CHAPTER 19

Save Blocks to Execute Later

It's hard to believe, but we are not done with code blocks yet. There is still one more bit of software goodness that we can squeeze out of the programming construct that keeps on giving. Two chapters ago we called on code blocks as iterators to sequence through collections. In the last chapter we looked to the block as a mechanism for delivering the right code to the right context. The topic of this chapter is similar to the last: using blocks as a device for delivering your code where it is needed. The difference is that, while the execute around technique was all about getting your code to the right place, this chapter will focus on using blocks to transport your code through time. In the pages that follow, we are going to learn how you can grab hold of the code block that is passed into your method and simply hang on to it until you need it.

Explicit Blocks

So far in our adventures with code blocks we have used `block_supplied?` to determine whether someone has passed in a code block to a method and `yield` to fire off the block. As we have seen, `block_supplied?` and `yield` rely on the fact that Ruby treats a code block appended to the end of a method call as a sort of implicit parameter to the call, a parameter that only `yield` and `block_supplied?` know how to get at.

However, implicitly is not the only way to pass blocks to your methods. If you add a parameter prefixed with an ampersand to the end of your parameter list,[1] Ruby will turn any block passed into the method into a garden-variety parameter. After you have captured a block with an explicit parameter, you can run it by calling its `call` method. Here, for example, is a very simple method with an explicit code block parameter:

```
def run_that_block( &that_block )
  puts "About to run the block"
  that_block.call
  puts "Done running the block"
end
```

It's also trivially easy to figure out whether the caller actually did pass in a block: Just check to see if the value of the block parameter is `nil`:

```
  that_block.call if that_block
```

Explicit code blocks are easy and clear enough that some Ruby programmers (including me!) habitually use them rather than the implicit variety. Explicit block parameters make it easy to determine at a glance which methods expect a code block. Methods with an explicit code block parameter can also treat the block as an ordinary object instead of some freakish special case. Stylistic considerations aside, explicit code block parameters allow you to do something that is impossible with the implicit variety: When you use explicit block parameters, you can hold onto the block and store a reference to it like any other object. And that means you can execute the block later, perhaps much later, possibly long after the method that caught the block has returned.

The Call Back Problem

To see the utility of being able to hold on to code blocks, let's imagine that some of your colleagues are writing `CopyEdit`, a word-processing program built around your `Document` object. The `CopyEdit` folks are thrilled with the Zenlike simplicity of your `Document` class, but they have requested an enhancement. They need a call back to go

1. The phrase "end of your parameter list" really does mean the end. The `&block` parameter goes after all the other stuff in your parameter list, including those starred catch-all parameters.

off when the document is read from a file, and another that will fire when the document is saved.

The traditional way to solve this problem is to build separate listener objects, something like this:

```
class DocumentSaveListener
  def on_save( doc, path)
    puts "Hey, I've been saved!"
  end
end

class DocumentLoadListener
  def on_load( doc, path)
    puts "Hey I've been loaded!"
  end
end
```

You then give your documents references to the listeners and call the methods at the right time:

```
class Document

  attr_accessor :load_listener
  attr_accessor :save_listener

  # Most of the class omitted...

  def load( path )
    @content = File.read( path )
    load_listener.on_load( self, path ) if load_listener
  end

  def save( path )
    File.open( path, 'w') { |f| f.print( @contents ) }
    save_listener.on_save( self, path ) if save_listener
  end
end
```

Armed with all this, you can hook up your listener to a document and find out when the document gets loaded and saved:

```
doc = Document.new( 'Example', 'Russ', 'It was a dark...' )
doc.load_listener = DocumentLoadListener.new
doc.save_listener = DocumentSaveListener.new

doc.load( 'example.txt' )
doc.save( 'example.txt' )
```

Run the code above and you will see:

```
Hey I've been loaded!
Hey, I've been saved!
```

This listener class approach has some real advantages. The listening code is separate from the inner workings of the `Document` class, and you can swap different listeners in and out whenever you like. The trouble with separate listener classes is that, well, they are a lot of trouble. To make the traditional listener object approach work you need to create those listener classes, instantiate them, and manage their relationships with your documents.

Banking Blocks

A different, and really elegant, way to solve the call back problem is to use explicit code block parameters. Think about it: When you capture a code block in an explicit parameter, you end up with an object that has a single method (the `call` method) containing some code. Isn't this exactly what we laboriously built when we created the `DocumentSaveListener` and `DocumentLoadListener` classes?

Here's our `Document` class rewritten to use code blocks as call backs:

```
class Document

  # Most of the class omitted...

  def on_save( &block )
    @save_listener = block
  end
```

```
    def on_load( &block )
      @load_listener = block
    end

    def load( path )
      @content = File.read( path )
      @load_listener.call( self, path ) if @load_listener
    end

    def save( path )
      File.open( path, 'w' ) { |f| f.print( @contents ) }
      @save_listener.call( self, path ) if @save_listener
    end
  end
```

Listening for the comings and goings of documents is now much simpler; no need for those extra listener objects:

```
my_doc = Document.new( 'Block Based Example', 'russ', '' )

my_doc.on_load do |doc|
  puts "Hey, I've been loaded!"
end

my_doc.on_save do |doc|
  puts "Hey, I've been saved!"
end
```

Not only is the block-based version shorter, using the `on_load` and `on_save` methods has a nice declarative feel to it—concise and clear.

Saving Code Blocks for Lazy Initialization

Being able to capture a code block for later use opens ups other possibilities: For example, you can use saved code blocks for lazy initialization. To see how this works, let's return to the problem of creating lazy documents. This time, imagine that we need to deal with a large number of archival documents. Mostly we just need the title and author of the document so that we can display them to the user, but occasionally we

need the actual content. It would be nice if we could avoid reading the content until we absolutely need it. To this end, we might do something like this:

```
class ArchivalDocument
  attr_reader :title, :author

  def initialize(title, author,  path)
    @title = title
    @author = author
    @path = path
  end

  def content
    @content ||= File.read( @path )
  end
end
```

At first glance the solution shown here seems fine, but it does have one real drawback. The problem is that the `ArchivalDocument` class knows all about where the document content comes from: a file. Contrast this with the original `Document` class, which neither knew nor cared where its contents originated. With `ArchivalDocument`, if you suddenly decide you want to get your document text via HTTP or FTP, well, you are kind of stuck.

Fortunately, we can fix this problem with a simple wave of our saved-code-block wand. Instead of passing in a path when we make a new `ArchivalDocument` instance, we pass in a block, one that returns the document contents when it is called:

```
class BlockBasedArchivalDocument
  attr_reader :title, :author

  def initialize(title, author, &block)
    @title = title
    @author = author
    @initializer_block = block
  end

  def content
    if @initializer_block
      @content = @initializer_block.call
      @initializer_block = nil
```

```
    end
    @content
  end
end
```

This latest implementation means we can still get our document contents from a file, like this:

```
file_doc =  BlockBasedArchivalDocument.new( 'file', 'russ' ) do
  File.read( 'some_text.txt' )
end
```

But we can also get them via HTTP:

```
google_doc = BlockBasedArchivalDocument.new('http', 'russ') do
  Net::HTTP.get_response('www.google.com', '/index.html').body
end
```

Or just make something up:

```
boring_doc = BlockBasedArchivalDocument.new('silly', 'russ') do
  'Ya' * 100
end
```

The examples above look a lot like the initialization block examples we saw in the last chapter, but there is a critical difference. In those earlier examples, the `initialize` method called the code block immediately as the object was being constructed. In contrast, the `BlockBasedArchivalDocument` class waits until someone actually calls the `content` method before firing off the block. In fact, if you never call `content`, then the block will never get called. By using a code block, we get the best of both worlds. We can conjure up the document contents in any way we want, and the conjuring is delayed until we actually need it.

Instant Block Objects

Sometimes it's handy to produce a code block object right here, right now. You want to get hold of the object version of a block, which is actually an instance of the `Proc`

class, without creating a method to catch it. You might, for example, want to create a block object that you can use as the default value for your document listeners. Fortunately, Ruby supplies you with a method for just such an occasion: `lambda`. Here is our earlier document example, the one with the listeners, rewritten to use `lambda` to create a default `Proc` object.

```
class Document
  DEFAULT_LOAD_LISTENER = lambda do |doc, path|
    puts "Loaded: #{path}"
  end

  DEFAULT_SAVE_LISTENER = lambda do |doc, path|
    puts "Saved: #{path}"
  end

  attr_accessor :title, :author, :content

  def initialize( title, author, content='' )
    @title = title
    @author = author
    @content = content
    @save_listener = DEFAULT_SAVE_LISTENER
    @load_listener = DEFAULT_LOAD_LISTENER
  end

  # Rest of the class omitted...
end
```

The idea behind the `lambda` method is that you pass it a code block and the method will pass the corresponding `Proc` object right back at you.

Staying Out of Trouble

When it comes to creating `Proc` objects, beware of false friends. Although calling `Proc.new` is nearly synonymous with `lambda`:

```
from_proc_new = Proc.new { puts "hello from a block" }
```

It's not quite synonymous enough. The object you get back from `Proc.new` differs from what you would get back from `lambda` in two key ways. One relatively innocu-

ous difference is that a `Proc.new` object is very forgiving of the number of arguments passed to its `call` method. Pass too few and it will set the excess block parameters to `nil`; pass too many and it will quietly ignore the extra arguments. In contrast, the `call` method on an object returned by `lambda` acts more like a regular method and will throw an exception if you mess up the argument count.

The second difference is much more critical. Objects from `Proc.new` feature all of the interesting `return`, `break`, and `next` behavior that we touched on in the last couple of chapters. For example, if a `Proc.new` block executes an explicit `return`, Ruby will try to return not just from the block but from the *method that created the block*. This behavior is great for iterators, but it can be a disaster for applications that hang onto code blocks long after the method that created them has returned. In contrast, the `Proc` object returned from `lambda` acts more like a portable method—a return from a `lambda` wrapped block will simply return *from the block* and no further.

Although the issues are deep, the lessons are simple. Lesson one is that if you are calling a method that takes a block, pause for a second before you put a `return`, `next`, or `break` in that block. Does it make sense here? Lesson two is that if you want a block object that behaves like the ones that Ruby generates when you pass a couple of braces into a method, use `Proc.new`. If you want something that will behave more like a regular object with a single method, use `lambda`.[2]

You can also get into trouble with the closure nature of code blocks. The fact that code blocks drag along the variables from the code that defines them is mostly a convenience, but it can also have unexpected and unpleasant consequences. Mostly this has to do with variables staying in scope, and therefore in existence, for longer than you might expect. For example, suppose you write a method that needs to create a large array, one that it will use for a short while. No problem: Just keep the array in a local variable in the method and the array will go out of scope when the method returns:

```
def some_method( doc )
  big_array = Array.new( 10000000 )

  # Do something with big_array...

end
```

2. Just to make things even more exciting, Ruby supports a third way to create `Proc` instances, the `proc` method. The interesting bit is that in 1.8, `proc` was synonymous with `lambda`. In 1.9 it is more like `Proc.new`. Some days I wonder why I get of bed.

So far, so good. But what if you happen to create one of those long-lasting blocks while that large array is still in scope?

```
def some_method( doc )
  big_array = Array.new( 10000000 )

  # Do something with big_array...

  doc.on_load do |d|
    puts "Hey, I've been loaded!"
  end
end
```

What happens is this: Because the big array was in scope when you created the block and the block drags along the local environment with it, the block holds onto a reference to the array—even if it never uses it. This means that the array, with all ten million elements, is going to stay around for as long as the block does, in this case as long as the document is around. Once you are aware of what is going on, a very simple solution offers itself:

```
def some_method(doc)
  big_array = Array.new(10000000)

  # Do something with big_array...

  # And now get rid of it!

  big_array = nil

  doc.on_load do |d|
    puts "Hey, I've been loaded!"
  end
end
```

The lesson here is not that holding onto block references is dangerous, but that you should keep in mind the stuff that you might be unconsciously dragging along with your blocks.

In the Wild

We caught a glimpse of saved code blocks way back in Chapter 9 when we looked at RSpec. If you look at a spec:

```
it "should know how many words it contains" do
  doc = Document.new('example', 'russ', 'hello world')
  doc.word_count.should == 2
end
```

You will see an example of a code block that gets saved for use later. In this case, later comes when RSpec runs all the tests.

You can also find lots of saved code blocks in Rails. There are, for example, the filters you can set up in your controller:

```
class DocumentController < ActionController::Base
  before_filter do | controller |
    # Do something before each action...
  end

  # Rest of the controller...
end
```

As well as the life-cycle hooks in ActiveRecord:

```
class DocumentVersion < ActiveRecord::Base
  after_destroy do | doc_version |
    # My Document is gone!
  end
end
```

Both Rails methods say, "Here's a code block, hang onto it and run it when the time is right."

Both Rake and Capistrano also make extensive use of saved code blocks. Both tools are built around the idea of a task—some defined bit of work that occasionally needs doing, the difference being that while Rake focuses on doing things here on your local machine, Capistrano seeks to rule, or at least configure, machines scattered across the network.

Rake and Capistrano capture what needs to be done in blocks, which they salt away until the time is right. Here is a very simple Capistrano task that knows how to list the `/home` directory on the production machines:

```
desc "List the home directories"
task :list_home, :role => 'production' do
  run "ls -l /home"
end
```

If you pull the covers off of Capistrano, you will find some familiar-looking code. Here, for example, is the Capistrano task method:[3]

```
def task(name, options={}, &block)
  name = name.to_sym
  raise ArgumentError, "expected a block" unless block_given?

  # Some code deleted...

  tasks[name] = TaskDefinition.new( name, self,
    {:desc => next_description(:reset)}.merge(options), &block)

  # Some more code deleted...
end
```

For our purposes the key bit of the `task` method is right there at the end. Notice how the code passes `&block` into the `TaskDefinition` constructor. The `TaskDefinition` class holds onto the block, ready to fire it off should Capistrano decide that this is the task that needs to be done.

Wrapping Up

In this chapter we finished our look at code blocks by seeing them as long-lived containers for Ruby code. We saw how, by using explicit block parameters, you can delay running the code inside of a block until you need it. We looked at a couple of practical applications of this including using code blocks as call backs, whereby your object

3. Edited, as they say on television, to fit your screen.

grabs a code block and executes it when some event occurs. We also saw how you can use code blocks to effect lazy initialization. Since there are rarely silver linings without dark clouds, we also looked at how the code block habit of vacuuming up all the variables in scope when you create the block can turn on you if you are not careful.

Although this chapter completes our hard look at code blocks, we're not really done with them. Code blocks are such an important part of Ruby programming that they will continue to pop right up until the end of this book. For now we are going to turn our attention to Ruby's system of programming hooks, a mechanism devoted to keeping you informed about what's happening as your application runs. It turns out that one way of doing this is to let you supply a code block and . . . . Well, perhaps we should leave that for the next chapter.