CHAPTER 17

Use Blocks to Iterate

For programmers new to Ruby, code blocks are generally the first sign that they have definitely departed Kansas. Part syntax, part method, and part object, the code block is one of the key features that gives the Ruby programming language its unique feel. In fact, code blocks are kind of the Swiss Army Knife of Ruby programming; we use them for everything from initializing objects to building DSLs. In the next few chapters we will take a hard look at code blocks and the things you can do with them. The idea is to get beyond simply passing blocks to other people's methods and move on to squeezing the last bit of utility out of the code blocks that get passed into your own methods.

We're going to kick things off by looking at the most familiar use of code blocks—as iterators. After a quick review of the mechanics of code blocks we will move on to building simple iterators, iterators that can sequence through garden-variety collections. From there we will move on to iterators that run through collections that never actually exist. Next, we will also explore how you can mess up a perfectly good iterator and at how a misbehaving iterator can bring your code to a screeching halt. Finally, we will close by taking a quick tour of the myriad iterating code blocks that you will find in existing Ruby programs.

A Quick Review of Code Blocks

Since code blocks are one of the higher speed bumps on the road to real Ruby fluency, let's take a minute to review the basics. In Ruby you create code blocks by tacking them on to the end of a method call, like this:

```
do_something do
  puts "Hello from inside the block"
end
```

Or this:

```
do_something { puts "Hello from inside the block" }
```

When you tack a block onto the end of a method call, Ruby will package up the block as sort of a secret argument and (behind the scenes) passes this secret argument to the method. Inside the method you can detect whether your caller has actually passed in a block with the `block_given?` method and fire off the block (if there is one) with `yield`:

```
def do_something
  yield if block_given?
end
```

Blocks can take arguments, which you supply as arguments to `yield`, so that if you do this:

```
def do_something_with_an_arg
  yield("Hello World") if block_given?
end

do_something_with_an_arg do |message|
  puts "The message is #{message}"
end
```

You will get:

```
The message is Hello World
```

Finally, like most everything else in Ruby, code blocks always return a value—the last expression that the block executes—which your yielding method can either use or ignore as it sees fit. So if you run this:

```
def print_the_value_returned_by_the_block
  if block_given?
    value = yield
    puts "The block returned #{value}"
  end
end

print_the_value_returned_by_the_block { 3.14159 / 4.0 }
```

You will see the value of $\pi/4$:

```
The block returned 0.7853975
```

One Word after Another

The difference between the `do_something` and `do_something_with_an_arg` methods and a real iterator method is simple: An iterator method calls its block once for each element in some collection, passing the element into the block as a parameter. For example, we could add an iterator to our `Document` class, one that runs through all the words in the document:

```
class Document

  # Stuff omitted...

  def each_word
    word_array = words
    index = 0
    while index < words.size
      yield( word_array[index] )
      index += 1
    end
  end
end
```

The `Document` class, now sports a very respectable-looking Ruby iterator method, one that you can use like any other, so that running this:

```
d = Document.new( 'Truth', 'Gump', 'Life is like a box of ...' )
d.each_word {|word| puts word}
```

Will result in this:

```
Life
is
like
a
box
of
...
```

Although this `each_word` method is a good, simple illustration of how to build an iterating method, it does go out of its way to work hard. A real implementation of `each_word` would almost certainly take advantage of the existing `each` method in the words array:

```
  def each_word
    words.each { |word| yield( word ) }
  end
```

There's nothing like using the wheel that's already there.

As Many Iterators as You Like

Although so far we have focused on adding *an* iterator to the `Document` class, a class can have as many iterator methods as make sense. So along with `each_word`, the `Document` class might also sport an `each_character` method:[1]

```
class Document
  def each_character
```

1. If you are using a pre-1.9 version of Ruby, the `each_character` method is misnamed since before 1.9 Ruby strings were just collections of bytes (really just integers), not actual characters. So if you are using 1.8.X, the `Document each_character` method will yield a series of numbers, each number the ordinal number of a character in the string. With Ruby 1.9, strings really are strings of characters, and the `each_character` method will yield a series of one-character strings.

```
      index = 0
      while index < @content.size
        yield( @content[index] )
        index += 1
      end
    end
  end
```

You are also free to name your iterator method anything you like, but it does make good sense to follow the Ruby convention and name your most obvious or commonly used iterator `each` and give any other iterators a name like `each_something_else`. We might, therefore, decide that words are the key elements of our documents and re-jigger our code to look something like:

```
  class Document

    # Stuff omitted

    def each
      # iterate over the words as in our first example
    end

    def each_character
      # iterate over the characters
    end
  end
```

Since it's the words that count, we have made the `each` method run through the document one word at a time while consigning the method that runs through each character a lesser name.

Iterating over the Ethereal

An aspect of iterators that beginners often overlook is that you can write iterators that run through collections that don't actually exist, at least not all at the same time. The simplest example of this sort of thing is the `times` method that you find on Ruby integers:

```
  12.times { |x| puts "The number is #{x}" }
```

This code will print out the first dozen integers, but it will print them without ever assembling a twelve-element collection. Instead, the `times` method produces each number one at a time and feeds it to the block. So far, so obvious. What's not so obvious is that you can use this same trick to build your own iterators. For example, people who are interested in determining the authorship of documents will sometimes gather statistics on which words are likely to appear together in a given document. If you had a document where the word "Mmmm" was frequently followed by the word "donuts," you might guess that the author was Homer Simpson. We can help these authorship-seeking scholars by providing an `each_word_pair` method in the `Document` class:

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

  def each_word_pair
    word_array = words
    index = 0
    while index < (word_array.size-1)
      yield word_array[index], word_array[index+1]
      index += 1
    end
  end
end
```

Armed with `each_word_pair`, we can write some code to print out every pair of adjacent words in one of the world's great bits of literature:

```
doc = Document.new('Donuts', '?', 'I love donuts mmmm donuts' )
doc.each_word_pair{ |first, second| puts "#{first} #{second}" }
```

Which will produce:

```
I love
love donuts
donuts mmmm
mmmm donuts
```

Notice that we never actually build a four-element array of all the word pairs: We simply generate the pairs on the fly.

Enumerable: Your Iterator on Steroids

As I say, the Ruby convention is to name the main iterator of your class `each`, and one of the themes of this book is that you should try to stick to the conventions. There is, however, another reason to follow the crowd and name that key iterating method `each`: Doing so enables you to use the `Enumerable` module. The `Enumerable` module is a mixin that endows classes with all sorts of interesting collection-related methods. Here's how `Enumerable` works: First, you make sure that your class has an `each` method, and then you include the `Enumerable` module in your class, like this:

```
class Document
  include Enumerable

  # Most of the class omitted...

  def each
    words.each { |word| yield( word ) }
  end
end
```

The simple act of including `Enumerable` adds a plethora of collection-related methods to your class, methods that all rely on your `each` method. So, if you create an instance of the `Enumerable`-enhanced `Document`:

```
doc = Document.new('Advice', 'Harry', 'Go ahead make my day')
```

Then you can find out whether your document includes a given word with `doc.include?`, so that `doc.include?("make")` will return true, but `doc.include?( "Punk")` will return false. `Enumerable` also enhances your class with a `to_a` method that returns an array of all of the items, in our case words, in your collection. The `Enumerable` module also adds methods that help you find things in your collection, methods with names like `find` and `find_all`.

`Enumerable` also contributes the `each_cons` method to your class. The `each_cons` method takes an integer and a block, and will repeatedly call the block, each time passing in an array of consecutive elements from the collection. So by including `Enumerable` in the `Document` class, the `each_word_pair` method would reduce down to:

```
def each_word_pair
  words.each_cons(2) {|array| yield array[0], array[1] }
end
```

Along the same lines as `each_cons`, `Enumerable` also supplies `each_slice`, which simply breaks up the collection in chunks of a given size and passes those into the block. Finally, if the elements in your collection define the <=> operator, you can use the `Enumerable`-supplied `sort` method, which will return a sorted array of all the elements in your collection. Since strings do indeed define <=>, we can get a sorted list of the words in our document with `doc.sort`.[2] In all, `Enumerable` adds nearly 40 methods to your class—not a bad return on the effort of implementing one or two methods and mixing in a single module.

Nor are you simply stuck if, as with the latest `Document` class with its `each` and `each_character` and `each_word_pair`, you have more than one iterating method. Along with `Enumerable`, Ruby also comes with the `Enumerator` class. If you create an `Enumerator` instance, passing in your collection and the name of the iterating method, what you will get is an object that knows how to sequence through your collection using that method. For example, if you make a new `Enumerator` based on a `Document` instance and the `each_character` method:

```
doc = Document.new('example', 'russ', "We are all characters")
enum = Enumerator.new( doc, :each_character )
```

Then you will end up with an object with all of the nice `Enumerable` methods based on the `each_character` method. Thus you can discover the number of characters in your document text:

```
puts enum.count
```

2. Mysteriously, arrays also implement all of these methods. This might lead you to suspect that the `Array` class includes the `Enumerable` module. You would be right.

Or sort the characters:

```
    pp enum.sort
```

To produce:

```
    [" ", " ", " ", "W", "a", "a", "a", "a", "c", ...]
```

Although the names are tricky, getting the hang of `Enumerable` and `Enumerator` is well worth the effort.

Staying Out of Trouble

The primary way that an iterator method can come to grief is by trusting the block too much. Remember, the code block you get handed in your iterator method is someone else's code. You need to regard the block as something akin to a hand grenade, ready to go off at any second. We saw one aspect of this when we talked about Ruby's collection classes: What happens if the code block changes the underlying collection? As we saw in Chapter 3, the Ruby collection classes generally throw up their arms and say "Don't do that!" Our `Document` class is, however, made of stronger stuff. Since the `Document each_word` method actually creates a new array before it starts, the document can change any which way while the iteration is going on. The `each_word` method will continue to sequence through the words that were there when the iteration started.

Blocks can also blow up in your face with an exception:

```
    doc.each_word do |word|
      raise 'boom' if word == 'now'
    end
```

A stray exception will not make much difference to the `each_word` method, but what if your iterator needs to acquire—and get rid of—some expensive resource? What if you have something like:

```
    def each_name
      name_server = open_name_server   # Get some expensive resource
      while name_server.has_more?
      yield name_server.read_name
    end
    name_server.close               # Close the expensive resource
  end
```

Now you have a problem. If the code block decides to raise an exception in mid-yield, then you'll never get a chance to clean up that expensive resource. The answer to this problem is easy:

```
  def each_name
    name_server = open_name_server   # Get some expensive resource
    begin
      while name_server.has_more?
        yield name_server.read_name
      end
    ensure
      name_server.close             # Close the expensive resource
    end
  end
```

Even an exception-free block is no guarantee that your iterating method will run to completion. Ruby allows applications to call `break` in mid-block. The idea is to give the code using an iterating method a way to escape early:

```
  def count_till_tuesday( doc )
    count = 0
    doc.each_word do |word|
      count += 1
      break if word == 'Tuesday'
    end
    count
```

When called from inside of a block, `break` will trigger a return out of *the method that called the block*. An explicit `return` from inside the block triggers an even bigger jump: It causes the method that defined (not called) the block to return. This is generally what you want to simulate breaking out of or returning from a built-in loop. Fortunately, like exceptions, both `break` and `return` will trigger any surrounding `ensure` clauses.

In the Wild

The fact is, you can't swing a dead cat in the Ruby world without hitting some block-based iterators. Iterators in Ruby range from the very mundane to the fairly exotic. At the boring end of the spectrum we have the `each` method on the `Array` and `Hash` classes that we looked at in Chapter 3. Equally unexciting, but still very useful, is `each` method on the built-in `Dir` class. The `Dir` `each` method will iterate over all the files in a given directory:

```
puts "Contents of /etc directory:"
etc_dir = Dir.new("/etc")
etc_dir.each {|entry| puts entry}
```

Slightly more interesting is the `each_address` method on the `Resolv`[3] class, which is part of the Ruby standard library. The `Resolv` class looks things up in DNS for you. Thus, with the `each_address` method you can discover all of the IP addresses associated with a given domain name. Run this:

```
require 'resolv'
Resolv.each_address( "www.google.com" ) {|x| puts x}
```

And you will see something like:

```
72.14.204.104
72.14.204.147
72.14.204.99
72.14.204.103
```

At the more esoteric end of the spectrum, we have the `each_object` method on the standard class `ObjectSpace`. Called without any parameters, `ObjectSpace.each_object` will run through all the objects in your Ruby interpreter[4] (yes, all of them!).

3. Yes, without the "e".

4. Do be aware that for performance reasons, JRuby disables `ObjectSpace` by default. This is one of the few incompatibilities between JRuby and the other Ruby implementations. You can un-disable it, but it is an incompatibility nevertheless.

Call `ObjectSpace.each_object` with a class and it will iterate through all of the instances of that class. For example, to see all of the strings that your Ruby interpreter knows about, run:

```
ObjectSpace.each_object(String) { |the_string| puts the_string }
```

Finally, if your taste in esoteric iterators runs more towards the mathematical, consider the `each` method on the `Prime` class. This method will call your block once for each and every prime number:

```
require 'mathn'

# Warning: According to Euclid, this never stops...

Prime.each {|x| puts "The next prime is #{x}" }
```

Of course, if you want to see all of the prime numbers, you'll need to be patient.

Wrapping Up

In this chapter we looked at the iconic application of code blocks, as iterators. We saw how you can build as many iterator methods as you like for your classes, methods that take a block and call it for each item in some collection. We also saw how you can build iterators that sequence through collections that don't actually exist. As long as you can come up with one element after another, you can build an iterator. We saw how `Enumerable` and `Enumerator` can enhance the iterating chops of your classes by providing most of the methods any collection class could imagine. We also saw how you need to treat the code blocks that get passed to your methods with a certain level of wary respect.

So much for iterators. In the next chapter we will look at a very different use for code blocks, one that turns the idea of an iterator on its head. Iterators are concerned with delivering item after item to a code block. We will see how you can reverse the polarity of code blocks and use them to deliver code to the right spot in your program.