CHAPTER 7
Code Blocks and Iteration

In Ruby, a code block (or just "block") is an object that contains some Ruby code, and the context neccesary to execute it. Code blocks are the most visually distinctive aspect of Ruby, and also one of the most confusing to newcomers from other languages. Essentially, a Ruby code block is a method that has no name.

Most other languages have something like a Ruby code block: C's function pointers, C++'s function objects, Python's lambdas and list comprehensions, Perl's anonymous functions, Java's anonymous inner classes. These features live mostly in the corners of those languages, shunned by novice programmers. Ruby can't be written without code blocks. Of the major languages, only Lisp is more block-oriented.

Unlike most other languages, Ruby makes code blocks easy to create and imposes few restrictions on them. In every other chapter of this book, you'll see blocks passed into methods like it's no big deal (which it isn't):

```
[1,2,3].each { |i| puts i}
# 1
# 2
# 3
```

In this chapter, we'll show you how to write that kind of method, the kinds of method that are useful to write that way, and when and how to treat blocks as first-class objects.

Ruby provides two syntaxes for creating code blocks. When the entire block will fit on one line, it's most readable when enclosed in curly braces:

```
[1,2,3].each { |i| puts i }
# 1
# 2
# 3
```

When the block is longer than one line, it's more readable to begin it with the `do` keyword and end it with the `end` keyword:

```
[1,2,3].each do |i|
  if i % 2 == 0
```

```
    puts "#{i} is even."
  else
    puts "#{i} is odd."
  end
end
# 1 is odd.
# 2 is even.
# 3 is odd.
```

Some people use the bracket syntax when they're interested in the return value of the block, and the `do...end` syntax when they're interested in the block's side effects.

Keep in mind that the bracket syntax has a higher precedence than the `do..end` syntax. Consider the following two snippets of code:

```
1.upto 3 do |x|
  puts x
end
# 1
# 2
# 3

1.upto 3 { |x| puts x }
# SyntaxError: compile error
```

In the second example, the code block binds to the number 3, not to the function call `1.upto 3`. A standalone variable can't take a code block, so you get a compile error. When in doubt, use parentheses.

```
1.upto(3) { |x| puts x }
# 1
# 2
# 3
```

Usually the code blocks passed into methods are anonymous objects, created on the spot. But you can instantiate a code block as a `Proc` object by calling `lambda`. See Recipe 7.1 for more details.

```
hello = lambda { "Hello" }
hello.call
# => "Hello"

log = lambda { |str| puts "[LOG] #{str}" }
log.call("A test log message.")
# [LOG] A test log message.
```

Like any method, a block can accept arguments. A block's arguments are defined in a comma-separated list at the beginning of the block, enclosed in pipe characters:

```
{1=>2, 2=>4}.each { |k,v| puts "Key #{k}, value #{v}" }
# Key 1, value 2
# Key 2, value 4
```

Arguments to blocks look almost like arguments to methods, but there are a few restrictions: you can't set default values for block arguments, you can't expand hashes or arrays inline, and a block cannot itself take a block argument.*

Since `Proc` objects are created like other objects, you can create factory methods whose return values are customized pieces of executable Ruby code. Here's a simple factory method for code blocks that do multiplication:

```
def times_n(n)
  lambda { |x| x * n }
end
```

The following code uses the factory to create and use two customized methods:

```
times_ten = times_n(10)
times_ten.call(5)                                   # => 50
times_ten.call(1.25)                                # => 12.5

circumference = times_n(2*Math::PI)
circumference.call(10)        # => 62.8318530717959
circumference.call(3)                               # => 18.8495559215388
[1, 2, 3].collect(&circumference)
# => [6.28318530717959, 12.5663706143592, 18.8495559215388]
```

You may have heard people talking about Ruby's "closures." What is a closure, and how is it different from a block? In Ruby, there is no difference between closures and blocks. Every Ruby block is also a closure.†

So what makes a Ruby block a closure? Basically, a Ruby block carries around the context in which it was defined. A block can reference the variables that were in scope when it was defined, even if those variables later go out of scope. Here's a simple example; see Recipe 7.4 for more.

```
ceiling = 50
# Which of these numbers are less than the target?
[1, 10, 49, 50.1, 200].select { |x| x < ceiling }
# => [1, 10, 49]
```

The variable `ceiling` is within scope when the block is defined, but it goes out of scope when the flow of execution enters the `select` method. Nonetheless, the block can access `ceiling` from within `select`, because it carries its context around with it. That's what makes it a closure.

We suspect that a lot of people who say "closures" when talking about Ruby blocks just do it to sound smart. Since we've already ruined any chance we might have had

* In Ruby 1.9, a block can itself take a block argument: `|arg1, arg2, &block|`. This makes methods like `Module#define_method` more useful. In Ruby 2.0, you'll be able to give default values to block arguments.

† Someone could argue that a block isn't *really* a closure if it never actually uses any of the context it carries around: you could have done the same job with a "dumb" block, assuming Ruby supported those. For simplicity's sake, we do not argue this.

at sounding smart, we've decided to refer to Ruby closures as just plain "blocks" throughout this book. The only exceptions are in the rare places where we must discuss the context that makes Ruby's code blocks real closures, rather than "dumb" blocks.

7.1 Creating and Invoking a Block

Problem

You want to put some Ruby code into an object so you can pass it around and call it later.

Solution

By this time, you should familiar with a block as some Ruby code enclosed in curly brackets. You might think it possible to define a block object as follows:

```
aBlock = { |x| puts x }                # WRONG

# SyntaxError: compile error
```

That doesn't work because a block is only valid Ruby syntax when it's an argument to a method call. There are several equivalent methods that take a block and return it as an object. The most favored method is `Kernel#lambda`:*

```
aBlock = lambda { |x| puts x }         # RIGHT
```

To call the block, use the `call` method:

```
aBlock.call "Hello World!"
# Hello World!
```

Discussion

The ability to assign a bit of Ruby code to a variable is very powerful. It lets you write general frameworks and plug in specific pieces of code at the crucial points.

As you'll find out in Recipe 7.2, you can accept a block as an argument to a method by prepending `&` to the argument name. This way, you can write your own trivial version of the `lambda` method:

```
def my_lambda(&aBlock)
  aBlock
end

b = my_lambda { puts "Hello World My Way!" }
b.call
# Hello World My Way!
```

* The name `lambda` comes from the lambda calculus (a mathematical formal system) via Lisp.

A newly defined block is actually a Proc object.

```
b.class                                 # => Proc
```

You can also initialize blocks with the Proc constructor or the method Kernel#proc. The methods Kernel#lambda, Kernel#proc, and Proc.new all do basically the same thing. These three lines of code are nearly equivalent:

```
aBlock = Proc.new { |x| puts x }
aBlock = proc { |x| puts x }
aBlock = lambda { |x| puts x }
```

What's the difference? Kernel#lambda is the preferred way of creating block objects, because it gives you block objects that act more like Ruby methods. Consider what happens when you call a block with the wrong number of arguments:

```
add_lambda = lambda { |x,y| x + y }

add_lambda.call(4)
# ArgumentError: wrong number of arguments (1 for 2)

add_lambda.call(4,5,6)
# ArgumentError: wrong number of arguments (3 for 2)
```

A block created with lambda acts like a Ruby method. If you don't specify the right number of arguments, you can't call the block. But a block created with Proc.new acts like the anonymous code block you pass into a method like Enumerable#each:

```
add_procnew = Proc.new { |x,y| x + y }

add_procnew.call(4)
# TypeError: nil can't be coerced into Fixnum

add_procnew.call(4,5,6)                                 # => 9
```

If you don't specify enough arguments when you call the block, the rest of the arguments are given nil. If you specify too many arguments, the extra arguments are ignored. Unless you want this kind of behavior, use lambda.

In Ruby 1.8, Kernel#proc acts like Kernel#lambda. In Ruby 1.9, Kernel#proc acts like Proc.new, as better befits its name.

See Also

- Recipe 7.2, "Writing a Method That Accepts a Block"
- Recipe 10.4, "Getting a Reference to a Method"

7.2 Writing a Method That Accepts a Block

Problem

You want to write a method that can accept and call an attached code block: a method that works like Array#each, Fixnum#upto, and other built-in Ruby methods.

Solution

You don't need to do anything special to make your method capable of accepting a block. Any method can use a block if the caller passes one in. At any time in your method, you can call the block with `yield`:

```
def call_twice
  puts "I'm about to call your block."
  yield
  puts "I'm about to call your block again."
  yield
end

call_twice { puts "Hi, I'm a talking code block." }
# I'm about to call your block.
# Hi, I'm a talking code block.
# I'm about to call your block again.
# Hi, I'm a talking code block.
```

Another example:

```
def repeat(n)
 if block_given?
   n.times { yield }
  else
    raise ArgumentError.new("I can't repeat a block you don't give me!")
  end
end

repeat(4) { puts "Hello." }
# Hello.
# Hello.
# Hello.
# Hello.

repeat(4)
# ArgumentError: I can't repeat a block you don't give me!
```

Discussion

Since Ruby focuses so heavily on iterator methods and other methods that accept code blocks, it's important to know how to use code blocks in your own methods.

You don't have to do anything special to make your method capable of taking a code block. A caller can pass a code block into *any* Ruby method; it's just that there's no point in doing that if the method never invokes `yield`.

```
puts("Print this message.") { puts "And also run this code block!" }
# Print this message.
```

The `yield` keyword acts like a special method, a stand-in for whatever code block was passed in. When you call it, it's exactly as the code block were a `Proc` object and you had invoked its `call` method.

This may seem mysterious if you're unfamiliar with the practice of passing blocks around, but it is usually the preferred method of calling blocks in Ruby. If you feel more comfortable receiving a code block as a "real" argument to your method, see Recipe 7.3.

You can pass in arguments to `yield` (they'll be passed to the block) and you can do things with the value of the `yield` statement (this is the value of the last statement in the block).

Here's a method that passes arguments into its code block, and uses the value of the block:

```
def call_twice
  puts "Calling your block."
  ret1 = yield("very first")
  puts "The value of your block: #{ret1}"

  puts "Calling your block again."
  ret2 = yield("second")
  puts "The value of your block: #{ret2}"
end

call_twice do |which_time|
  puts "I'm a code block, called for the #{which_time} time."
  which_time == "very first" ? 1 : 2
end
# Calling your block.
# I'm a code block, called for the very first time.
# The value of your block: 1
# Calling your block again.
# I'm a code block, called for the second time.
# The value of your block: 2
```

Here's a more realistic example. The method `Hash#find` takes a code block, passes each of a hash's key-value pairs into the code block, and returns the first key-value pair for which the code block evaluates to true.

```
squares = {0=>0, 1=>1, 2=>4, 3=>9}
squares.find { |key, value| key > 1 }          # => [2, 4]
```

Suppose we want a method that works like `Hash#find`, but returns a new hash containing *all* the key-value pairs for which the code block evaluates to true. We can do this by passing arguments into the `yield` statement and using its result:

```
class Hash
  def find_all
    new_hash = Hash.new
    each { |k,v| new_hash[k] = v if yield(k, v) }
    new_hash
  end
end

squares.find_all { |key, value| key > 1 }        # => {2=>4, 3=>9}
```

As it turns out, the Hash#delete_if method already does the inverse of what we want. By negating the result of our code block, we can make Hash#delete_if do the job of Hash#find_all. We just need to work off of a duplicate of our hash, because delete_if is a destructive method:

```
squares.dup.delete_if { |key, value| key > 1 }   # => {0=>0, 1=>1}
squares.dup.delete_if { |key, value| key <= 1 }  # => {2=>4, 3=>9}
```

Hash#find_all turns out to be unnecessary, but it made for a good example.

You can write a method that takes an *optional* code block by calling Kernel#block_given? from within your method. That method returns true only if the caller of your method passed in a code block. If it returns false, you can raise an exception, or you can fall back to behavior that doesn't need a block and never uses the yield keyword.

If your method calls yield and the caller didn't pass in a code block, Ruby will throw an exception:

```
[1, 2, 3].each
# LocalJumpError: no block given
```

See Also

- Recipe 7.3, "Binding a Block Argument to a Variable"

7.3 Binding a Block Argument to a Variable

Problem

You've written a method that takes a code block, but it's not enough for you to simply call the block with yield. You need to somehow bind the code block to a variable, so you can manipulate the block directly. Most likely, you need to pass it as the code block to another method.

Solution

Put the name of the block variable at the end of the list of your method's arguments. Prefix it with an ampersand so that Ruby knows it's a block argument, not a regular argument.

An incoming code block will be converted into a Proc object and bound to the block variable. You can pass it around to other methods, call it directly using call, or yield to it as though you'd never bound it to a variable at all. All three of the following methods do exactly the same thing:

```
def repeat(n)
  n.times { yield } if block_given?
end
repeat(2) { puts "Hello." }
# Hello.
# Hello.
```

```
def repeat(n, &block)
  n.times { block.call } if block
end
repeat(2) { puts "Hello." }
# Hello.
# Hello.

def repeat(n, &block)
  n.times { yield } if block
end
repeat(2) { puts "Hello." }
# Hello.
# Hello.
```

Discussion

If `&foo` is the name of a method's last argument, it means that the method accepts an optional block named `foo`. If the caller chooses to pass in a block, it will be made available as a `Proc` object bound to the variable `foo`. Since it is an optional argument, `foo` will be `nil` if no block is actually passed in. This frees you from having to call `Kernel#block_given?` to see whether or not you got a block.

When you call a method, you can pass in any `Proc` object as the code block by prefixing the appropriate variable name with an ampersand. You can even do this on a `Proc` object that was originally passed in as a code block to your method.

Many methods for collections, like `each`, `select`, and `detect`, accept code blocks. It's easy to wrap such methods when your own methods can bind a block to a variable. Here, a method called `biggest` finds the largest element of a collection that gives a true result for the given block:

```
def biggest(collection, &block)
  block ? collection.select(&block).max : collection.max
end

array = [1, 2, 3, 4, 5]
biggest(array) {|i| i < 3}                  # => 2
biggest(array) {|i| i != 5 }                # => 4
biggest(array)                              # => 5
```

This is also very useful when you need to write a frontend to a method that takes a block. Your wrapper method can bind an incoming code block to a variable, then pass it as a code block to the other method.

This code calls a code block `limit` times, each time passing in a random number between `min` and `max`:

```
def pick_random_numbers(min, max, limit)
  limit.times { yield min+rand(max+1) }
end
```

This code is a wrapper method for `pick_random_numbers`. It calls a code block 6 times, each time with a random number from 1 to 49:

```
def lottery_style_numbers(&block)
  pick_random_numbers(1, 49, 6, &block)
end

lottery_style_numbers { |n| puts "Lucky number: #{n}" }
# Lucky number: 20
# Lucky number: 39
# Lucky number: 41
# Lucky number: 10
# Lucky number: 41
# Lucky number: 32
```

The code block argument must always be the very last argument defined for a method. This means that if your method takes a variable number of arguments, the code block argument goes *after* the container for the variable arguments:

```
def invoke_on_each(*args, &block)
  args.each { |arg| yield arg }
end

invoke_on_each(1, 2, 3, 4) { |x| puts x ** 2 }
# 1
# 4
# 9
# 16
```

See Also

- Recipe 8.11, "Accepting or Passing a Variable Number of Arguments"
- Recall from the chapter introduction that in Ruby 1.8, a code block cannot itself take a block argument; this is fixed in Ruby 1.9

7.4 Blocks as Closures: Using Outside Variables Within a Code Block

Problem

You want to share variables between a method, and a code block defined within it.

Solution

Just reference the variables, and Ruby will do the right thing. Here's a method that adds a certain number to every element of an array:

```
def add_to_all(array, number)
  array.collect { |x| x + number }
end

add_to_all([1, 2, 3], 10)               # => [11, 12, 13]
```

`Enumerable#collect` can't access `number` directly, but it's passed a block that can access it, since `number` was in scope when the block was defined.

Discussion

A Ruby block is a *closure*: it carries around the context in which it was defined. This is useful because it lets you define a block as though it were part of your normal code, then tear it off and send it to a predefined piece of code for processing.

A Ruby block contains references to the variable bindings, not copies of the values. If the variable changes later, the block will have access to the new value:

```
tax_percent = 6
position = lambda do
  "I have always supported a #{tax_percent}% tax on imported limes."
end
position.call
# => "I have always supported a 6% tax on imported limes."

tax_percent = 7.25
position.call
# => "I have always supported a 7.25% tax on imported limes."
```

This works both ways: you can rebind or modify a variable from within a block.

```
counter = 0
4.times { counter += 1; puts "Counter now #{counter}"}
# Counter now 1
# Counter now 2
# Counter now 3
# Counter now 4
counter                                                 # => 4
```

This is especially useful when you want to simulate `inject` or `collect` in conjunction with a strange iterator. You can create a storage object outside the block, and add things to it from within the block. This code simulates `Enumerable#collect`, but it collects the elements of an array in reverse order:

```
accumulator = []
[1, 2, 3].reverse_each { |x| accumulator << x + 1 }

accumulator                 # => [4, 3, 2]
```

The `accumulator` variable is not within the scope of `Array#reverse_each`, but it is within the scope of the block.

7.5 Writing an Iterator Over a Data Structure

Problem

You've created a custom data structure, and you want to implement an `each` method for it, or you want to implement an unusual way of iterating over an existing data structure.

Solution

Complex data structures are usually constructed out of the basic data structures: hashes, arrays, and so on. All of the basic data structures have defined the each method. If your data structure is composed entirely of scalar values and these simple data structures, you can write a new each method in terms of the each methods of its components.

Here's a simple tree data structure. A tree contains a single value, and a list of children (each of which is a smaller tree).

```
class Tree
  attr_reader :value
  def initialize(value)
    @value = value
    @children = []
  end

  def <<(value)
    subtree = Tree.new(value)
    @children << subtree
    return subtree
  end
end
```

Here's code to create a specific Tree (Figure 7-1):

```
t = Tree.new("Parent")
child1 = t << "Child 1"
child1 << "Grandchild 1.1"
child1 << "Grandchild 1.2"
child2 = t << "Child 2"
child2 << "Grandchild 2.1"
```

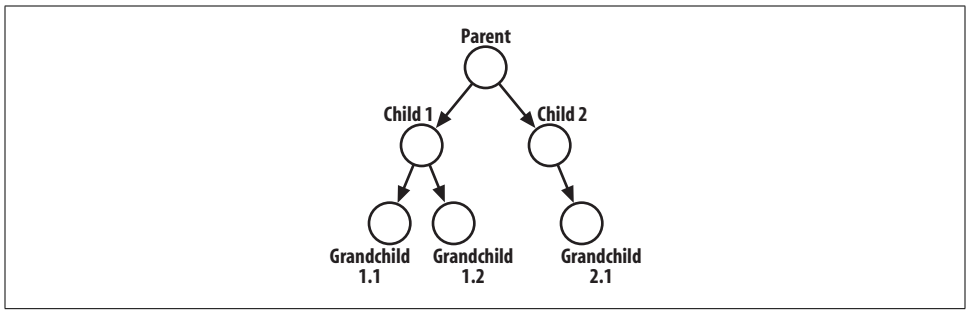

Figure 7-1. A simple tree

How can we iterate over this data structure? Since a tree is defined recursively, it makes sense to iterate over it recursively. This implementation of Tree#each yields the value stored in the tree, then iterates over its children (the children are stored in an array, which already supports each) and recursively calls Tree#each on every child tree.

```
class Tree
  def each
    yield value
    @children.each do |child_node|
      child_node.each { |e| yield e }
    end
  end
end
```

The each method traverses the tree in a way that looks right:

```
t.each { |x| puts x }
# Parent
# Child 1
# Grandchild 1.1
# Grandchild 1.2
# Child 2
# Grandchild 2.1
```

Discussion

The simplest way to build an iterator is recursively: to use smaller iterators until you've covered every element in your data structure. But what if those iterators aren't there? More likely, what if they're there but they give you elements in the wrong order? You'll need to go down a level and write some loops.

Loops are somewhat declassé in Ruby because iterators are more idiomatic, but when you're *writing* an iterator you may have no choice but to use a loop. Here's a reprint of an iterator from Recipe 4.1, which illustrates how to use a `while` loop to iterate over an array from both sides:

```
class Array
 def each_from_both_sides( )
    front_index = 0
    back_index = self.length-1
    while front_index <= back_index
      yield self[front_index]
      front_index += 1
      if front_index <= back_index
        yield self[back_index]
        back_index -= 1
      end
    end
  end
end

%w{Curses! been again! foiled I've}.each_from_both_sides { |x| puts x }
# Curses!
# I've
# been
# foiled
# again!
```

Here are two more simple iterators. The first one yields each element multiple times in a row:

```
module Enumerable
  def each_n_times(n)
    each { |e| n.times { yield e } }
  end
end

%w{Hello Echo}.each_n_times(3) { |x| puts x }
# Hello
# Hello
# Hello
# Echo
# Echo
# Echo
```

The next one returns the elements of an `Enumerable` in random order; see Recipe 4.10 for a more efficient way to do the shuffling.

```
module Enumerable
  def each_randomly
    (sort_by { rand }).each { |e| yield e }
  end
end
%w{Eat at Joe's}.each_randomly { |x| puts x }
# Eat
# Joe's
# at
```

See Also

- Recipe 4.1, "Iterating Over an Array"
- Recipe 4.10, "Shuffling an Array"
- Recipe 5.7, "Iterating Over a Hash"
- Recipe 7.6, "Changing the Way an Object Iterates"
- Recipe 7.8, "Stopping an Iteration"
- Recipe 7.9, "Looping Through Multiple Iterables in Parallel"

7.6 Changing the Way an Object Iterates

Problem

You want to use a data structure as an `Enumerable`, but the object's implementation of `#each` doesn't iterate the way you want. Since all of `Enumerable`'s methods are based on `each`, this makes them all useless to you.

Discussion

Here's a concrete example: a simple array.

```
array = %w{bob loves alice}
array.collect { |x| x.capitalize }
# => ["Bob", "Loves", "Alice"]
```

Suppose we want to call `collect` on this array, but we don't want `collect` to use `each`: we want it to use `reverse_each`. Something like this hypothetical `collect_reverse` method:

```
array.collect_reverse { |x| x.capitalize }
# => ["Alice", "Loves", "Bob"]
```

Actually defining a `collect_reverse` method would add significant new code and only solve part of the problem. We could overwrite the array's `each` implementation with a singleton method that calls `reverse_each`, but that's hacky and it would surely have undesired side effects.

Fortunately, there's an elegant solution with no side effects: wrap the object in an `Enumerator`. This gives you a new object that acts like the old object would if you'd swapped out its `each` method:

```
require 'enumerator'
reversed_array = array.to_enum(:reverse_each)
reversed_array.collect { |x| x.capitalize }
# => ["Alice", "Loves", "Bob"]

reversed_array.each_with_index do |x, i|
  puts %{#{i}=>"#{x}"}
end
# 0=>"alice"
# 1=>"loves"
# 2=>"bob"
```

Note that you can't use the `Enumerator` for our array as though it were the actual array. Only the methods of `Enumerable` are supported:

```
reversed_array[0]
# NoMethodError: undefined method `[]' for #<Enumerable::Enumerator:0xb7c2cc8c>
```

Discussion

Whenever you're tempted to reimplement one of the methods of `Enumerable`, try using an `Enumerator` instead. It's like modifying an object's `each` method, but it doesn't affect the original object.

This can save you a lot of work. Suppose you have a tree data structure that provides three different iteration styles: `each_prefix`, `each_postfix`, and `each_infix`. Rather than implementing the methods of `Enumerable` for all three iteration styles, you can let `each_prefix` be the default implementation of `each`, and call `tree.to_enum(:each_postfix)` or `tree.to_enum(:each_infix)` if you need an `Enumerable` that acts differently.

A single underlying object can have multiple Enumerable objects. Here's a second Enumerable for our simple array, in which each acts like each_with_index does for the original array:

```
array_with_index = array.enum_with_index
array_with_index.each do |x, i|
  puts %{#{i}=>"#{x}"}
end
# 0=>"bob"
# 1=>"loves"
# 2=>"alice"

array_with_index.each_with_index do |x, i|
  puts %{#{i}=>#{x.inspect}}
end
# 0=>["bob", 0]
# 1=>["loves", 1]
# 2=>["alice", 2]
```

When you require 'enumerator', Enumerable sprouts two extra enumeration methods, each_cons and each_slice. These make it easy to iterate over a data structure in chunks. An example is the best way to show what they do:

```
sentence = %w{Well, now I've seen everything!}

two_word_window = sentence.to_enum(:each_cons, 2)
two_word_window.each { |x| puts x.inspect }
# ["Well,", "now"]
# ["now", "I've"]
# ["I've", "seen"]
# ["seen", "everything!"]

two_words_at_a_time = sentence.to_enum(:each_slice, 2)
two_words_at_a_time.each { |x| puts x.inspect }
# ["Well,", "now"]
# ["I've", "seen"]
# ["everything!"]
```

Note how any arguments passed into to_enum are passed along as arguments to the iteration method itself.

In Ruby 1.9, the Enumerable::Enumerator class is part of the Ruby core; you don't need the require statement. Also, each_cons and each_slice are built-in methods of Enumerable.

See Also

- Recipe 7.9, "Looping Through Multiple Iterables in Parallel"
- Recipe 20.6, "Running a Code Block on Many Objects Simultaneously"

7.7 Writing Block Methods That Classify or Collect

Problem

The basic block methods that come with the Ruby standard library aren't enough for you. You want to define your own method that classifies the elements in an enumeration (like `Enumerable#detect` and `Enumerable#find_all`), or that does a transformation on each element in an enumeration (like `Enumerable#collect`).

Solution

You can usually use `inject` to write a method that searches or classifies an enumeration of objects. With `inject` you can write your own versions of methods such as `detect` and `find_all`:

```
module Enumerable
  def find_no_more_than(limit)
    inject([]) do |a,e|
      a << e if yield e
      return a if a.size >= limit
      a
    end
  end
end
```

This code finds at most three of the even numbers in a list:

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
a.find_no_more_than(3) { |x| x % 2 == 0 }            # => [2, 4, 6]
```

If you find yourself needing to write a method like `collect`, it's probably because, for your purposes, `collect` itself yields elements in the wrong order. You can't use `inject`, because that yields elements in the same order as `collect`.

You need to find or write an iterator that yields elements in the order you want. Once you've done that, you have two options: you can write a `collect` equivalent on top of the iterator method, or you can use the iterator method to build an `Enumerator` object, and call its `collect` method (as seen in Recipe 7.6).

Discussion

We discussed these block methods in more detail in Chapter 4, because arrays are the simplest and most common enumerable data type. But almost any data structure can be enumerated, and a more complex data structure can be enumerated in more different ways.

As you'll see in Recipe 9.4, the `Enumerable` methods, like `detect` and `inject`, are actually implemented in terms of `each`. The `detect` and `inject` methods yield to the code block every element that comes out of `each`. The value of the `yield` statement is used to determine whether the element matches some criteria.

In a method like `detect`, the iteration may stop once it finds an element that matches. In a method like `find_all`, the iteration goes through all elements, collecting the ones that match.

Methods like `collect` work the same way, but instead of returning a subset of elements based on what the code block says, they collect the values returned by the code block in a new data structure, and return the data structure once the iteration is completed.

If you're using a particular object and you wish its `collect` method used a different iterator, then you should turn the object into an `Enumerator` and call its `collect` method. But if you're writing a class and you want to expose a new `collect`-like method, you'll have to define a new method.* In that case, the best solution is probably to expose a method that returns a custom `Enumerator`: that way, your users can use all the methods of `Enumerable`, not just `collect`.

See Also

- Recipe 4.5, "Sorting an Array"
- Recipe 4.11, "Getting the N Smallest Items of an Array"
- Recipe 4.15, "Partitioning or Classifying a Set"
- Recipe 7.6, "Changing the Way an Object Iterates"
- If all you want is to make your custom data structure support the methods of `Enumerable`, see Recipe 9.4, "Implementing Enumerable: Write One Method, Get 22 Free"

7.8 Stopping an Iteration

Problem

You want to interrupt an iteration from within the code block you passed into it.

Solution

The simplest way to interrupt execution is to use `break`. A `break` statement will jump out of the closest enclosing loop defined in the current method:

```
1.upto(10) do |x|
  puts x
  break if x == 3
end
# 1
# 2
# 3
```

* Of course, behind the scenes, your method could just create an appropriate `Enumerator` and call its `collect` implemenation.

Discussion

The `break` statement is simple but it has several limitations. You can't use `break` within a code block defined with `Proc.new` or (in Ruby 1.9 and up) `Kernel#proc`. If this is a problem for you, use `lambda` instead:

```
aBlock = Proc.new do |x|
  puts x
  break if x == 3
  puts x + 2
end

aBlock.call(5)
# 5
# 7

aBlock.call(3)
# 3
# LocalJumpError: break from proc-closure
```

More seriously, you can't use `break` to jump out of multiple loops at once. Once a loop has run, there's no way to know whether it completed normally or by using `break`.

The simplest way around this problem is to enclose the code you want to skip within a `catch` block with a descriptive symbolic name. You can then `throw` the corresponding symbol when you want to jump to the end of the `catch` block. This lets you skip out of any number of nested loops and method calls.

The `throw/catch` syntax isn't exception handling—exceptions use a `raise/rescue` syntax. This is a special flow control construct designed to replace the use of exceptions for flow control (as sometimes happens in Java programs). It's a bit like an old-style global GOTO, capable of suddenly moving execution to a faraway part of your program. It keeps your code more readable than a GOTO, though, because it's restricted: a `throw` can only jump to the end of a corresponding `catch` block.

The best example of the `catch`..`throw` syntax is the `Find.find` function described in Recipe 6.12. When you pass a code block into `Find.find`, it yields up every directory and file in a certain directory tree. When your code block is given a directory, it can stop `find` from recursing into that directory by calling `Find.prune`, which throws a `:prune` symbol. Using `break` would stop the `find` operation altogether; throwing a symbol lets `Find.prune` know to just skip one directory.

Here's a simplified view of the `Find.find` and `Find.prune` code:

```
def find(*paths)
  paths.each do |p|
    catch(:prune) do
      # Process p as a file or directory...
    end
    # When you call Find.prune you'll end up here.
```

```
  end
end

def prune
  throw :prune
end
```

When you call `Find.prune`, execution jumps to immediately after the `catch(:prune)` block. `Find.find` then starts processing the next file or directory.

See Also

- Recipe 6.12, "Walking a Directory Tree"
- `ri Find`

7.9 Looping Through Multiple Iterables in Parallel

Problem

You want to traverse multiple iteration methods simultaneously, probably to match up the corresponding elements in several different arrays.

Solution

The `SyncEnumerator` class, defined in the `generator` library, makes it easy to iterate over a bunch of arrays or other `Enumerable` objects in parallel. Its `each` method yields a series of arrays, each array containing one item from each underlying `Enumerable` object:

```
require 'generator'

enumerator = SyncEnumerator.new(%w{Four seven}, %w{score years},
                                %w{and ago})
enumerator.each do |row|
  row.each { |word| puts word }
  puts '---'
end
# Four
# score
# and
# ---
# seven
# years
# ago
# ---

enumerator = SyncEnumerator.new(%w{Four and}, %w{score seven years ago})
enumerator.each do |row|
  row.each { |word| puts word }
  puts '---'
end
# Four
```

```
# score
# ---
# and
# seven
# ---
# nil
# years
# ---
# nil
# ago
# ---
```

You can reproduce the workings of a `SyncEnumerator` by wrapping each of your `Enumerable` objects in a `Generator` object. This code acts like `SyncEnumerator#each`, only it yields each individual item instead of arrays containing one item from each `Enumerable`:

```
def interosculate(*enumerables)
  generators = enumerables.collect { |x| Generator.new(x) }
  done = false
  until done
    done = true
    generators.each do |g|
      if g.next?
        yield g.next
        done = false
      end
    end
  end
end

interosculate(%w{Four and}, %w{score seven years ago}) do |x|
  puts x
end
# Four
# score
# and
# seven
# years
# ago
```

Discussion

Any object that implements the `each` method can be wrapped in a `Generator` object. If you've used Java, think of a `Generator` as being like a Java `Iterator` object. It keeps track of where you are in a particular iteration over a data structure.

Normally, when you pass a block into an iterator method like `each`, that block gets called for every element in the iterator without interruption. No code outside the block will run until the iterator is done iterating. You can stop the iteration by writing a `break` statement inside the code block, but you can't restart a broken iteration later from the same place—unless you use a `Generator`.

Think of an iterator method like each as a candy dispenser that pours out all its candy in a steady stream once you push the button. The Generator class lets you turn that candy dispenser into one which dispenses only one piece of candy every time you push its button. You can carry this new dispenser around and ration your candy more easily.

In Ruby 1.8, the Generator class uses continuations to achieve this trick. It sets bookmarks for jumping out of an iteration and then back in. When you call Generator#next the generator "pumps" the iterator once (yielding a single element), sets a bookmark, and returns control back to your code. The next time you call Generator#next, the generator jumps back to its previously set bookmark and "pumps" the iterator once more.

Ruby 1.9 uses a more efficient implementation based on threads. This implementation calls each Enumerable object's each method (triggering the neverending stream of candy), but it does it in a separate thread for each object. After each piece of candy comes out, Ruby freezes time (pauses the thread) until the next time you call Generator#next.

It's simple to wrap an array in a generator, but if that's all there were to generators, you wouldn't need to mess around with Generators or even SyncEnumerables. It's easy to simulate the behavior of SyncEnumerable for arrays by starting an index into each array and incrementing it whenever you want to get another item from a particular array. Generator methods are truly useful in their ability to turn *any* type of iteration into a single-item candy dispenser.

Suppose that you want to use the functionality of a generator to iterate over an array, but you have an unusual type of iteration in mind. For instance, consider an array that looks like this:

```
l = ["junk1", 1, "junk2", 2, "junk3", "junk4", 3, "junk5"]
```

Let's say you'd like to iterate over the list but skip the "junk" entries. Wrapping the list in a generator object doesn't work; it gives you all the entries:

```
g = Generator.new(l)
g.next                                  # => "junk1"
g.next                                  # => 1
g.next                                  # => "junk2"
```

It's not difficult to write an iterator method that skips the junk. Now, we don't want an iterator method—we want a Generator object—but the iterator method is a good starting point. At least it proves that the iteration we want can be implemented in Ruby.

```
def l.my_iterator
  each { |e| yield e unless e =~ /^junk/ }
end

l.my_iterator { |x| puts x }
```

```
# 1
# 2
# 3
```

Here's the twist: when you wrap an array in a `Generator` or a `SyncEnumerable` object, you're actually wrapping the array's `each` method. The `Generator` doesn't just happen to yield elements in the same order as `each`: it's actually calling `each`, but using continuation (or thread) trickery to pause the iteration after each call to `Generator#next`.

By defining an appropriate code block and passing it into the `Generator` constructor, you can make a generation object of out of any piece of iteration code—not only the `each` method. The generator will know to call and interrupt that block of code, just as it knows to call and interrupt `each` when you pass an array into the constructor. Here's a generator that iterates over our array the way we want:

```
g = Generator.new { |g| l.each { |e| g.yield e unless e =~ /^junk/ } }
g.next                                          # => 1
g.next                                          # => 2
g.next                                          # => 3
```

The `Generator` constructor can take a code block that accepts the generator object itself as an argument. This code block performs the iteration that you'd like to have wrapped in a generator. Note the basic similarity of the code block to the body of the `l#my_iterator` method. The only difference is that instead of the `yield` keyword we call the `Generator#yield` function, which handles some of the work involved with setting up and jumping to the continuations (`Generator#next` handles the rest of the continuation work).

Once you see how this works, you can eliminate some duplicate code by wrapping the `l#my_iterator` method itself in a `Generator`:

```
g = Generator.new { |g| l.my_iterator { |e| g.yield e } }
g.next                                          # => 1
g.next                                          # => 2
g.next                                          # => 3
```

Here's a version of the `interosculate` method that can wrap methods as well as arrays. It accepts any combination of `Enumerable` objects and `Method` objects, turns each one into a `Generator` object, and loops through all the `Generator` objects, getting one element at a time from each:

```
def interosculate(*iteratables)
  generators = iteratables.collect do |x|
    if x.is_a? Method
      Generator.new { |g| x.call { |e| g.yield e } }
    else
      Generator.new(x)
    end
  end
  done = false
  until done
```

```
    done = true
    generators.each do |g|
      if g.next?
        yield g.next
        done = false
      end
    end
  end
end
```

Here, we pass `interosculate` an array and a `Method` object, so that we can iterate through two arrays in opposite directions:

```
words1 = %w{Four and years}
words2 = %w{ago seven score}
interosculate(words1, words2.method(:reverse_each)) { |x| puts x }
# Four
# score
# and
# seven
# years
# ago
```

See Also

- Recipe 7.5, "Writing an Iterator Over a Data Structure"
- Recipe 7.6, "Changing the Way an Object Iterates"

7.10 Hiding Setup and Cleanup in a Block Method

Problem

You have a setup method that always needs to run before custom code, or a cleanup method that needs to run afterwards. You don't trust the person writing the code (possibly yourself) to remember to call the setup and cleanup methods.

Solution

Create a method that runs the setup code, yields to a code block (which contains the custom code), then runs the cleanup code. To make sure the cleanup code always runs, even if the custom code throws an exception, use a `begin`/`finally` block.

```
def between_setup_and_cleanup
  setup
  begin
   yield
  finally
    cleanup
  end
end
```

Here's a concrete example. It adds a DOCTYPE and an HTML tag to the beginning of an HTML document. At the end, it closes the HTML tag it opened earlier. This saves you a little bit of work when you're generating HTML files.

```
def write_html(out, doctype=nil)
  doctype ||= %{<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
           "http://www.w3.org/TR/html4/loose.dtd">}
  out.puts doctype
  out.puts '<html>'
  begin
    yield out
  ensure
    out.puts '</html>'
  end
end

write_html($stdout) do |out|
  out.puts '<h1>Sorry, the Web is closed.</h1>'
end
# <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
#     "http://www.w3.org/TR/html4/loose.dtd">
# <html>
# <h1>Sorry, the Web is closed.</h1>
# </html>
```

Discussion

This useful technique shows up most often when there are scarce resources (such as file handles or database connections) that *must* be closed when you're done with them, lest they all get used up. A language that makes the programmer remember these resources tends to leak those resources, because programmers are lazy. Ruby makes it easy to be lazy and still do the right thing.

You've probably used this technique already, with the the `Kernel#open` and `File#open` methods for opening files on disk. These methods accept a code block that manipulates an already open file. They open the file, call your code block, and close the file once you're done:

```
open('output.txt', 'w') do |out|
  out.puts 'Sorry, the filesystem is also closed.'
end
```

Ruby's standard `cgi` module takes the `write_html` example to its logical conclusion.* You can construct an entire HTML document by nesting blocks inside each other. Here's a small Ruby CGI that outputs much the same document as the `write_html` example above.

```
#!/usr/bin/ruby

# closed_cgi.rb
```

* But your code will be more maintainable if you do HTML with templates instead of writing it in Ruby code.

```
require 'cgi'
c = CGI.new("html4")

c.out do
   c.html do
      c.h1 { 'Sorry, the Web is closed.' }
  end
end
```

Note the multiple levels of blocks: the block passed into `CGI#out` simply calls `CGI#html` to generate the DOCTYPE and the `<html>` tags. The `<html>` tags contain the result of a call to `CGI#h1`, which encloses some plain text in `<h1>` tags. The program produces this output:

```
Content-Type: text/html
Content-Length: 137

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<HTML><H1>Sorry, the Web is closed.</H1></HTML>
```

The `XmlMarkup` class in Ruby's `builder` gem works the same way: you can write Ruby code that resembles the structure of the document it creates:

```
require 'rubygems'
require 'builder'
xml = Builder::XmlMarkup.new.message('type' => 'apology') do |b|
  b.content('Sorry, Web Services are closed.')
end
puts xml
# <message type="apology">
#  <content>Sorry, Web Services are closed.</content>
# </message>
```

See Also

- Recipe 6.13, "Locking a File," uses this technique to create a method that locks a file, and automatically unlocks it when you're done using it
- Recipe 11.9, "Creating and Modifying XML Documents"
- Recipe 20.11, "Avoiding Deadlock," uses this technique to have your thread lock multiple resources in the right order, and unlock them when you're done using them

7.11 Coupling Systems Loosely with Callbacks

Problem

You want to combine different types of objects without hardcoding them full of references to each other.

Solution

Use a callback system, in which objects register code blocks with each other to be executed as needed. An object can call out to its registered callbacks when it needs something, or it can send notification to the callbacks when it does something.

To implement a callback system, write a "register" or "subscribe" method that accepts a code block. Store the registered code blocks as `Proc` objects in a data structure: probably an array (if you only have one type of callback) or a hash (if you have multiple types). When you need to call the callbacks, iterate over the data structure and `call` each of the registered code blocks.

Here's a mixin module that gives each instance of a class its own hash of "listener" callback blocks. An outside object can listen for a particular event by calling `subscribe` with the name of the event and a code block. The dispatcher itself is responsible for calling `notify` with an appropriate event name at the appropriate time, and the outside object is responsible for passing in the name of the event it wants to "listen" for.

```
module EventDispatcher
  def setup_listeners
    @event_dispatcher_listeners = {}
  end

  def subscribe(event, &callback)
    (@event_dispatcher_listeners[event] ||= []) << callback
  end

  protected
  def notify(event, *args)
    if @event_dispatcher_listeners[event]
      @event_dispatcher_listeners[event].each do |m|
        m.call(*args) if m.respond_to? :call
      end
    end
    return nil
  end
end
```

Here's a `Factory` class that keeps a set of listeners. An outside object can choose to be notified every time a `Factory` object is created, or every time a `Factory` object produces a widget:

```
class Factory
  include EventDispatcher

  def initialize
    setup_listeners
  end

  def produce_widget(color)
    #Widget creation code goes here...
```

```
    notify(:new_widget, color)
  end
end
```

Here's a listener class that's interested in what happens with `Factory` objects:

```
class WidgetCounter
  def initialize(factory)
    @counts = Hash.new(0)
    factory.subscribe(:new_widget) do |color|
      @counts[color] += 1
      puts "#{@counts[color]} #{color} widget(s) created since I started watching."
    end
  end
end
```

Finally, here's the listener in action:

```
f1 = Factory.new
WidgetCounter.new(f1)
f1.produce_widget("red")
# 1 red widget(s) created since I started watching.

f1.produce_widget("green")
# 1 green widget(s) created since I started watching.

f1.produce_widget("red")
# 2 red widget(s) created since I started watching.

# This won't produce any output, since our listener is listening to
# another Factory.
Factory.new.produce_widget("blue")
```

Discussion

Callbacks are an essential technique for making your code extensible. This technique has many names (callbacks, hook methods, plugins, publish/subscribe, etc.) but no matter what terminology is used, it's always the same. One object asks another to call a piece of code (the callback) when some condition is met. This technique works even when the two objects know almost nothing about each other. This makes it ideal for refactoring big, tightly integrated systems into smaller, loosely coupled systems.

In a pure listener system (like the one given in the Solution), the callbacks set up lines of communication that always move from the event dispatcher to the listeners. This is useful when you have a master object (like the `Factory`), from which numerous lackey objects (like the `WidgetCounter`) take all their cues.

But in many loosely coupled systems, information moves both ways: the dispatcher calls the callbacks and then uses the return results. Consider the stereotypical web portal: a customizable homepage full of HTML boxes containing sports scores, weather predictions, and so on. Since new boxes are always being added to the

system, the core portal software shouldn't have to know anything about a specific box. The boxes should also know as little about the core software as possible, so that changing the core doesn't require a change to all the boxes.

A simple change to the `EventDispatcher` class makes it possible for the dispatcher to use the return values of the registered callbacks. The original implementation of `EventDispatcher#notify` called the registered code blocks, but ignored their return value. This version of `EventDispatcher#notify` yields the return values to a block passed in to `notify`:

```
module EventDispatcher
  def notify(event, *args)
      if @event_dispatcher_listeners[event]
      @event_dispatcher_listeners[event].each do |m|
        yield(m.call(*args)) if m.respond_to? :call
      end
    end
    return nil
  end
end
```

Here's an insultingly simple portal rendering engine. It lets boxes register to be rendered inside an HTML table, on one of two rows on the portal page:

```
class Portal
  include EventDispatcher

  def initialize
    setup_listeners
  end

  def render
    puts '<table>'
    render_block = Proc.new { |box| puts "  <td>#{box}</td>" }
    [:row1, :row2].each do |row|
      puts ' <tr>'
      notify(row, &render_block)
      puts ' </tr>'
    end
  puts '</table>'
  end
end
```

Here's the rendering engine rendering a specific user's portal layout. This user likes to see a stock ticker and a weather report on the left, and a news box on the right. Note that there aren't even any classes for these boxes; they're so simple they can be implemented as anonymous code blocks:

```
portal = Portal.new
portal.subscribe(:row1) { 'Stock Ticker' }
portal.subscribe(:row1) { 'Weather' }
portal.subscribe(:row2) { 'Pointless, Trivial News' }
```

```
portal.render
# <table>
#  <tr>
#   <td>Stock Ticker</td>
#   <td>Weather</td>
#  </tr>
#  <tr>
#   <td>Pointless, Trivial News</td>
#  </tr>
# </table>
```

If you want the registered listeners to be shared across all instances of a class, you can make `listeners` a class variable, and make `subscribe` a module method. This is most useful when you want listeners to be notified whenever a new instance of the class is created.