Extracted from:

Programming Ruby

The Pragmatic Programmers' Guide

Second Edition

Chapter 4

Containers, Blocks, and Iterators

A jukebox with one song is unlikely to be popular (except perhaps in some very, very scary bars), so pretty soon we'll have to start thinking about producing a catalog of available songs and a playlist of songs waiting to be played. Both of these are *containers:* objects that hold references to one or more other objects.

Both the catalog and the playlist need a similar set of methods: add a song, remove a song, return a list of songs, and so on. The playlist may perform additional tasks, such as inserting advertising every so often or keeping track of cumulative play time, but we'll worry about these things later. In the meantime, it seems like a good idea to develop some kind of generic `SongList` class, which we can specialize into catalogs and playlists.

Containers

Before we start implementing, we'll need to work out how to store the list of songs inside a `SongList` object. We have three obvious choices. We could use the Ruby `Array` type, use the Ruby `Hash` type, or create our own list structure. Being lazy, for now we'll look at arrays and hashes and choose one of these for our class.

Arrays

The class `Array` holds a collection of object references. Each object reference occupies a position in the array, identified by a non-negative integer index.

You can create arrays by using literals or by explicitly creating an `Array` object. A literal array is simply a list of objects between square brackets.

```
a = [ 3.14159, "pie", 99 ]
a.class    →   Array
a.length   →   3
a[0]       →   3.14159
a[1]       →   "pie"
a[2]       →   99
a[3]       →   nil

b = Array.new
b.class    →   Array
b.length   →   0
b[0] = "second"
b[1] = "array"
b          →   ["second", "array"]
```

Arrays are indexed using the `[ ]` operator. As with most Ruby operators, this is actually a method (an instance method of class `Array`) and hence can be overridden in subclasses. As the example shows, array indices start at zero. Index an array with a non-negative integer, and it returns the object at that position or returns `nil` if nothing is there. Index an array with a negative integer, and it counts from the end.

```
a = [ 1, 3, 5, 7, 9 ]
a[-1]    →   9
a[-2]    →   7
a[-99]   →   nil
```

This indexing scheme is illustrated in more detail in Figure 4.1 on the following page.

You can also index arrays with a pair of numbers, `[ start, count ]`. This returns a new array consisting of references to `count` objects starting at position `start`.

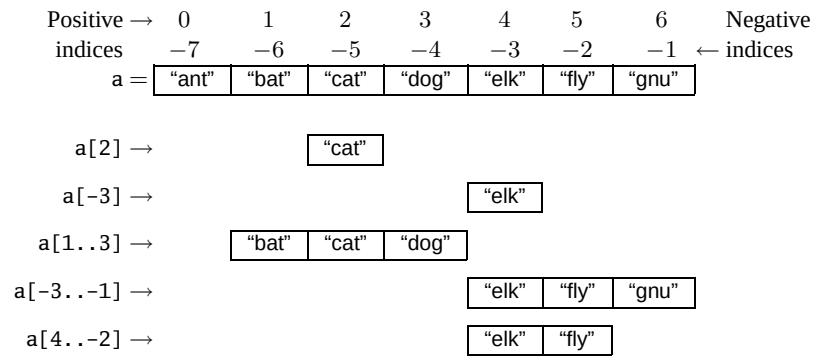
```
a = [ 1, 3, 5, 7, 9 ]
a[1, 3]    →   [3, 5, 7]
a[3, 1]    →   [7]
a[-3, 2]   →   [5, 7]
```

Finally, you can index arrays using ranges, in which start and end positions are separated by two or three periods. The two-period form includes the end position, and the three-period form does not.

```
a = [ 1, 3, 5, 7, 9 ]
a[1..3]     →   [3, 5, 7]
a[1...3]    →   [3, 5]
a[3..3]     →   [7]
a[-3..-1]   →   [5, 7, 9]
```

The `[ ]` operator has a corresponding `[ ]=` operator, which lets you set elements in the array. If used with a single integer index, the element at that position is replaced by whatever is on the right side of the assignment. Any gaps that result will be filled with `nil`.

Figure 4.1. How arrays are indexed




```
a = [ 1, 3, 5, 7, 9 ]   →   [1, 3, 5, 7, 9]
a[1] = 'bat'            →   [1, "bat", 5, 7, 9]
a[-3] = 'cat'           →   [1, "bat", "cat", 7, 9]
a[3] = [ 9, 8 ]         →   [1, "bat", "cat", [9, 8], 9]
a[6] = 99               →   [1, "bat", "cat", [9, 8], 9, nil, 99]
```

If the index to `[ ]=` is two numbers (a start and a length) or a range, then those elements in the original array are replaced by whatever is on the right side of the assignment. If the length is zero, the right side is inserted into the array before the start position; no elements are removed. If the right side is itself an array, its elements are used in the replacement. The array size is automatically adjusted if the index selects a different number of elements than are available on the right side of the assignment.

```
a = [ 1, 3, 5, 7, 9 ]   →   [1, 3, 5, 7, 9]
a[2, 2] = 'cat'         →   [1, 3, "cat", 9]
a[2, 0] = 'dog'         →   [1, 3, "dog", "cat", 9]
a[1, 1] = [ 9, 8, 7 ]   →   [1, 9, 8, 7, "dog", "cat", 9]
a[0..3] = []            →   ["dog", "cat", 9]
a[5..6] = 99, 98        →   ["dog", "cat", 9, nil, nil, 99, 98]
```

Arrays have a large number of other useful methods. Using these, you can treat arrays as stacks, sets, queues, dequeues, and fifos. A complete list of array methods starts on page 406.

Hashes

Hashes (sometimes known as *associative arrays*, *maps*, or *dictionaries*) are similar to arrays in that they are indexed collections of object references. However, while you index arrays with integers, you can index a hash with objects of any type: strings, regular expressions, and so on. When you store a value in a hash, you actually supply

two objects—the index, normally called the *key*, and the value. You can subsequently retrieve the value by indexing the hash with the same key. The values in a hash can be objects of any type.

The example that follows uses hash literals: a list of *key* => *value* pairs between braces.

```
h = { 'dog' => 'canine', 'cat' => 'feline', 'donkey' => 'asinine' }

h.length   →   3
h['dog']   →   "canine"
h['cow'] = 'bovine'
h[12]    = 'dodecine'
h['cat'] = 99
h          →   {"cow"=>"bovine", "cat"=>99, 12=>"dodecine",
               "donkey"=>"asinine", "dog"=>"canine"}
```

Compared with arrays, hashes have one significant advantage: they can use any object as an index. However, they also have a significant disadvantage: their elements are not ordered, so you cannot easily use a hash as a stack or a queue.

You'll find that hashes are one of the most commonly used data structures in Ruby. A full list of the methods implemented by class `Hash` starts on page 471.

Implementing a SongList Container

After that little diversion into arrays and hashes, we're now ready to implement the jukebox's `SongList`. Let's invent a basic list of methods we need in our `SongList`. We'll want to add to it as we go along, but this will do for now.

`append(song)` → `list`
: Append the given song to the list.

`delete_first()` → `song`
: Remove the first song from the list, returning that song.

`delete_last()` → `song`
: Remove the last song from the list, returning that song.

`[index]` → `song`
: Return the song at the integer *index*.

`with_title(title)` → `song`
: Return the song with the given title.

This list gives us a clue to the implementation. The ability to append songs at the end, and remove them from both the front and end, suggests a *dequeue*—a double-ended queue—which we know we can implement using an `Array`. Similarly, the ability to return a song at an integer position in the list is supported by arrays.

However, you also need to be able to retrieve songs by title, which may suggest using a hash, with the title as a key and the song as a value. Could we use a hash? Well, possibly, but this causes problems. First, a hash is unordered, so we'd probably need to use an ancillary array to keep track of the list. A second, bigger problem is that a hash does not support multiple keys with the same value. That would be a problem for our playlist, where the same song may be queued for playing multiple times. So, for now we'll stick with an array of songs, searching it for titles when needed. If this becomes a performance bottleneck, we can always add some kind of hash-based lookup later.

We'll start our class with a basic `initialize` method, which creates the `Array` we'll use to hold the songs and stores a reference to it in the instance variable `@songs`.

```
class SongList
  def initialize
    @songs = Array.new
  end
end
```

The `SongList#append` method adds the given song to the end of the `@songs` array. It also returns *self*, a reference to the current `SongList` object. This is a useful convention, as it lets us chain together multiple calls to `append`. We'll see an example of this later.

```
class SongList
  def append(song)
    @songs.push(song)
    self
  end
end
```

Then we'll add the `delete_first` and `delete_last` methods, trivially implemented using `Array#shift` and `Array#pop`, respectively.

```
class SongList
  def delete_first
    @songs.shift
  end
  def delete_last
    @songs.pop
  end
end
```

So far, so good. Our next method is `[ ]`, which accesses elements by index. These kind of simple delegating methods occur frequently in Ruby code: don't worry if your code ends up containing a bunch of one- or two-line methods—it's a sign that you're designing things correctly.

```
class SongList
  def [](index)
    @songs[index]
  end
end
```

At this point, a quick test may be in order. To do this, we're going to use a testing framework called TestUnit that comes with the standard Ruby distributions. We won't describe it fully yet (we do that in the *Unit Testing* chapter starting on page 143). For now, we'll just say that the method `assert_equal` checks that its two parameters are equal, complaining bitterly if they aren't. Similarly, the method `assert_nil` complains unless its parameter is `nil`. We're using these assertions to verify that the correct songs are deleted from the list.

The test contains some initial housekeeping, necessary to tell Ruby to use the TestUnit framework and to tell the framework that we're writing some test code. Then we create a `SongList` and four songs and append the songs to the list. (Just to show off, we use the fact that `append` returns the `SongList` object to chain together these method calls.) We can then test our `[ ]` method, verifying that it returns the correct song (or `nil`) for a set of indices. Finally, we delete songs from the start and end of the list, checking that the correct songs are returned.

```
require 'test/unit'
class TestSongList < Test::Unit::TestCase
  def test_delete
    list = SongList.new
    s1 = Song.new('title1', 'artist1', 1)
    s2 = Song.new('title2', 'artist2', 2)
    s3 = Song.new('title3', 'artist3', 3)
    s4 = Song.new('title4', 'artist4', 4)
    list.append(s1).append(s2).append(s3).append(s4)
    assert_equal(s1, list[0])
    assert_equal(s3, list[2])
    assert_nil(list[9])
    assert_equal(s1, list.delete_first)
    assert_equal(s2, list.delete_first)
    assert_equal(s4, list.delete_last)
    assert_equal(s3, list.delete_last)
    assert_nil(list.delete_last)
  end
end
```

produces:

```
Loaded suite -
Started
.
Finished in 0.002314 seconds.

1 tests, 8 assertions, 0 failures, 0 errors
```

The running test confirms that eight assertions were executed in one test method, and they all passed. We're on our way to a working jukebox!

Now we need to add the facility that lets us look up a song by title. This is going to involve scanning through the songs in the list, checking the title of each. To do this, we first need to spend a couple of pages looking at one of Ruby's neatest features: iterators.

Blocks and Iterators

Our next problem with `SongList` is to implement the method `with_title` that takes a string and searches for a song with that title. This seems straightforward: we have an array of songs, so we'll go through it one element at a time, looking for a match.

```
class SongList
  def with_title(title)
    for i in 0...@songs.length
      return @songs[i] if title == @songs[i].name
    end
    return nil
  end
end
```

This works, and it looks comfortingly familiar: a `for` loop iterating over an array. What could be more natural?

It turns out there *is* something more natural. In a way, our `for` loop is somewhat too intimate with the array; it asks for a length, and it then retrieves values in turn until it finds a match. Why not just ask the array to apply a test to each of its members? That's just what the `find` method in `Array` does.

```
class SongList
  def with_title(title)
    @songs.find {|song| title == song.name }
  end
end
```

The method `find` is an *iterator*—a method that invokes a block of code repeatedly. Iterators and code blocks are among the more interesting features of Ruby, so let's spend a while looking into them (and in the process we'll find out exactly what that line of code in our `with_title` method actually does).

Implementing Iterators

A Ruby iterator is simply a method that can invoke a block of code. At first sight, a block in Ruby looks just like a block in C, Java, C#, or Perl. Unfortunately, in this case looks are deceiving—a Ruby block *is* a way of grouping statements, but not in the conventional way.

First, a block may appear only in the source adjacent to a method call; the block is written starting on the same line as the method call's last parameter (or the closing

parenthesis of the parameter list). Second, the code in the block is not executed at the time it is encountered. Instead, Ruby remembers the context in which the block appears (the local variables, the current object, and so on) and then enters the method. This is where the magic starts.

Within the method, the block may be invoked, almost as if it were a method itself, using the `yield` statement. Whenever a `yield` is executed, it invokes the code in the block. When the block exits, control picks back up immediately after the `yield`.[1] Let's start with a trivial example.

```
def three_times
  yield
  yield
  yield
end
three_times { puts "Hello" }
```

produces:

```
Hello
Hello
Hello
```

The block (the code between the braces) is associated with the call to the method `three_times`. Within this method, `yield` is called three times in a row. Each time, it invokes the code in the block, and a cheery greeting is printed. What makes blocks interesting, however, is that you can pass parameters to them and receive values from them. For example, we could write a simple function that returns members of the Fibonacci series up to a certain value.[2]

```
def fib_up_to(max)
  i1, i2 = 1, 1        # parallel assignment (i1 = 1 and i2 = 1)
  while i1 <= max
    yield i1
    i1, i2 = i2, i1+i2
  end
end
fib_up_to(1000) {|f| print f, " " }
```

produces:

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

---

1. Programming-language buffs will be pleased to know that the keyword `yield` was chosen to echo the `yield` function in Liskov's language CLU, a language that is more than 20 years old and yet contains features that still haven't been widely exploited by the CLU-less.

2. The basic Fibonacci series is a sequence of integers, starting with two 1s, in which each subsequent term is the sum of the two preceding terms. The series is sometimes used in sorting algorithms and in analyzing natural phenomena.

In this example, the `yield` statement has a parameter. This value is passed to the associated block. In the definition of the block, the argument list appears between vertical bars. In this instance, the variable `f` receives the value passed to the `yield`, so the block prints successive members of the series. (This example also shows parallel assignment in action. We'll come back to this on page 85.) Although it is common to pass just one value to a block, this is not a requirement; a block may have any number of arguments.

If the parameters to a block are existing local variables, those variables will be used as the block parameters, and their values may be changed by the block's execution. The same thing applies to variables inside the block: if they appear for the first time in the block, they're local to the block. If instead they first appeared outside the block, the variables will be shared between the block and the surrounding environment.[3]

In this (contrived) example, we see that the block inherits the variables `a` and `b` from the surrounding scope, but `c` is local to the block (the method `defined?` returns `nil` if its argument is not defined).

```
a = [1, 2]
b = 'cat'
a.each {|b| c = b * a[1] }
a             →   [1, 2]
b             →   2
defined?(c)   →   nil
```

A block may also return a value to the method. The value of the last expression evaluated in the block is passed back to the method as the value of the `yield`. This is how the `find` method used by class `Array` works.[4] Its implementation would look something like the following.

```
class Array
  def find
    for i in 0...size
      value = self[i]
      return value if yield(value)
    end
    return nil
  end
end

[1, 3, 5, 7, 9].find {|v| v*v > 30 }   →   7
```

This passes successive elements of the array to the associated block. If the block returns `true`, the method returns the corresponding element. If no element matches, the method returns `nil`. The example shows the benefit of this approach to iterators. The `Array`

3. Although extremely useful at times, this feature may lead to unexpected behavior and is hotly debated in the Ruby community. It is possible that Ruby 2.0 will change the way blocks inherit local variables.

4. The `find` method is actually defined in module `Enumerable`, which is mixed into class `Array`.

class does what it does best, accessing array elements, leaving the application code to concentrate on its particular requirement (in this case, finding an entry that meets some mathematical criteria).

Some iterators are common to many types of Ruby collections. We've looked at `find` already. Two others are `each` and `collect`. `each` is probably the simplest iterator—all it does is yield successive elements of its collection.

```
[ 1, 3, 5, 7, 9 ].each {|i| puts i }
```

produces:

```
1
3
5
7
9
```

The `each` iterator has a special place in Ruby; on page 97 we'll describe how it's used as the basis of the language's `for` loop, and starting on page 113 we'll see how defining an `each` method can add a whole lot more functionality to your class for free.

Another common iterator is `collect`, which takes each element from the collection and passes it to the block. The results returned by the block are used to construct a new array. For instance:

```
["H", "A", "L"].collect {|x| x.succ }   →   ["I", "B", "M"]
```

Iterators are not limited to accessing existing data in arrays and hashes. As we saw in the Fibonacci example, an iterator can return derived values. This capability is used by Ruby input/output classes, which implement an iterator interface that returns successive lines (or bytes) in an I/O stream. (This example uses `do...end` to define a block. The only difference between this notation and using braces to define blocks is precedence: `do...end` binds lower than `{...}`. We discuss the impact of this on page 341.)

```
f = File.open("testfile")
f.each do |line|
  puts line
end
f.close
```

produces:

```
This is line one
This is line two
This is line three
And so on...
```

1.8

Let's look at just one more useful iterator. The (somewhat obscurely named) `inject` method (defined in the module `Enumerable`) lets you accumulate a value across the

members of a collection. For example, you can sum all the elements in an array, and find their product, using code such as

```
[1,3,5,7].inject(0) {|sum, element| sum+element}          →  16
[1,3,5,7].inject(1) {|product, element| product*element}  →  105
```

`inject` works like this: the first time the associated block is called, `sum` is set to `inject`'s parameter and `element` is set to the first element in the collection. The second and subsequent times the block is called, `sum` is set to the value returned by the block on the previous call. The final value of `inject` is the value returned by the block the last time it was called. There's one final wrinkle: if `inject` is called with no parameter, it uses the first element of the collection as the initial value and starts the iteration with the second value. This means that we could have written the previous examples as

```
[1,3,5,7].inject {|sum, element| sum+element}          →  16
[1,3,5,7].inject {|product, element| product*element}  →  105
```

Internal and External Iterators

It's worth spending a paragraph comparing Ruby's approach to iterators to that of languages such as C++ and Java. In the Ruby approach, the iterator is internal to the collection—it's simply a method, identical to any other, that happens to call `yield` whenever it generates a new value. The thing that uses the iterator is just a block of code associated with this method.

In other languages, collections don't contain their own iterators. Instead, they generate external helper objects (for example, those based on Java's `Iterator` interface) that carry the iterator state. In this, as in many other ways, Ruby is a transparent language. When you write a Ruby program, you concentrate on getting the job done, not on building scaffolding to support the language itself.

It's probably also worth spending a paragraph looking at why Ruby's internal iterators aren't always the best solution. One area where they fall down badly is where you need to treat an iterator as an object in its own right (for example, passing the iterator into a method that needs to access each of the values returned by that iterator). It's also difficult to iterate over two collections in parallel using Ruby's internal iterator scheme. Fortunately, Ruby 1.8 comes with the `Generator` library (described on page 662), which implements external iterators in Ruby for just such occasions.

1.8

Blocks for Transactions

Although blocks are often the target of an iterator, they also have other uses. Let's look at a few.

You can use blocks to define a chunk of code that must be run under some kind of transactional control. For example, you'll often open a file, do something with its contents, and then want to ensure that the file is closed when you finish. Although you can do this

using conventional code, an argument exists for making the file responsible for closing itself. We can do this with blocks. A naive implementation (ignoring error handling) could look something like the following.

```
class File
  def File.open_and_process(*args)
    f = File.open(*args)
    yield f
    f.close()
  end
end


File.open_and_process("testfile", "r") do |file|
  while line = file.gets
    puts line
  end
end
```

produces:

```
This is line one
This is line two
This is line three
And so on...
```

`open_and_process` is a *class method*—it may be called independently of any particular file object. We want it to take the same arguments as the conventional `File.open` method, but we don't really care what those arguments are. To do this, we specified the arguments as `*args`, meaning "collect the actual parameters passed to the method into an array named `args`." We then call `File.open`, passing it `*args` as a parameter. This expands the array back into individual parameters. The net result is that `open_and_process` transparently passes whatever parameters it received to `File.open`.

Once the file has been opened, `open_and_process` calls `yield`, passing the open file object to the block. When the block returns, the file is closed. In this way, the responsibility for closing an open file has been passed from the user of file objects back to the files themselves.

The technique of having files manage their own life cycle is so useful that the class `File` supplied with Ruby supports it directly. If `File.open` has an associated block, then that block will be invoked with a file object, and the file will be closed when the block terminates. This is interesting, as it means that `File.open` has two different behaviors: when called with a block, it executes the block and closes the file. When called without a block, it returns the file object. This is made possible by the method `Kernel.block_given?`, which returns `true` if a block is associated with the current method. Using this method, you could implement something similar to the standard `File.open` (again, ignoring error handling) using the following.

```
class File
  def File.my_open(*args)
    result = file = File.new(*args)
    # If there's a block, pass in the file and close
    # the file when it returns
    if block_given?
      result = yield file
      file.close
    end
    return result
  end
end
```

This has one last twist: in the previous examples of using blocks to control resources, we haven't addressed error handling. If we wanted to implement these methods properly, we'd need to ensure that we closed files even if the code processing that file somehow aborted. We do this using exception handling, which we talk about later (starting on page 101).

Blocks Can Be Closures

Let's get back to our jukebox for a moment (remember the jukebox?). At some point we'll be working on the code that handles the user interface—the buttons that people press to select songs and control the jukebox. We'll need to associate actions with those buttons: press START and the music starts. It turns out that Ruby's blocks are a convenient way to do this. Let's start by assuming that the people who made the hardware implemented a Ruby extension that gives us a basic button class. (We talk about extending Ruby beginning on page 261.)

```
start_button = Button.new("Start")
pause_button = Button.new("Pause")
# ...
```

What happens when the user presses one of our buttons? In the `Button` class, the hardware folks rigged things so that a callback method, `button_pressed`, will be invoked. The obvious way of adding functionality to these buttons is to create subclasses of `Button` and have each subclass implement its own `button_pressed` method.

```
class StartButton < Button
  def initialize
    super("Start")       # invoke Button's initialize
  end
  def button_pressed
    # do start actions...
  end
end
start_button = StartButton.new
```

This has two problems. First, this will lead to a large number of subclasses. If the interface to `Button` changes, this could involve us in a lot of maintenance. Second, the actions performed when a button is pressed are expressed at the wrong level; they are not a feature of the button but are a feature of the jukebox that uses the buttons. We can fix both of these problems using blocks.

```
songlist = SongList.new
class JukeboxButton < Button
  def initialize(label, &action)
    super(label)
    @action = action
  end
  def button_pressed
    @action.call(self)
  end
end
start_button = JukeboxButton.new("Start") { songlist.start }
pause_button = JukeboxButton.new("Pause") { songlist.pause }
```

The key to all this is the second parameter to `JukeboxButton#initialize`. If the last parameter in a method definition is prefixed with an ampersand (such as `&action`), Ruby looks for a code block whenever that method is called. That code block is converted to an object of class `Proc` and assigned to the parameter. You can then treat the parameter as any other variable. In our example, we assigned it to the instance variable `@action`. When the callback method `button_pressed` is invoked, we use the `Proc#call` method on that object to invoke the block.

So what exactly do we have when we create a `Proc` object? The interesting thing is that it's more than just a chunk of code. Associated with a block (and hence a `Proc` object) is all the context in which the block was *defined*: the value of `self` and the methods, variables, and constants in scope. Part of the magic of Ruby is that the block can still use all this original scope information even if the environment in which it was defined would otherwise have disappeared. In other languages, this facility is called a *closure*.

Let's look at a contrived example. This example uses the method `lambda`, which converts a block to a `Proc` object.

```
def n_times(thing)
  return lambda {|n| thing * n }
end

p1 = n_times(23)
p1.call(3)   →   69
p1.call(4)   →   92
p2 = n_times("Hello ")
p2.call(3)   →   "Hello Hello Hello "
```

The method `n_times` returns a `Proc` object that references the method's parameter, `thing`. Even though that parameter is out of scope by the time the block is called, the parameter remains accessible to the block.

Containers Everywhere

Containers, blocks, and iterators are core concepts in Ruby. The more you write in Ruby, the more you'll find yourself moving away from conventional looping constructs. Instead, you'll write classes that support iteration over their contents. And you'll find that this code is compact, easy to read, and a joy to maintain.