—as `String` does not in this example—or accept that your expressions will be sensitive to the order in which you write them.[3]

## Staying Out of Trouble

So when should you define operators for your classes and when should you just stick to ordinary methods? Like most software engineering questions, the answer to this one is a resounding "It depends." Mainly it depends on the kind of object you are defining and the specific operations it supports.

The easiest case is where you find yourself building a class that has some natural, intuitive operator definitions. Envy the authors of the Ruby `Matrix` and `Vector` classes: The answer to the question of whether to have a + operator—and what that operator should do—is as close as the nearest linear algebra textbook. Similarly, the built-in `Set` class very logically maps the boolean | and & operators to the union and intersection operations. If you find yourself in a similar situation—you are building a class that has a natural, well-understood meaning for the operators—then count yourself lucky and start coding.

Another easy case is where you are building a class that, although it doesn't come with a whole set of universally understood operators, does have a few operator targets of opportunity. If you are writing some kind of collection class, it's an easy decision to add an << operator. In the same vein, if your class has some natural indexing tendencies, then defining `[]` and perhaps `[]=` may not be a bad idea.

3. Don't get the idea that all is lost with making String cooperate with our Document class. As we will see in Chapter 24, we can teach an old String some new tricks.

Finally, there's the case where, even though there are no widely accepted operators in the domain you are modeling, you realize that many of the methods on your class behave in a way that parallels the ordinary arithmetic operators. Perhaps you are modeling organizational structures and you realize that when you put two employees together you can get a department:

```
department = employee_1 + employee_2
```

And combining two departments will give you a division:

```
division = department_1 + department_2
```

What you can do is invent your very own operator-based object calculus, with a hierarchy of increasingly complex organizational types and a rich set of operators . . .

Actually, what you can do is pull back from the brink. Assigning arbitrary, far-fetched meanings to the common operators is one thing that gives programmer-defined operators a bad name. Remember, the goal is clear and concise code. Code that requires me to recall that I get a department when I add two employees together, but that a department minus an employee is still a (smaller) department, is going to be anything but clear. If you find that your operators are starting to take on a life of their own, then perhaps you have gone a little too far. In any event, it is always a good idea to provide ordinary method names as aliases for your fancy operators as a sort of escape valve, just in case other programmers fail to appreciate the elegance of adding two departments together.

You also need to have proper respect for the generally accepted meanings of the common operators. Operators are nice because they are an easy way of firing off a complicated set of ideas in the head of anyone reading your program. Write this very simple code:

```
a + b
```

And you have, with a single character, conjured up a whole cloud of ideas in your readers' heads, a cloud that goes by the name *addition*. Very convenient, but also a bit dangerous. The danger lies in knowing just how big the cloud is and exactly where its boundaries are. For example, if the users of your class are thinking about plain old elementary school addition, then they will be sure that the `+` operator is commutative, that adding `a` to `b` will produce the same result as adding `b` to `a`. On the other hand,

if they are thinking about higher math—or strings—they might not be so certain. In the same vein, we saw earlier that people do tend to assume that if `a + b` is defined, then `b + a` will also be defined. If you define an operator that doesn't quite live up to its symbol—if `b + a` throws an exception or you define a multiplication operator that isn't commutative or a subtraction that is—then you owe it to the next engineer to at least document the fact.

Sometimes even documentation isn't enough: There are some assumptions that are absolutely core. Show me this expression:

```
c = a + b
```

And then tell me that this code changes the value `b`, and I will be inclined to throw you out of a window. The mental cloud of ideas around every operator holds some absolutely unshakable assumptions—and woe to you if you violate them.

## In the Wild

A good example of an operator that might be commutative—but isn't—is the + operator for instances of the `Time` class. Get yourself an instance of `Time`, perhaps like this:

```
now = Time.now
```

And you can roll the clock forward by simply adding some seconds:

```
one_minute_from_now = now + 60
```

Unfortunately, since `Fixnum` does't know about `Time`, you can't write the expression the other way around:

```
one_minute_from_now = 60 + now  # Bang!
```

Your Ruby installation also has some more exotic operator specimens. Exhibit A is the string formatting operator, `%`. The formatting operator is great when you need to construct a string and you need more control than the usual Ruby `"string #{interpolation}"` gives you. A very simple formatting example would look something like this:

```
"The value of n is %d" % 42
```

This will result in `"The value of n is 42"`—the `%d` in the string signals that this is the place where the value on the right side of the `%` operator should be inserted. If that were all there was, you would probably be better off with plain old string interpolation. The beauty of the format operator is that it gives you very fine control over how the values get inserted into the string. For example, imagine you have the components of a date in three separate variables:

```
day = 4
month = 7
year = 1776
```

Now imagine that you need to format these three date components into a string suitable for use as a filename. You could get there very easily with:

```
file_name = 'file_%02d%02d%d' % [ day, month, year ]
```

Run the code shown here and you will end up with a convenient zero-filled string: `file_04071776`. A nice thing about the formatting operator is that if `%` does not shout "Format!" to you, you can use the equivalent `sprintf` method,[4] which is defined on all Ruby objects.

If you do find the `%` formatting operator a little odd, then you will need to fasten your seat belt for our final example. Think back to Chapter 9 where we discussed specifying program behavior with RSpec. Recall that with RSpec, instead of writing an assertion that some condition is true, you write code that looks like this:

```
x.should == 42
```

Let's pass over the details of what the `should` method does and focus on the == operator. RSpec has turned the meaning of the == operator completely inside out. The garden variety == method is there to answer a question—"Are these two things equal?" But the RSpec version of == is more like an enforcer: If the two values are equal, nothing much happens; but if they are not equal, the test fails. The cool thing about RSpec is that its authors managed to find an alternative meaning for == that not only works in Ruby but also in the squishy computer between your ears. Quite a trick, but not one that is easy to repeat.

4. As all the former C programmers sit up and take notice.

## Wrapping Up

In this chapter we have looked at the ups and downs of defining your own Ruby operators. The up part is that the actual mechanics of defining Ruby operators is very easy—you define a method with the right name. The down part is actually getting that operator to work the way your users might expect it to work. This turns out to be harder for operators than with garden-variety methods because people have some strong built-in expectations of what a given operator should do. The wise coder tries to respect those expectations.

So much for the Ruby operators. Almost. I have deliberately ignored one set of operators in this chapter: those having to do with equality. The reason I have ignored the familiar == and the somewhat less commonplace === operators is that equality is an entire topic in itself, one that we'll tackle in the next chapter.