Addison-Wesley Professional Ruby Series

"Sticking to its tried and tested formula of cutting right to the techniques the modern day Rubyist needs to know, the latest edition of The Ruby Way *keeps its strong reputation going for the latest generation of the Ruby language."*

—**PETER COOPER,** Editor *of Ruby Weekly*

THE RUBY WAY

THIRD EDITION

HAL FULTON

with **ANDRÉ ARKO**

Praise for *The Ruby Way,* Third Edition

"Sticking to its tried and tested formula of cutting right to the techniques the modern day Rubyist needs to know, the latest edition of *The Ruby Way* keeps its strong reputation going for the latest generation of the Ruby language."

Peter Cooper
Editor of *Ruby Weekly*

"The authors' excellent work and meticulous attention to detail continues in this latest update; this book remains an outstanding reference for the beginning Ruby programmer—as well as the seasoned developer who needs a quick refresh on Ruby. Highly recommended for anyone interested in Ruby programming."

Kelvin Meeks
Enterprise Architect

Praise for Previous Editions of *The Ruby Way*

"Among other things, this book excels at explaining metaprogramming, one of the most interesting aspects of Ruby. Many of the early ideas for Rails were inspired by the first edition, especially what is now Chapter 11. It puts you on a rollercoaster ride between 'How could I use this?' and 'This is so cool!' Once you get on that rollercoaster, there's no turning back."

David Heinemeier Hansson
Creator of Ruby on Rails,
Founder at Basecamp

"The appearance of the second edition of this classic book is an exciting event for Rubyists—and for lovers of superb technical writing in general. Hal Fulton brings a lively erudition and an engaging, lucid style to bear on a thorough and meticulously exact exposition of Ruby. You palpably feel the presence of a teacher who knows a tremendous amount and really wants to help you know it too."

David Alan Black
Author of *The Well-Grounded Rubyist*

Once again, beware of string ranges:

```
s1 = "2".."5"
str = "28"
s1.include?(str)    # true (misleading!)
```

6.2.5 Converting to Arrays

When we convert a range to an array, the interpreter simply applies `succ` repeatedly until the end is reached, appending each item onto an array that is returned:

```
r = 3..12
arr = r.to_a     # [3,4,5,6,7,8,9,10,11,12]
```

This naturally won't work with `Float` ranges. It may sometimes work with `String` ranges, but this should be avoided because the results will not always be obvious or meaningful.

6.2.6 Backward Ranges

Does a *backward* range make any sense? Yes and no. For example, this is a perfectly valid range:

```
r = 6..3
x = r.begin              # 6
y = r.end                # 3
flag = r.end_excluded?   # false
```

As you see, we can determine its starting and ending points and whether the end is included in the range. However, that is nearly *all* we can do with such a range.

```
arr = r.to_a       # []
r.each {|x| p x}   # No iterations
y = 5
r.include?(y)      # false (for any value of y)
```

Does that mean that backward ranges are necessarily "evil" or useless? Not at all. It is still useful, in some cases, to have the endpoints encapsulated in a single object.

In fact, arrays and strings frequently take "backward ranges" because these are zero-indexed from the left but "minus one"-indexed from the right. Therefore, we can use expressions like these:

```
string = "flowery"
str1   = string[0..-2]   # "flower"
str2   = string[1..-2]   # "lower"
str3   = string[-5..-3]  # "owe" (actually a forward range)
```

6.2.7 The Flip-Flop Operator

When the range operator is used in a condition, it is treated specially. This usage of `..` is called the *flip-flop operator* because it is essentially a toggle that keeps its own state rather than a true range.

This trick, apparently originating with Perl, is useful. But understanding how it works takes a little effort.

Imagine we had a Ruby source file with embedded docs between `=begin` and `=end` tags. How would we extract and output only those sections? (Our state toggles between "inside" a section and "outside" a section, hence the flip-flop concept.) The following piece of code, while perhaps unintuitive, will work:

```
file.each_line do |line|
  puts line if (line=~/=begin/)..(line=~/=end/)
end
```

How can this work? The magic all happens in the flip-flop operator.

First, realize that this "range" is preserving a state internally, but this fact is hidden. When the left endpoint becomes true, the range itself returns true; it then remains true until the right endpoint becomes true, and the range toggles to false.

This kind of feature might be used in some cases, such as parsing section-oriented config files, selecting ranges of items from lists, and so on.

However, I personally don't like the syntax, and others are also dissatisfied with it. Removing it has been discussed publicly at bugs.ruby-lang.org/issues/5400, and Matz himself has said that it will eventually be removed.

So what's wrong with the flip-flop? Here is my opinion.

First, in the preceding example, take a line with the value `=begin`. As a reminder, the `=~` operator does not return true or false as we might expect; it returns the position of the match (a `Fixnum`) or `nil` if there was no match. So then the expressions in the range evaluate to `0` and `nil`, respectively.

However, if we try to construct a range from `0` to `nil`, it gives us an error because it is nonsensical:

```
range = 0..nil    # error!
```

Furthermore, bear in mind that in Ruby, only `false` and `nil` evaluate to false; everything else evaluates as `true`. Then a range ordinarily would not evaluate as `false`.

```
puts "hello" if x..y
# Prints "hello" for any valid range x..y
```

And again, suppose we stored these values in variables and then used the variables to construct the range. This doesn't work because the test is always true:

```
file.each_line do |line|
  start = line=~/=begin/
  stop = line=~/=end/
  puts line if start..stop
end
```

What if we put the range itself in a variable? This doesn't work either because, once again, the test is always true:

```
file.each_line do |line|
  range = (line=~/=begin/)..(line=~/=end/)
  puts line if range
end
```

To understand this, we have to understand that the entire range (with both endpoints) is reevaluated each time the loop is run, but the internal state is also factored in. The flip-flop operator is therefore not a true range at all. The fact that it looks like a range but is not is considered a bad thing by some.

Finally, think of the endpoints of the flip-flop. They are reevaluated every time, but this reevaluation cannot be captured in a variable that can be substituted. In effect, the flip-flop's endpoints are like `procs`. They are not values; they are code. The fact that something that looks like an ordinary expression is really a `proc` is also undesirable.

Having said all that, the functionality is still useful. Can we write a class that encapsulates this function without being so cryptic and magical? As it turns out, this is not difficult. Listing 6.1 introduces a simple class called `Transition` that mimics the behavior of the flip-flop.

Listing 6.1 The `Transition` Class

```
class Transition
  A, B = :A, :B
  T, F = true, false

          # state,p1,p2  => newstate, result
  Table = {[A,F,F]=>[A,F], [B,F,F]=>[B,T],
           [A,T,F]=>[B,T], [B,T,F]=>[B,T],
           [A,F,T]=>[A,F], [B,F,T]=>[A,T],
           [A,T,T]=>[A,T], [B,T,T]=>[A,T]}

  def initialize(proc1, proc2)
    @state = A
    @proc1, @proc2 = proc1, proc2
  end

  def check?
    p1 = @proc1.call ? T : F
    p2 = @proc2.call ? T : F
    @state, result = *Table[[@state,p1,p2]]
    return result
  end
end
```

In the `Transition` class, we use a simple state machine to manage transitions. We initialize it with a pair of `procs` (the same ones used in the flip-flop). We do lose a little convenience in that any variables (such as `line`) used in the `procs` must already be in scope. But we now have a solution with no "magic" in it, where all expressions behave as they do any other place in Ruby.

Here's a slight variant on the same solution. Let's change the `initialize` method to take two arbitrary expressions:

```
  def initialize(flag1, flag2)
    @state = A
    @flag1, @flag2 = flag1, flag2
  end

  def check?(item)
```

```
      p1 = (@flag1 === item) ? T : F
      p2 = (@flag2 === item) ? T : F
      @state, result = *Table[[@state, p1, p2]]
      return result
    end
```

The case equality operator is used to check the relationship of the starting and ending flags with the variable.

Here is how we use the new version:

```
trans = Transition.new(/=begin/, /=end/)
file.each_line do |line|
  puts line if trans.check?(line)
end
```

I do recommend an approach like this, which is more explicit and less magical.

6.2.8 Custom Ranges

Let's look at an example of a range made up of some arbitrary object. Listing 6.2 shows a simple class to handle Roman numerals.

Listing 6.2 A Roman Numeral Class

```
class Roman
  include Comparable

  I,IV,V,IX,X,XL,L,XC,C,CD,D,CM,M =
    1, 4, 5, 9, 10, 40, 50, 90, 100, 400, 500, 900, 1000

  Values = %w[M CM D CD C XC L XL X IX V IV I]

  def Roman.encode(value)
    return "" if self == 0
    str = ""
    Values.each do |letters|
      rnum = const_get(letters)
      if value >= rnum
        return(letters + str=encode(value-rnum))
      end
    end
    str
  end

  def Roman.decode(rvalue)
    sum = 0
```

```
    letters = rvalue.split('')
    letters.each_with_index do |letter,i|
      this = const_get(letter)
      that = const_get(letters[i+1]) rescue 0
      op = that > this ? :- : :+
      sum = sum.send(op,this)
    end
    sum
  end

  def initialize(value)
    case value
      when String
        @roman = value
        @decimal = Roman.decode(@roman)
      when Symbol
        @roman = value.to_s
        @decimal = Roman.decode(@roman)
      when Numeric
        @decimal = value
        @roman = Roman.encode(@decimal)
    end
  end

  def to_i
    @decimal
  end

  def to_s
    @roman
  end

  def succ
    Roman.new(@decimal+1)
  end

  def <=>(other)
    self.to_i <=> other.to_i
  end
end

def Roman(val)
  Roman.new(val)
end
```

I'll cover a few highlights of this class first. It can be constructed using a string or a symbol (representing a Roman numeral) or a `Fixnum` (representing an ordinary Hindu-Arabic decimal number). Internally, conversion is performed, and both forms are stored. There is a "convenience method" called `Roman`, which simply is a shortcut