CHAPTER 3

Take Advantage of Ruby's Smart Collections

Here's something to do the next time you have a rainy afternoon to kill: Download two or three significant Ruby programs, maybe utilities that you use every day or applications you have heard of or perhaps projects whose names you just happen to like. Once you have the code, settle in and start reading it; figure out how it works and why it's put together the way it is. One of the things you are likely to find is arrays, lots and lots of arrays. You will probably also come across a good number of hashes mixed in with those arrays.

In the pages that follow we are going to look at the Ruby versions of these two venerable data structures and discover that they have some surprising talents. Along the way we will also see how you can create arrays in a surprising number of ways, how you can run through the elements of an array without a hint of an index, and how orderly the Ruby hash class is. We will also take a tour of some of the rougher parts of the Ruby collection landscape and, finally, we will visit a bit with one of the lesser known of the Ruby collection classes.

Literal Shortcuts

Before you can do anything with a collection you need to have one. A common way to conjure up a collection is with a literal, but the trouble with collection literals is that they have parts—the things in the collection—and spelling out all of those individual

elements can make for very inelegant code. Thus a simple literal array of words can turn into a thicket of quotes and commas:

```
poem_words = [ 'twinkle', 'little', 'star', 'how', 'I', 'wonder' ]
```

Fortunately, Ruby gives us some convenient syntactical shortcuts for exactly this situation. If you need to initialize an array of strings, where none of the strings have embedded spaces, you can simply say:

```
poem_words = %w{ twinkle little star how I wonder }
```

Hash literals also present you with a couple of choices. The traditional hash literal has you associating your keys with your values with the so-called **hash rocket**, =>, like this:

```
freq = { "I" => 1, "don't" => 1, "like" => 1, "spam" => 963 }
```

This last form is perfectly general purpose in that it doesn't care about the types of the hash keys. If you happen to be using symbols as your keys—a very common practice—then you can cut this:

```
book_info = { :first_name => 'Russ', :last_name => 'Olsen' }
```

Down to this:

```
book_info = { first_name: 'Russ', last_name: 'Olsen' }
```

Either way you get exactly the same hash.[1]

Instant Arrays and Hashes from Method Calls

Another way to get hold of a collection is to have Ruby manufacture one for you during the method-calling process. Although Ruby methods typically take a fixed number of arguments, it is possible to build methods that take a variable number of

1. Well, you get the same hash if you are using Ruby 1.9, which is where this shorter hash literal syntax made its appearance.

arguments. If you have a basic set of arguments and simply want to allow your callers to be able to omit one here or there, you can simply specify defaults. This method, for instance, will take one or two arguments:

```
def load_font( name, size = 12 )
  # Go font hunting...
end
```

Sometimes, however, what you need is a method that will take a completely arbitrary set of arguments. Ruby has a special syntax for this: If, in your method definition, you stick an asterisk before one of your parameter names, that parameter will soak up any extra arguments passed to the method. The value of the starred parameter will be an array containing all the extra arguments. Thus this method:

```
def echo_all( *args )
  args.each { |arg| puts arg }
end
```

Will take any number of arguments and print them out. You can only have one starred parameter, but it can be pretty much anywhere in your parameter list:[2]

```
def echo_at_least_two( first_arg, *middle_args, last_arg )
  puts "The first argument is #{first_arg}"
  middle_args.each { |arg|puts "A middle argument is #{arg}" }
  puts "The last argument is #{last_arg}"
end
```

In practice this means that you can sometimes rely on Ruby to manufacture an array for you. Take the `add_authors` method for example:

```
class Document

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

2. In Ruby 1.9 it can be anywhere. In 1.8 the starred argument needed to be at the end of the argument list.

```
  def add_authors( names )
    @author += " #{names.join(' ')}"
  end
end
```

The `add_authors` method lets the caller specify a number of different names as the author of a document. As it is written above, you pass the `add_authors` method an array of names:

```
doc.add_authors( [ 'Strunk', 'White' ] )
```

Once it has the array of author's names, `add_authors` uses the array method `join` to combine the elements into a single space-delimited string, which it then appends onto `@author`.

This initial stab at `add_authors` does work, but it is not as smooth as it might be: By adding a star to the `names` argument you can turn your method into one that takes any number of arguments—all delivered in an array.[3] Adding the star, like so:

```
class Document

  # Most of the class omitted...

  def add_authors( *names )
    @author += " #{names.join(' ')}"
  end
end
```

Means that you can relieve your users of the need to wrap the names in an array:

```
doc.add_authors( 'Strunk', 'White' )
```

The jargon for a star used in this context is **splat**, as in "splat names." Think of an exploding array, with elements flying every which way.

3. The star actually works the other way too: If you happen to find yourself holding a three-element array and want to pass that trio of objects to a method that takes three arguments, you can simply say `some_method( *my_array )`. Ah, symmetry.

Hashes also have a bonus method-passing feature. Any method can, of course, take a literal hash as an argument. Thus, we might have a method that will locate a font by name and size, and pass the values in as a hash:

```
def load_font( specification_hash )
  # Load a font according to  specification_hash[:name] etc.
end
```

So that we can load a font like this:

```
load_font( { :name => 'times roman', :size => 12 })
```

The bonus comes if the hash is at the end[4] of the argument list—as it is in the `load_font` example. In that case we can leave the braces off when we call the method. Given this, we can shorten our call to `load_font` down to:

```
load_font( :name => 'times roman', :size => 12 )
```

Written without the braces, this method call has a nice, named parameter feel to it. You can even go one step further and leave the parentheses off, resulting in something that looks more like a command and less like a method call:

```
load_font :name => 'times roman', :size => 12
```

Running Through Your Collection

Once you have a collection, you will probably want to do something with it. Frequently that something will involve iterating through the collection one element at a time. Depending on your programming background, you might be tempted to do the iterating with an index-based loop, perhaps like this:

```
words = %w{ Mary had a little lamb }

for i in 0..words.size
  puts words[i]
end
```

4. That is, the end of the argument list not counting any explicit block argument. More about this in Chapter 19.

I said it in the last chapter, but it bears repeating here: Don't do that. The way to run through a Ruby collection is with the `each` method:

```
words.each { |word| puts word }
```

Hashes also sport an `each` method, though the `Hash` version comes with a twist: If the block you supply to the `Hash` rendition of `each` takes a single argument, like this:

```
movie = { title: '2001', genre: 'sci fi', rating: 10 }

movie.each { |entry| pp entry }
```

Then `each` will call the block with a series of two element arrays, where the first element is a key and the second the corresponding value, so that the code above will print:

```
[:title, "2001"]
[:genre, "sci fi"]
[:rating, 10]
```

Alternatively, you can supply a block that takes two arguments, like this:

```
movie.each { |name, value| puts "#{name} => #{value}"}
```

Both hashes and arrays have many talents beyond the `each` method: Both of these classes come equipped with very extensive APIs. To take a simple example, imagine that we want to add a method to the `Document` class, a method that will return the index of a particular word. Without thinking we might just grab for `each`:

```
def index_for( word )
  i = 0
  words.each do |this_word|
    return i if word == this_word
    i += 1
  end
  nil
end
```

A much better way is to look at the `Array` API and come up with the `find_index` method:

```
def index_for( word )
  words.find_index { |this_word| word == this_word }
end
```

Two other collection methods you want to be sure to include in your mental toolkit are `map` and `inject`. Like `each` and `find_index`, `map` takes a block and runs through the collection calling the block for each element. The difference is that instead of making some kind of decision based on the return from the block the way `find_index` does, `map` cooks up a new array containing everything the block returned, in order. The `map` method is incredibly useful for transforming the contents of a collection en mass. For example, if you wanted an array of all the word lengths, you might turn to `map`:

```
pp doc.words.map { |word| word.size }
```

Which would print:

```
[3, 5, 2, 3, 4]
```

Alternatively, you might want an all-lowercase version of the words in your document. That's easy, too:

```
lower_case_words = doc.words.map { |word| word.downcase }
```

The `inject` method is a bit harder to get your arms around but well worth the trouble. To see what `inject` can do for you, imagine that you need to write a method that will compute the average length of the words in a document. To compute the average, we need two things: the total number of words—which we can easily get by calling `word_count`—and the total length of all the words (not counting white space), which is going to take some work. To do that work, we might use `each`, something like this:

```
class Document

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

```
    def average_word_length
      total = 0.0
      words.each { |word| total += word.size }

      total / word_count
    end
  end
```

Can we improve on this? You bet: This is the kind of problem that the `inject` method is tailor-made to solve. Like `each`, `inject` takes a block and calls the block with each element of the collection. Unlike `each`, `inject` passes in two arguments to the block: Along with each element, you get the current result—the current sum if you are adding up all of the word lengths. Each time `inject` calls the block, it replaces the current result with the return value of the previous call to the block. When `inject` runs out of elements, it returns the result as its return value.

Here, for example, is our `average_word_length` method, reworked to use `inject`:

```
    def average_word_length
      total = words.inject(0.0){ |result, word| word.size + result}
      total / word_count
    end
```

The argument passed into `inject`—`0.0` in the example—is the initial value of the result. If you don't supply an initial value, `inject` will skip calling the block for the first element of the array and simply use that element as the initial result.

Altogether, array instances carry around an even one hundred public methods while hashes include more than eighty. A key part of knowing how to program in Ruby is getting to know this extensive toolkit.

Beware the Bang!

One thing you really want to know is which methods will actually change your collection and which will leave it be. This is not as obvious as you might expect. For example, arrays come with a `reverse` method, which, unsurprisingly, switches the order of the elements. You would think that reversing any nontrivial array is surely going to change it:

```
a = [ 1, 2, 3]
a.reverse
```

Unfortunately, you would think wrong: The `reverse` method does not change anything. Print out the array from the previous example and you would see its order undisturbed. The `reverse` method actually returns a reversed *copy* of the array, so that this:

```
pp a.reverse
pp a
```

Will print:

```
[3, 2, 1]
[1, 2, 3]
```

If you really want to reverse the array in place, you need `reverse!`, with an exclamation point at the end:

```
a.reverse!
pp a
```

Run this code and you will see:

```
[3, 2, 1]
```

In the same way, the array `sort` method returns a sorted copy, while `sort!` sorts the array in place.

Don't, however, get the idea that only methods with names ending in `!` will change your collection. Remember, the Ruby convention is that an exclamation point at the end of a method name indicates that the method is the dangerous or surprising version of a pair of methods. Since making a modified copy of a collection seems very safe while changing a collection in place can be a bit dicey, we have `sort` and `sort!`. The punch line is that there are many `!`-less methods in Ruby's collection classes, methods that will cheerfully change your collection in place as an intrinsic part of what they do. Thus, `push`, `pop`, `delete`, and `shift` are as capable of changing your

array as `sort!`. Be sure you know what a method is going to do to your collection before you call it.

Rely on the Order of Your Hashes

Ruby arrays and hashes have one thing in common that takes many programmers by surprise: They are both ordered. Order and arrays go together like coders and coffee—it's hard to imagine one without the other. But hashes have, in many programming languages and in earlier versions of Ruby, usually been any unruly bunch. The hashes available in many other languages and in the bad old days of Ruby would mix up the entries in a more or less random order. With the advent of Ruby 1.9, however, hashes have become firmly ordered. Create a new hash, perhaps with a literal:

```
hey_its_ordered = { first: 'mama', second: 'papa', third: 'baby' }
```

And iterate through it:

```
hey_its_ordered.each { |entry| pp entry }
```

And the items will stay firmly ordered:

```
[:first, "mama"]
[:second, "papa"]
[:third, "baby"]
```

If you add items to an existing hash, they get sent to the end of the line. Thus if we add a fourth entry to our hash:

```
hey_its_ordered[:fourth] = 'grandma'
```

Then granny will take up residence at the far end of the hash. Changing the value of an existing entry, however, does not disturb its place in line, so if we set `hey_its_ordered[:first]` to `'mom'`, it would stay right there at the front.

In the Wild

One reason that the basic collections are so pervasive in the Ruby code base is that Ruby programmers tend to reach for them at times when programmers from other

language traditions might go for a more specialized class. Thus, if you call `File.readlines('/etc/passwd')` you *don't* get back an instance of some specific line holding class: What you do get is a simple array full of strings. In the same spirit, ask an object for its public methods by calling `object.public_methods` or a class for its forebearers by calling `my_class.ancestors` and in each case you will get back a plain old array of method names or super classes.[5] And it's not just arrays: Ruby programmers frequently use hashes to specify all sorts of things. Thus Rails applications tend to have lots of code that looks like this:

```
link_to "Font", :controller =>"fonts", :action =>"show", :id =>@font
```

Perhaps the ultimate expression of this "use the simple collections" philosophy can be found in the `XmlSimple` gem. This wonderful bit of software allows you to turn any XML file, like this:

```
<characters>
  <super-hero>
    <name>Spiderman</name>
    <origin>Radioactive Spider</origin>
  </super-hero>
  <super-hero>
    <name>Hulk</name>
    <origin>Gamma Rays</origin>
  </super-hero>
  <super-hero>
    <name>Reed Richards</name>
    <origin>Cosmic Rays</origin>
  </super-hero>
</characters>
```

Into a convenient set of nested hashes and array:

```
{"super-hero"=>
  [{"name"=>["Spiderman"], "origin"=>["Radioactive Spider"]},
   {"name"=>["Hulk"], "origin"=>["Gamma Rays"]},
   {"name"=>["Reed Richards"], "origin"=>["Cosmic Rays"]}]}
```

5. Well, superclasses and mixed-in modules.

With just a couple of lines of code:

```
require 'xmlsimple'
data = XmlSimple.xml_in('dc.xml')
```

There are two reasons for this preference for the bare collections over more specialized classes. First, the Ruby collection classes are so powerful that often there is no practical reason to create a custom-tailored collection simply to get some specialized feature. Frequently, a call to `each` or `map` is all you really need. And second, all things being equal, Ruby programmers actually prefer to work with generic collections. To the Ruby way of thinking, one less class is one less thing to go wrong. In addition, I know exactly how an array is going to react to a call to `each` or `map`, which isn't necessarily true of the `SpecializedCollectionOfStuff`. When the problem is complexity, the cure might just be simplicity.

Staying Out of Trouble

If you look back over the pages of this chapter you will see that it is just full of calls to methods like `each` and `map`, methods that run through some collection, calling a block for each element. There is a lot of coding goodness in these methods, but beware: They will turn on you if you abuse them. The easiest way to screw up one of these iterating methods is to change the collection out from underneath the method. Here, for example, is a seriously misguided attempt to remove all of the negative numbers from an array:

```
array = [ 0, -10, -9, 5, 9 ]
array.each_index {|i| array.delete_at(i) if array[i] < 0}
pp array
```

The trouble with this code is that it will tend to leave some negative numbers behind: Removing the first one (the `-10`) from the array messes up the internal indexing of the `each` method, so much that it will miss the second negative number, leaving us with a result of:

```
[0, -9, 5, 9]
```

Although the problem in this example is reasonably easy to spot,[6] this is sadly not always the case: In particular, if your collection is being shared by several different threads, you run the real risk of one thread modifying a collection that the other thread is iterating over.

You will also want to take care when adding new elements well past the existing end of your array. Although there is not much danger of tacking on elements one by one with `push` or `<<`, you can also plug new values anywhere you want:

```
array = []
array[24601] = "Jean Valjean"
```

Since arrays are continuous, the second line above instantly created 24,602 new elements of array, most of them set to `nil`.

Another problem with arrays and hashes is that because they are so tremendously useful we sometimes reach for them when we should really be using something else. Imagine, for instance, that we are interested in knowing whether a specific word appears in a document. We could do this with a hash, using the words for keys and anything—perhaps `true`—as the values:

```
word_is_there = {}
words.each { |word| word_is_there[ word ] = true }
```

Alternatively, we might create an array that will keep track of each unique word:

```
unique = []
words.each { |word| unique << word unless unique.include?(word) }
```

The trouble with both of these approaches is that they are roundabout: The whole point of a hash is to map keys to values, but in our hash-based implementation we are really just interested in the keys—the words. The array-based version is not much better. Since arrays don't really mind duplicate values, every time we put a new word in

6. . . . and fix: A much cleaner way to get rid of those pesky negative numbers—a way that actually works—is to simply say `array.delete_if {|x| x < 0}`.

the array we need to search the whole thing (via the `include?` method) to see if the word is already there.

What we really need is a collection that doesn't allow duplicates but does feature very fast and easy answers to the "is this object in there?" question. The punch line is that we need a *set*. Fortunately, Ruby comes with a perfectly serviceable, if sometimes forgotten, `Set` class:

```
require 'set'

word_set = Set.new( words )
```

In coding, as in carpentry, you need the right tool for the job.

Wrapping Up

Making effective use of the collection classes is every bit as much a part of writing clear, idiomatic Ruby as getting the indentation and variable names right. The reason for this is very simple: The Ruby collection classes are like giant levers—if you know how to use them you can move a lot of computing weight with very little effort. Part of being a good Ruby programmer is knowing the `Array` and `Hash` classes, knowing what they can do, and knowing when to use them and when to reach for something else.

Of course, once you have your collections you are going to need to put something in them. Frequently that something will be a string.