Chapter 4

Containers, Blocks, and Iterators

Most real programs deal with collections of data: the people in a course, the songs in your playlist, the books in the store. Ruby comes with two built-in classes to handle these collections: arrays and hashes.[1] Mastery of these two classes is key to being an effective Ruby programmer. This mastery may take some time, because both classes have large interfaces.

But it isn't just these classes that give Ruby its power when dealing with collections. Ruby also has a block syntax that lets you encapsulate chunks of code. When paired with collections, these blocks become powerful iterator constructs. In this chapter, we'll look at the two collection classes as well as blocks and iterators.

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. (In the code examples that follow, we're often going to show the value of expressions such as a[0] in a comment at the end of the line. If you simply typed this fragment of code into a file and executed it using Ruby, you'd see no output—you'd need to add something like a call to puts to have the values written to the console.)

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

1. Some languages call hashes *associative arrays* or *dictionaries*.

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

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

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:

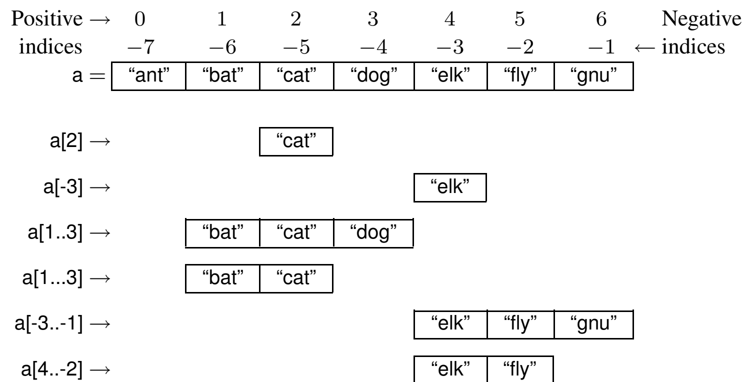
```
a = [ 1, 3, 5, 7, 9 ]
a[1..3]     # =>   [3, 5, 7]
a[1...3]    # =>   [3, 5]
a[3..3]     # =>   [7]
a[-3..-1]   # =>   [5, 7, 9]
```

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:

```
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.

Figure 4.1. How Arrays Are Indexed




```
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 them, you can treat arrays as stacks, sets, queues, dequeues, and FIFO queues.

For example, push and pop add and remove elements from the end of an array, so you can use it as a stack:

```
stack = []
stack.push "red"
stack.push "green"
stack.push "blue"
p stack
puts stack.pop
puts stack.pop
puts stack.pop
p stack
```

produces:

```
["red", "green", "blue"]
blue
green
red
[]
```

Similarly, unshift and shift add and remove elements from the head of an array. Combine shift and push, and you have a first-in first-out (FIFO) queue:

```
queue = []
queue.push "red"
queue.push "green"
puts queue.shift
puts queue.shift
```

produces:

```
red
green
```

The first and last methods return the *n* entries at the head or end of an array without removing them:

```
array = [ 1, 2, 3, 4, 5, 6, 7 ]
p array.first(4)
p array.last(4)
```

produces:

```
[1, 2, 3, 4]
[4, 5, 6, 7]
```

A complete list of array methods starts on page 447. It is well worth firing up irb and playing with them.

Hashes

Hashes (sometimes known as *associative arrays*, *maps*, or *dictionaries*) are similar to arrays in that they are indexed collections of object references. However, although you index arrays with integers, you can index a hash with objects of any type: symbols, strings, regular expressions, and so on. When you store a value in a hash, you actually supply two objects—the index, which is normally called the *key*, and the entry to be stored with that key. You can subsequently retrieve the entry by indexing the hash with the same key value that you used to store it.

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          # =>   {"dog"=>"canine", "cat"=>99, "donkey"=>"asinine",
                  "cow"=>"bovine", 12=>"dodecine"}
```

In the previous example, the hash keys were strings. If instead we wanted them to be symbols, we could write the hash literal using either the old syntax with => or the new *key: value* syntax introduced in Ruby 1.9.

1.9

```
h = { dog: 'canine', cat: 'feline', donkey: 'asinine' }
# same as...
h = { :dog => 'canine', :cat => 'feline', :donkey => 'asinine' }
```

1.9

Compared with arrays, hashes have one significant advantage: they can use any object as an index. And, as of Ruby 1.9, you'll find something that might be surprising: Ruby remembers the order in which you add items to a hash. When you subsequently iterate over the entries, Ruby will return them in that order.

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 533.

Word Frequency: Using Hashes and Arrays

Let's round off this section with a simple program that calculates the number of times each word occurs in some text. (So, for example, in this sentence the word *the* occurs two times.)

The problem breaks down into two parts. First, given some text as a string, return a list of words. That sounds like an array. Then, build a count for each distinct word. That sounds like a use for a hash—we can index it with the word and use the corresponding entry to keep a count.

Let's start with the method that splits a string into words:

```
def words_from_string(string)
  string.downcase.scan(/[\w']+/)
end
```

This method uses two very useful String methods: downcase returns a lowercase version of a string, and scan returns an array of substrings that match a given pattern. In this case, the pattern is [\w']+, which matches sequences containing "word characters" and single quotes.

We can play with this method. Notice how the result is an array:

```
p words_from_string("But I didn't inhale, he said (emphatically)")
```

produces:

```
["but", "i", "didn't", "inhale", "he", "said", "emphatically"]
```

Our next task is to calculate word frequencies. To do this, we'll create a hash object indexed by the words in our list. Each entry in this hash stores the number of times that word occurred. Let's say we already have read part of the list, and we have seen the word *the* already. Then we'd have a hash that contained this:

```
{ ...,  "the" => 1, ... }
```

If the variable next_word contained the word *the*, then incrementing the count is as simple as this:

```
counts[next_word] += 1
```

We'd then end up with a hash containing the following:

```
{ ...,  "the" => 2, ... }
```

Our only problem is what to do when we encounter a word for the first time. We'll try to increment the entry for that word, but there won't be one, so our program will fail. There are a number of solutions to this. One is to check to see whether the entry exists before doing the increment:

```
if counts.has_key?(next_word)
  counts[next_word] += 1
else
  counts[next_word] = 1
end
```

However, there's a tidier way. If we create a hash object using Hash.new(0), the parameter (0 in this case) will be used as the hash's default value—it will be the value returned if you look up a key that isn't yet in the hash. Using that, we can write our count_frequency method:

```
def count_frequency(word_list)
  counts = Hash.new(0)
  for word in word_list
    counts[word] += 1
  end
  counts
end
p count_frequency(["sparky", "the", "cat", "sat", "on", "the", "mat"])
```

produces:

```
{"sparky"=>1, "the"=>2, "cat"=>1, "sat"=>1, "on"=>1, "mat"=>1}
```

One little job left. The hash containing the word frequencies is ordered based on the first time it sees each word. It would be better to display the results based on the frequencies of the words. We can do that using the hash's sort_by method. When you use sort_by, you give it a block that tells the sort what to use when making comparisons. In our case, we'll just use the count. The result of the sort is an array containing a set of two-element arrays, each subarray corresponding to a key/entry pair in the original hash. This makes our whole program:

Download **samples/tutcontainers_21.rb**

```
def words_from_string(string)
  string.downcase.scan(/[\w']+/)
end
def count_frequency(word_list)
  counts = Hash.new(0)
  for word in word_list
    counts[word] += 1
  end
  counts
end
raw_text  = File.read("para.txt")
word_list = words_from_string(raw_text)
counts    = count_frequency(word_list)
sorted    = counts.sort_by {|word, count| count}
top_five  = sorted.last(5)
```

```
for i in 0...5           # (this is ugly code
  word = top_five[i][0]  #  which we'll fix shortly)
  count = top_five[i][1]
  puts "#{word}:  #{count}"
end
```

produces:

```
that:  2
sounds:  2
like:  2
the:  3
a:  6
```

At this point, a quick test may be in order. To do this, we're going to use a testing framework called Test::Unit 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 198). For now, we'll just say that the method assert_equal checks that its two parameters are equal, complaining bitterly if they aren't. We'll use assertions to test our two methods, one method at a time. (That's one reason why we wrote them as separate methods—it makes them testable in isolation.)

Here are some tests for the word_from_string method:

Download **samples/tutcontainers_22.rb**

```
require_relative  'words_from_string.rb'
require 'test/unit'
class TestWordsFromString < Test::Unit::TestCase
  def test_empty_string
    assert_equal([], words_from_string(""))
    assert_equal([], words_from_string("     "))
  end
  def test_single_word
    assert_equal(["cat"], words_from_string("cat"))
    assert_equal(["cat"], words_from_string("  cat   "))
  end
  def test_many_words
    assert_equal(["the", "cat", "sat", "on", "the", "mat"],
         words_from_string("the cat sat on the mat"))
  end
  def test_ignores_punctuation
    assert_equal(["the", "cat's", "mat"],
         words_from_string("<the!> cat's, -mat-"))
  end
end
```

produces:

```
Loaded suite /tmp/prog
Started
....
Finished in 0.000578 seconds.

4 tests, 6 assertions, 0 failures, 0 errors, 0 skips
```

The test starts by requiring the source file containing our words_from_string method, along with the unit test framework itself. It then defines a test class. Within that class, any methods whose names start test are automatically run by the testing framework. The results show that four test methods ran, successfully executing six assertions:

Download **samples/tutcontainers_23.rb**

```
require_relative  'count_frequency.rb'
require 'test/unit'
class TestCountFrequency < Test::Unit::TestCase
  def test_empty_list
    assert_equal({}, count_frequency([]))
  end
  def test_single_word
    assert_equal({"cat" => 1}, count_frequency(["cat"]))
  end
  def test_two_different_words
    assert_equal({"cat" => 1, "sat" => 1},
                 count_frequency(["cat", "sat"]))
  end
  def test_two_words_with_adjacent_repeat
    assert_equal({"cat" => 2, "sat" => 1},
                 count_frequency(["cat", "cat", "sat"]))
  end
  def test_two_words_with_non_adjacent_repeat
    assert_equal({"cat" => 2, "sat" => 1},
                 count_frequency(["cat", "sat", "cat"]))
  end
end
```

produces:

```
Loaded suite /tmp/prog
Started
.....
Finished in 0.000534 seconds.

5 tests, 5 assertions, 0 failures, 0 errors, 0 skips
```

Blocks and Iterators

In our program that wrote out the results of our word frequency analysis, we had the following loop:

```
for i in 0...5
  word = top_five[i][0]
  count = top_five[i][1]
  puts "#{word}:  #{count}"
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 magically knows that we're iterating over five elements, and it retrieves values in turn from the array. To do this, it has to know that the structure it is working with is an array of two-element subarrays. This is a whole lot of coupling.

Instead, we could write this code like this:

```
top_five.each do |word, count|
  puts "#{word}:  #{count}"
end
```

The method each is an *iterator*—a method that invokes a block of code repeatedly. In fact, some Ruby programmers might write this more compactly as this:

```
puts top_five.map { |word, count| "#{word}:  #{count}" }
```

Just how far you take this is a matter of taste. But, however you use them, iterators and code blocks are among the more interesting features of Ruby, so let's spend a while looking into them.

Blocks

A *block* is simply a chunk of code enclosed between either braces or the keywords do and end. The two forms are identical except for precedence, which we'll see in a minute. All things being equal, the current Ruby style seems to favor using braces for blocks that fit on one line and do/end when a block spans multiple lines:

```
some_array.each {|value| puts value * 3 }
sum = 0
other_array.each do |value|
  sum += value
  puts value / sum
end
```

You can think of a block as being somewhat like the body of an anonymous method. Just like a method, the block can take parameters (but, unlike a method, those parameters appear at the start of the block between vertical bars). Both the blocks in the preceding example take a single parameter, value. And, just like a method, the body of a block is not executed when Ruby first sees it. Instead, the block is saved away to be called later.

Blocks can appear in Ruby source code only immediately after the *invocation* of some method. If the method takes parameters, the block appears after these. In a way, you can almost think of the block as being one extra parameter, passed to that method. Let's look at a simple example that sums the squares of the numbers in an array:

```
sum = 0
[1, 2, 3, 4].each do |value|
  square = value * value
  sum   += square
end
puts sum
```

produces:

```
30
```

The block is being called by the each method once for each element in the array. The element is passed to the block as the value parameter. But there's something subtle going on, too. Take a look at the sum variable. It's declared outside the block, updated inside the block, and then passed to puts after the each method returns.

This illustrates an important rule: if there's a variable inside a block with the same name as a variable in the same scope outside the block, the two are the same—there's only one variable sum in the preceding program. (You can override this behavior, as we'll see later.)

If, however, a variable appears only inside a block, then that variable is local to the block—in the preceding program, we couldn't have written the value of square at the end of the code, because square is not defined at that point. It is defined only inside the block itself.

Although simple, this behavior can lead to unexpected problems. For example, say our program was dealing with drawing different shapes. We might have this:

```
square = Shape.new(sides: 4) # assume Shape defined elsewhere
#
# .. lots of code
#
sum = 0
[1, 2, 3, 4].each do |value|
  square = value * value
  sum   += square
end
puts sum
square.draw   # BOOM!
```

This code would fail, because the variable square, which originally held a Shape object, will have been overwritten inside the block and will hold a number by the time the each method returns. This problem doesn't bite often, but when it does, it can be very confusing.

1.9

Fortunately, Ruby 1.9 has a couple of answers.

First, parameters to a block are now *always* local to a block, even if they have the same name as locals in the surrounding scope. (You'll get a warning message if you run Ruby with the -w option.)

Download **samples/tutcontainers_30.rb**

```
value = "some shape"
[ 1, 2 ].each {|value| puts value }
puts value
```

produces:

```
1
2
some shape
```

Second, you can now define block local variables by putting them after a semicolon in the block's parameter list. So, in our sum-of-squares example, we should have indicated that the square variable was block-local by writing it as follows:

Download samples/tutcontainers_31.rb

```
square = "some shape"
sum = 0
[1, 2, 3, 4].each do |value; square|
  square = value * value    # this is a different variable
  sum   += square
end
puts sum
puts square
```

produces:

```
30
some shape
```

By making square block-local, values assigned inside the block will not affect the value of the variable with the same name in the outer scope.

Implementing Iterators

A Ruby iterator is simply a method that can invoke a block of code.

We said that a block may appear only in the source adjacent to a method call and that 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.[2] Let's start with a trivial example:

Download samples/tutcontainers_32.rb

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

produces:

```
Hello
Hello
Hello
```

---

2. 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 thirty years old and yet contains features that still haven't been widely exploited by the CLU-less.

The block (the code between the braces) is associated with the call to the three_times method. 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:[3]

Download **samples/tutcontainers_33.rb**

```
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
```

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 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 151.) 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.

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 (that is, a value other than nil or false), the method returns the corresponding element. If no

3. 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.

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

element matches, the method returns nil. The example shows the benefit of this approach to iterators. The Array class does what it does best, accessing array elements, and leaves the application code to concentrate on its particular requirement (in this case, finding an entry that meets some 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 162, we'll describe how it's used as the basis of the language's for loop, and starting on page 100, 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 (also known as map), 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. The following example uses the succ method, which increments a string value:

```
["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:

```
f = File.open("testfile")
f.each do |line|
  puts "The line is: #{line}"
end
f.close
```

produces:

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

Sometimes you want to keep track of how many times you've been through the block. The each_with_index is your friend. It calls its block with two parameters: the current element of the iteration and the count (which starts at zero, just like array indices):

```
f = File.open("testfile")
f.each_with_index do |line, index|
  puts "Line #{index} is: #{line}"
end
f.close
```

produces:

```
Line 0 is: This is line one
Line 1 is: This is line two
Line 2 is: This is line three
Line 3 is: And so on...
```

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 this:

```
[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. One more thing: 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 like this:

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

1.9

And, just to add to the mystique of inject, you can also give it the name of the method you want to apply to successive elements of the collection. These examples work because, in Ruby, addition and multiplication are simply methods on numbers, and :+ is the symbol corresponding to the method +:

```
[1,3,5,7].inject(:+)   # =>   16
[1,3,5,7].inject(:*)   # =>   105
```

Enumerators—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 basic 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 a call to this method.

In other languages, collections don't contain their own iterators. Instead, they implement methods that 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 also worth spending another 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.

1.9 Fortunately, Ruby 1.9 comes with a built-in Enumerator class, which implements external iterators in Ruby for just such occasions.

One way to create an Enumerator object is to call the to_enum method (or its synonym, enum_for) on a collection such as an array or a hash:

```
a = [ 1, 3, "cat" ]
h = { dog: "canine", fox: "lupine" }

# Create Enumerators
enum_a = a.to_enum
enum_h = h.to_enum

enum_a.next   # =>   1
enum_h.next   # =>   [:dog, "canine"]
enum_a.next   # =>   3
enum_h.next   # =>   [:fox, "lupine"]
```

Most of the internal iterator methods—the ones that normally yield successive values to a block—will also return an Enumerator object if called without a block:

```
a = [ 1, 3, "cat" ]

enum_a = a.each # create an Enumerator using an internal iterator

enum_a.next   # =>   1
enum_a.next   # =>   3
```

Ruby has a method called loop that does nothing but repeatedly invoke its block. Typically, your code in the block will break out of the loop when some condition occurs. But loop is also smart when you use an Enumerator—when an enumerator object runs out of values inside a loop, the loop will terminate cleanly. The following example shows this in action—the loop ends when the three-element enumerator runs out of values.[5]

```
short_enum = [1, 2, 3].to_enum
long_enum  = ('a'..'z').to_enum
loop do
  puts "#{short_enum.next} - #{long_enum.next}"
end
```

produces:

```
1 - a
2 - b
3 - c
```

5. You can also handle this in your own iterator methods by rescuing the StopIteration exception, but because we haven't talked about exceptions yet, we won't go into details here.

Enumerators Are Objects

Enumerators take something that's normally executable code (the act of iterating) and turn it into an object. This means that you can do things programatically with enumerators that aren't easily done with regular loops.

For example, the Enumerable module defines each_with_index. This invokes its host class's each method, returning successive values along with an index:

```
result = []
[ 'a', 'b', 'c' ].each_with_index {|item, index| result << [item, index] }
result   # =>   [["a", 0], ["b", 1], ["c", 2]]
```

But what if you wanted to iterate and receive an index but use a different method than each to control that iteration? For example, you might want to iterate over the characters in a string. There's no method called each_char_with_index built into the String class.

Enumerators to the rescue. You can use the fact that the each_char method of strings will return an enumerator if you don't give it a block, and you can then call each_with_index on that enumerator:

```
result = []
"cat".each_char.each_with_index {|item, index| result << [item,  index] }
result   # =>   [["c", 0], ["a", 1], ["t", 2]]
```

In fact, this is such a common use of enumerators that Matz has given us with_index, which makes the code read better:

```
result = []
"cat".each_char.with_index {|item, index| result << [item,  index] }
result   # =>   [["c", 0], ["a", 1], ["t", 2]]
```

You can also create the Enumerator object explicitly—in this case we'll create one that will call our string's each_char method. We can call to_a on that enumerator to iterate over it and get the result:

```
enum = "cat".enum_for(:each_char)
enum.to_a   # =>   ["c", "a", "t"]
```

If the method we're using as the basis of our enumerator takes parameters, we can pass them to enum_for:

```
enum_good = (1..10).enum_for(:each_slice, 3)
enum_good.to_a   # =>   [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10]]
```

Enumerators Are Generators and Filters

(This is more advanced material that can be skipped on first reading.) As well as creating enumerators from existing collections, you can create an explicit enumerator, passing it a block. The code in the block will be used when the enumerator object needs to supply a fresh value to your program. However, the block isn't simply executed from top to bottom. Instead, the block is executed in parallel with the rest of your program's code. Execution starts at the top and pauses when the block yields a value to your code. When the code needs the next value, execution resumes at the statement following the yield. This lets you write enumerators that generate infinite sequences (among other things):

Download **samples/tutcontainers_50.rb**

```
triangular_numbers = Enumerator.new do |yielder|
  number = 0
  count = 1
  loop do
    number += count
    count += 1
    yielder.yield number
  end
end
5.times { puts triangular_numbers.next }
```

produces:

```
1
3
6
10
15
```

Enumerator objects are also enumerable (that is to say, the methods available to enumerable objects are also available to them). That means we can use enumerable's methods (such as first) on them:

```
triangular_numbers = Enumerator.new do |yielder|
  # ...
end
p triangular_numbers.first(5)
```

produces:

```
[1, 3, 6, 10, 15]
```

You have to be slightly careful with enumerators that can generate infinite sequences. Some of the regular enumerator methods such as count and select will happily try to read the whole enumeration before returning a result. If you want a version of select that works with infinite sequences, you'll need to write it yourself. Here's a version that gets passed an enumerator and a block and returns a new enumerator containing values from the original for which the block returns true. We'll use it to return triangular numbers that are multiples of 10.

Download **samples/tutcontainers_52.rb**

```
triangular_numbers = Enumerator.new do |yielder|
  # ... as before
end
def infinite_select(enum, &block)
  Enumerator.new do |yielder|
    enum.each do |value|
      yielder.yield(value) if block.call(value)
    end
  end
end
p infinite_select(triangular_numbers) {|val| val % 10 == 0}.first(5)
```

produces:

```
[10, 120, 190, 210, 300]
```

Here we use the &block notation to pass the block as a parameter to the infinite_select method.

As Brian Candler pointed out in [ruby-core:19679], you can make this more convenient by adding filters such as infinite_select directly to the Enumerator class. Here's an example that returns the first five triangular numbers that are multiples of 10 and that have the digit 3 in them:

Download **samples/tutcontainers_53.rb**

```
triangular_numbers = Enumerator.new do |yielder|
  # ... as before
end
class Enumerator
  def infinite_select(&block)
    Enumerator.new do |yielder|
      self.each do |value|
        yielder.yield(value) if block.call(value)
      end
    end
  end
end
p triangular_numbers
   .infinite_select {|val| val % 10 == 0}
   .infinite_select {|val| val.to_s =~ /3/ }
   .first(5)
```

produces:

```
[300, 630, 1830, 3160, 3240]
```

Blocks for Transactions

Although blocks are often used as the target of an iterator, they 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 linear code, a version using blocks is simpler (and turns out to be less error prone). A naive implementation (ignoring error handling) could look something like the following:

Download **samples/tutcontainers_54.rb**

```
class File
  def self.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 receives 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 shifted from the users of file objects back to the file objects 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, because 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 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:

Download **samples/tutcontainers_55.rb**

```
class File
  def self.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 didn't address error handling. If we wanted to implement these methods properly, we'd need

to ensure that we closed a file even if the code processing that file somehow aborted. We do this using exception handling, which we talk about later (starting on page 167).

Blocks Can Be Objects

Blocks are like anonymous methods, but there's more to them than that. You can also convert a block into an object, store it in variables, pass it around, and then invoke its code sometime later.

Remember I said that you can think of blocks as being a little like an implicit parameter that's passed to a method? Well, you can also make that parameter explicit. 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.

Here's an example where we create a Proc object in one instance method and store it in an instance variable. We then invoke the proc from a second instance method.

Download **samples/tutcontainers_56.rb**

```
class ProcExample
  def pass_in_block(&action)
    @stored_proc = action
  end
  def use_proc(parameter)
    @stored_proc.call(parameter)
  end
end
eg = ProcExample.new
eg.pass_in_block { |param| puts "The parameter is #{param}" }
eg.use_proc(99)
```

produces:

```
The parameter is 99
```

See how the call method on a proc object invokes the code in the original block?

Many Ruby programs store and later call blocks in this way—it's a great way of implementing callbacks, dispatch tables, and so on.

But, you can go one step further. If a block can be turned into an object by adding an ampersand parameter to a method, what happens if that method then returns the Proc object to the caller?

Download **samples/tutcontainers_57.rb**

```
def create_block_object(&block)
  block
end
```

```
bo = create_block_object { |param| puts "You called me with #{param}" }
bo.call 99
bo.call "cat"
```

produces:

```
You called me with 99
You called me with cat
```

In fact, this is so useful that Ruby provides not one but two built-in methods that convert a block to an object.[6] Both lambda and Proc.new take a block and return an object of class Proc. The objects they return differ slightly in how they behave, but we'll hold off talking about that until page 364.

Download **samples/tutcontainers_58.rb**

```
bo = lambda { |param| puts "You called me with #{param}" }
bo.call 99
bo.call "cat"
```

produces:

```
You called me with 99
You called me with cat
```

Blocks Can Be Closures

Remember I said that a block can use local variables from the surrounding scope? So, let's look at a slightly different example of a block doing just that:

Download **samples/tutcontainers_59.rb**

```
def n_times(thing)
  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. This is called a *closure*—variables in the surrounding scope that are referenced in a block remain accessible for the life of that block and the life of any Proc object created from that block.

6. There's actually a third, proc, but it is effectively deprecated.

Here's another example, which is a method that returns a Proc object that returns successive powers of 2 when called:

Download **samples/tutcontainers_60.rb**

```
def power_proc_generator
  value = 1
  lambda { value += value }
end

power_proc = power_proc_generator

puts power_proc.call
puts power_proc.call
puts power_proc.call
```

produces:

```
2
4
8
```

An Alternative Notation

1.9 Ruby 1.9 has another way of creating Proc objects. Rather than write this:

```
lambda { |params| ... }
```

you can now write the following:[7]

```
->params { ... }
```

The parameters can be enclosed in optional parentheses. For example:

Download **samples/tutcontainers_63.rb**

```
proc1 = -> arg { puts "In proc1 with #{arg}" }
proc2 = -> arg1, arg2 { puts "In proc2 with #{arg1} and #{arg2}" }
proc3 = ->(arg1, arg2) { puts "In proc3 with #{arg1} and #{arg2}" }

proc1.call "ant"
proc2.call "bee", "cat"
proc3.call "dog", "elk"
```

produces:

```
In proc1 with ant
In proc2 with bee and cat
In proc3 with dog and elk
```

---

7. Let's start by getting something out of the way. Why ->? For compatibility across all the different source file encodings, Matz is restricted to using pure 7-bit ASCII for Ruby operators, and the choice of available characters is severely limited by the ambiguities inherent in the Ruby syntax. He felt that -> was (kind of) reminiscent of a Greek lambda character λ.

The -> form is more compact than using lambda and seems to be in favor when you want to pass one or more Proc objects to a method:

Download samples/tutcontainers_64.rb

```
def my_if(condition, then_clause, else_clause)
  if condition
    then_clause.call
  else
    else_clause.call
  end
end
5.times do |val|
  my_if val < 3,
        -> { puts "#{val} is small" },
        -> { puts "#{val} is big" }
end
```

produces:

```
0 is small
1 is small
2 is small
3 is big
4 is big
```

One good reason to pass blocks to methods is that you can reevaluate the code in those blocks at any time. Here's a trivial example of reimplementing a while loop using a method. Because the condition is passed as a block, it can be evaluated each time around the loop:

Download samples/tutcontainers_65.rb

```
def my_while(cond, &body)
  while cond.call
    body.call
  end
end
a = 0
my_while -> { a < 3 } do
   puts a
   a += 1
end
```

produces:

```
0
1
2
```

Block Parameter Lists

1.9

Prior to Ruby 1.9, blocks were to some extent the poor cousins of methods when it came to parameter lists. Methods could have splat args, default values, and block parameters, whereas blocks basically had just a list of names (and could accept a trailing splat argument). Now, however, blocks have the same parameter list capabilities as methods.

Blocks written using the old syntax take their parameter lists between vertical bars. Blocks written using the -> syntax take a separate parameter list before the block body. In both cases, the parameter list looks just like the list you can give to methods. It can take default values, splat args (described on page 143), and a block parameter (a trailing argument starting with an ampersand). You can write blocks that are just as versatile as methods.[8]

Here's a block using the original block notation:

Download **samples/tutcontainers_66.rb**

```
proc1 = lambda do |a, *b, &block|
  puts "a = #{a.inspect}"
  puts "b = #{b.inspect}"
  block.call
end
proc1.call(1, 2, 3, 4) { puts "in block1" }
```

produces:

```
a = 1
b = [2, 3, 4]
in block1
```

And here's one using the new -> notation:

Download **samples/tutcontainers_67.rb**

```
proc2 = -> a, *b, &block do
  puts "a = #{a.inspect}"
  puts "b = #{b.inspect}"
  block.call
end
proc2.call(1, 2, 3, 4) { puts "in block2" }
```

produces:

```
a = 1
b = [2, 3, 4]
in block2
```

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. If this all seems too weird, don't worry. After a while, it'll start to come naturally. And you'll have plenty of time to practice as you use Ruby libraries and frameworks.

8. Actually, they are more versatile, because these blocks are also closures, while methods are not.