CHAPTER 18

Execute Around with a Block

In the last chapter we looked at using code blocks as iterators. Code blocks do make great iterators, but simply running through a collection one element at a time in no way exhausts what you can do with a code block. In this chapter we will take the next step with code blocks by seeing how we can use them as a science fiction transporter device of sorts, capable of delivering code where it is needed in a shimmering flash of light.[1] Along the way we will examine how you can get data into and out of your code blocks. Finally, we will stumble across yet another reason why carefully chosen method names are worth their weight in gold.

Add a Little Logging

If I had to list my five favorite debugging aids, I'd probably have to ponder the bottom four, but number one would be easy: decent logging. Unglamorous and utilitarian, logging is one of those techniques that we don't talk about a lot, but that is vital in real-world applications. If, for example, we store our documents, keyed by name, in a database, we might write something like this:

1. OK, I made up the shimmering flash part.

```
class SomeApplication
  # ...

  def do_something
    doc = Document.load( 'resume.txt' )

    # Do something interesting with the document.

    doc.save
  end
end
```

We might, except that database interactions are not always successful, so we would probably want to sprinkle in some logging:

```
class SomeApplication

  def initialize( logger )
    @logger = logger
  end

  def do_something
    @logger.debug( 'Starting Document load' )
    doc = Document.load( 'resume.txt' )
    @logger.debug( 'Completed Document load' )

    # Do something interesting with the document.

    @logger.debug( 'Starting Document save' )
    doc.save
    @logger.debug( 'Completed Document save' )
  end
end
```

Even better is an explicit log message informing us of disaster:

```
class SomeApplication

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

```
  def do_something
    begin
      @logger.debug( 'Starting Document load' )
      @doc = Document.load( 'resume.txt' )
      @logger.debug( 'Completed Document load' )
    rescue
      @logger.error( 'Load failed!!' )
      raise
    end

    # Do something with the document...

    begin
      @logger.debug( 'Starting Document save' )
      @doc.save
      @logger.debug( 'Completed Document save' )
    rescue
      @logger.error( 'Save failed!!' )
      raise
    end
  end
end
```

This last bit of code is great. It gives us a full account of the document's adventures as it gets loaded and saved, and even shouts for help if something goes wrong. The only problem with this code is that it's horrible! We have managed to take a couple of simple operations—one lonely line of code to load a document and a similar line to save it—and turned them into a nightmare that goes on for half a page. And it only gets worse from there. Between the load and the save we will probably do something with the document, something that will likely rate its own logging. We might even (gasp!) need to deal with more than one document at a time.

Not only is all this logging code tedious to write, it is also hard to read, which means that we have managed to violate both ends of our "clear and concise code" goal. How can we do better? Perhaps we could write special load and save methods, methods that do the logging for us:

```
class SomeApplication

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

```
  def do_something
    @doc = load_with_logging( 'resume.txt'  )

    # Do something with the document...

    save_with_logging( @doc  )
  end

  def load_with_logging( doc  )
    begin
      @logger.debug( 'Starting Document load' )
      doc = Document.load( doc )
      @logger.debug( 'Completed Document load' )
    rescue
      @logger.error( 'Load failed!!' )
      raise
    end
    doc
  end

  def save_with_logging( doc )
    begin
      @logger.debug( 'Starting Document save' )
      doc.save
      @logger.debug( 'Completed Document save' )
    rescue
      @logger.error( 'Save failed!!' )
      raise
    end
  end
end
```

The trouble with this approach is that we will need one of these special "`with_logging`" methods for everything we do. On top of that, each one of our "`with_logging`" methods will keep endlessly repeating the same logging code.

Fortunately, there is a way out. Delivering code where it is needed is exactly what code blocks do so well. We can hide all of the logging nonsense—complete with its associated exception handling—in a method that takes a code block:

```
class SomeApplication

  def do_something
    with_logging('load') { @doc = Document.load( 'resume.txt' ) }

    # Do something with the document...

    with_logging('save') { @doc.save }
  end

  # Rest of the class omitted...

  def with_logging(description)
    begin
      @logger.debug( "Starting #{description}" )
      yield
      @logger.debug( "Completed #{description}" )
    rescue
      @logger.error( "#{description} failed!!")
      raise
    end
  end
end
```

Take a look at the `do_something` method in this last example—it's almost back to its original brevity, but *with logging*. Even better, it's obvious what is going on: We are loading and saving the document, *with logging*. Best of all, since we can pass an arbitrary block into `with_logging`, `with_logging` is completely general; we can do *anything* with logging:

```
class SomeApplication
  def do_something_silly
    with_logging( 'Compute miles in a light year' ) do
      186000 * 60 * 60 * 24 * 365
    end
  end
end
```

When It Absolutely Must Happen

This simple "bury the details in a method that takes a block" technique goes by the name of **execute around**. Use execute around when you have something—like the logging in the previous example—that needs to happen before or after some operation, or when the operation fails with a exception. Instead of laboriously sprinkling intention-obscuring code far and wide, you build a method that takes a code block. Inside the method you do whatever preparation needs doing; in our example it was logging the initial message. Then you call the block, followed by any clean-up work, which in the example was the second log message. You can (and probably should) also catch any exceptions that come roaring out of the block[2] and do the right thing with them.

Now, although the name "execute around" comes from the full-blown idea of a method that does something before and after the block gets executed, there is no rule saying you can't build a slightly degenerate execute around method that omits the after bit:

```
def log_before( description )
  @logger.debug( "Starting #{description}" )
  yield
end
```

Or the before part:

```
def log_after( description )
  yield
  @logger.debug( "Done #{description}" )
end
```

Even if you do leave one or the other parts out, the idea is the same: Execute around uses a code block to interleave some standard bit of processing with whatever it is that the block does.

2. Of course, unexpected exceptions never appear in any of my code, but I understand other people do occasionally experience them.

Setting Up Objects with an Initialization Block

Execute around can also help you get your objects initialized. You can, for example, change the `Document initialize` method to take a block, one that it calls with the new `Document` instance:

```
class Document
  attr_accessor :title, :author, :content

  def initialize(title, author, content = '')
    @title = title
    @author = author
    @content = content
    yield( self ) if block_given?
  end

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

Doing this allows an application that creates a new document to isolate the code that initializes the new document in the block:

```
new_doc = Document.new( 'US Constitution', 'Madison', '' ) do |d|
  d.content << 'We the people'
  d.content << 'In order to form a more perfect union'
  d.content << 'provide for the common defense'
end
```

Using execute around for initialization is generally less about making sure that things happen in a certain sequence and more about making the code readable: *Here*, it says, *is the code that sets up the new object*.

Dragging Your Scope along with the Block

A key part of doing a successful execute around method is paying attention to what goes into and what comes out of the code block. Let's start with the input side of the equation. Programmers who are still getting used to code blocks will sometimes have

the urge to pass lots of arguments from their application code, through the execute around method and down into the code block. For example, you might see something like this:

```
class SomeApplication

  # Rest of the class omitted...

  def do_something
    with_logging('load', nil) { @doc = Document.load( 'book' ) }

    # Do something with the document...

    with_logging('save', @doc) { |the_object| the_object.save }
  end


  def with_logging(description, the_object)
    begin
      @logger.debug( "Starting #{description}" )
      yield( the_object )
      @logger.debug( "Completed #{description}" )
    rescue
      @logger.error( "#{description} failed!!")
      raise
    end
  end
end
```

This code is more complex than it needs to be because it misses a key point about code blocks: All of the variables that are visible just before the opening `do` or `{` are still visible inside the code block. Code blocks drag along the scope in which they were created wherever they go. In the last example, this means that `@doc` object is automatically visible inside the code block—no need to pass it down as an argument.[3]

This doesn't mean that respectable execute around methods don't take any arguments. A good rule of thumb is that the only arguments you should pass from the

3. The technical term for objects with this scope dragging property is *closure*, and some Ruby programmers will use the terms *closure* and *block* interchangeably.

application into an execute around method are those that the execute around method itself, not the block, will use. We can see this in our `with_logging` method. We passed in strings like `'Document load'` and `'Document save'` to the `with_logging` method, strings used by the method itself.

Similarly, there is nothing wrong with the execute around method passing arguments that originate in the method itself into the block; in fact, many execute around methods do exactly that. For example, imagine that you need a method that opens a database connection, does something with it, and then ensures that the connection gets closed. You might come up with something like this:

```
def with_database_connection( connection_info )
  connection = Database.new( connection_info )
  begin
    yield( connection )
  ensure
    connection.close
  end
end
```

Note that the `with_database_connection` method creates the new database connection and then passes it into the block.

Carrying the Answers Back

Another thing you need to consider with execute around methods is that the application might want to return something from the block. It would be reasonable, for example, to expect that our light-year computing method would actually return a very large number:

```
def do_something_silly
  with_logging( 'Compute miles in a light year' ) do
    186000 * 60 * 60 * 24 * 365
  end
end
```

It might be reasonable, but right now it won't happen. The rub is that, up to now, all of our `with_logging` methods have simply tossed out the return value from the block. To make the example above work, we need to do something like:

```
def with_logging(description)
  begin
    @logger.debug( "Starting #{description}" )
    return_value = yield
    @logger.debug( "Completed #{description}" )
    return_value
  rescue
    @logger.error( "#{description} failed!!")
    raise
  end
end
```

This new and improved `with_logging` method captures the return value from the block and returns it as its own return value.

Staying Out of Trouble

Aside from making sure you know what arguments are going into your execute around method and making sure you deliver any return value out of it, the main way to go wrong with execute around is to forget about exceptions. In fact, exception handling is even more important with execute around than it is with iterators, because execute around is all about guarantees. The whole idea of execute around is that the caller is guaranteed that *this* will happen before the code block fires and *that* will happen after. Don't let some stray exception sully the reputation of your method for absolutely, positively getting the job done.

With execute around you also need to consider the human factor: A critical difference between just using execute around and really applying it elegantly lies in the name you pick for your method. A good name should make sense in the context of the application code, the code that is calling the method. Don't think of it so much as naming a new method as naming a new feature that you are adding to the Ruby language.

To see what I mean, imagine that we had been a little less careful in naming our `logging` execute around method:

```
execute_between_logging_statements( "update" ) do
  employee.load
  employee.status = :retired
  employee.save
end
```

Somehow the code above just doesn't sing to you the way this does:

```
with_logging( "update" ) do
  employee.load
  employee.status = :retired
  employee.save
end
```

Sure, we all know that no matter what the name is, it is just a method call. But if you blur your vision a little you can imagine that `with_logging` is an actual bit of Ruby syntax, like `while` or `if`. In fact, your imagination would not be that far off: After defining the `with_logging` method, you now have a language that not only will let you conditionally execute code with an `if` statement and repeatedly execute some code in a `while` loop, but also will let you execute some code wrapped in logging with `with_logging`.

In the Wild

If the `open_database` example of a few pages ago seems strangely familiar, it should—I modeled it on the very familiar `File.open` method that comes with Ruby:

```
# No open file here.

File.open('/etc/passwd') do |f|
  # File open here!
  # Begin cracking the passwords on Russ' computer...
end

# The password file is guaranteed to be closed here.
```

It's also easy to find execute around used to initialize objects. Part of building a Ruby gem, for example, is creating a `Gem::Specification` instance, which describes your new gem in excruciating detail. Here's Rake in the process of filling in its specification:

```
SPEC = Gem::Specification.new do |s|

  #### Basic information.
```

```
    s.name = 'rake'
    s.version = $package_version
    s.summary = "Ruby based make-like utility."
    s.description = <<-EOF
      Rake is a Make-like program implemented in Ruby. Tasks
      and dependencies are specified in standard Ruby syntax.
    EOF

    # Lots and lots omitted!
  end
```

By letting you put all the initialization code in the block, the `Gem::Specification` `initialize` method is helping you make your code a bit easier to read. This is code block as a literary device!

You can find another real-world example of execute around in the ActiveRecord `say_with_time` method. Here is `say_with_time` along with its buddy, `say`:

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

    def say(message, subitem=false)
      write "#{subitem ? "   ->" : "--"} #{message}"
    end

    def say_with_time(message)
      say(message)
      result = nil
      time = Benchmark.measure { result = yield }
      say "%.4fs" % time.real, :subitem
      say("#{result} rows", :subitem) if result.is_a?(Integer)
      result
    end
  end
```

As you can see, the `say_with_time` method takes a string argument as well as a block. When you call `say_with_time` it executes the block and then prints your message along with the amount of time that it took to execute the block.

Finally, since we have spent so much time in this chapter talking about using execute around to add logging to your code, we should probably also give equal time to

the ActiveRecord `silence` method, which turns logging off[4] for the duration of a code block.

Wrapping Up

In this chapter we have looked at the execute around technique. Execute around can help you cope with those times when you have code that frequently needs to come before or after some other code, or both. Execute around suggests that you build a method that takes a block; inside of that method you execute whatever code needs executing before and after you call the block. Creating an execute around method is simple, but, like any method that you build, you do need to pay attention to get the maximum mileage. Name your execute around method carefully—keeping in mind how the method will be used—and pay particular attention to the arguments, both those that go into the method and those that pass between the method and the block.

4. Technically, `silence` turns the logging level up (or is it down?) to ERROR.