CHAPTER 20
Use Hooks to Keep Your Program Informed

Metaprogramming is one of those words that seems to exist purely to scare people. Are we talking about programming beyond programming? Programming turned up to 11? Programming in the next dimension? In fact, metaprogramming—at least as it is practiced in the Ruby world—is a very workman set of coding techniques that allow you to get the results you need with less code. Ruby support for metaprogramming starts by allowing your code to stay amazingly well informed about what's going on around it. With a tiny bit of effort, you can write Ruby programs that know when a new class is created, when a method gets called, and even when the application is about to exit. Of course, all this knowledge would be so much trivia if your program couldn't do anything about it. Fortunately, Ruby programs can do all sorts of things: They can decide that there are still just a few details to take care of before the application exits. They can decide that this error is not really an error but a reasonable request. And they can even reprogram themselves.

In this chapter we will begin our exploration of metaprogramming with the "staying informed" side of the equation by looking at hooks. A Ruby hook is some way—sometimes by supplying a block and sometimes by just overriding a method—to specify the code to be executed when something specific happens. We are going to see how you can use hooks to find out that a class has gained a new subclass, or that a module has been included, or that your program is getting ready to terminate. As usual, we will spend time talking about how you would use these features, and also about how you might stay away from the pointy end of the hook.

Waking Up to a New Subclass

As I say, a **hook** is code that gets called to tell you that something is about to happen or has already happened. A great example of a hook is the one that tells you when a class gains a subclass. To stay informed of the appearance of new subclasses, you define a class-level method called `inherited`. To see how this might work, let's define a very simple base class that does indeed define `inherited`:

```
class SimpleBaseClass
  def self.inherited( new_subclass )
    puts "Hey #{new_subclass} is now a subclass of #{self}!"
  end
end
```

To see the `inherited` method in action, we just need to create a subclass:

```
class ChildClassOne < SimpleBaseClass
end
```

Define the `ChildClassOne` class and the `SimpleBaseClass` `inherited` hook will fire and print this:

```
Hey ChildClassOne is now a subclass of SimpleBaseClass!
```

The `inherited` hook is not a very complicated feature, but one question does immediately spring to mind: What the heck would we ever do with it? Well, you might manage a list of subclasses. To see how you could do that—and why it might be useful—imagine that you have documents stored in many different file formats. Some are stored in plain text files, some are in YAML files,[1] and some, sadly, might actually be stuck in XML files.

Since you do have a fair number of formats, it seems wise to separate the file-reading code from the `Document` class itself. That way you won't have a lot of file format conversion machinery cluttering up the `Document` class. Instead, you'll write a series of reader classes, where each reader class understands a single format and knows how to

1. YAML is a structured file format similar to, but more human friendly than, XML. If you have ever created a Rails database.yml file, you know what YAML looks like.

turn a file in that format into a Document instance. The simplest of the bunch is the one that reads plain text files:

```
class PlainTextReader < DocumentReader
  def self.can_read?(path)
    /.*\.txt/ =~ path
  end

  def initialize(path)
    @path = path
  end

  def read(path)
    File.open(path) do |f|
      title = f.readline.chomp
      author = f.readline.chomp
      content = f.read.chomp
      Document.new( title, author, content )
    end
  end
end
```

Ignoring the DocumentReader superclass, which we will come to in a minute, the PlainTextReader class is about as straightforward as they come. It has an initialize method that picks up the path to the plain text file and a read method that actually turns the contents of that file into a Document instance. The one little twist is the can_read? method: This class method returns true if the PlainTextReader is able to read the file whose path is passed in as an argument. In real life, can_read? would probably peek at the first few bytes of the file to see whether it is in a recognizable format. But to keep the example simple, PlainTextReader actually just looks at the file extension: If the name of the file ends in .txt, then PlainTextReader assumes it is up to reading the file.

We can also define similar readers for YAML and XML:

```
class YAMLReader < DocumentReader
  def self.can_read?(path)
    /.*\.yaml/ =~ path
  end
  def initialize(path)
    @path = path
  end

  def read(path)
    # Lots of simple YAML stuff omitted
  end
end

class XMLReader < DocumentReader
  def self.can_read?(path)
    /.*\.xml/ =~ path
  end

  def initialize(path)
    @path = path
  end

  def read(path)
    # Lots of complicated XML stuff omitted
  end
end
```

You now have all the parts needed to read the different file formats, but the question is, how do you pull them together? Ideally you would have a list of all of the reader classes, a list that the code could search looking for a class that is able to read a given file. This is where the `DocumentReader` superclass comes in:

```
class DocumentReader

  class << self
    attr_reader :reader_classes
  end

  @reader_classes = []

  def self.read(path)
    reader = reader_for(path)
    return nil unless reader
    reader.read(path)
  end
```

```
      def self.reader_for(path)
        reader_class = DocumentReader.reader_classes.find do |klass|
          klass.can_read?(path)
        end
        return reader_class.new(path) if reader_class
        nil
      end

      # One critical bit omitted, but stay tuned...
    end
```

`DocumentReader` sports the ultimate `read` method, a class method that takes a path, calls the `reader_for` method to find a reader for the path, and then uses that reader to read the file. The `reader_for` method looks through the `@reader_classes` array trying to find a volunteer to read the file.

So here is the 64-gigabyte question: How do you populate the `@reader_classes` array? Why, with the `inherited` hook:

```
      # ... the vital missing piece

      def self.inherited(subclass)
        DocumentReader.reader_classes << subclass
      end
```

Every time you define a new `DocumentReader` subclass—in other words, a new file reader—the `DocumentReader` `inherited` hook will go off and add the new class to the running list of readers. That list of reader classes is exactly what the code needs when it is time to find the correct reader for a file. The beauty of doing it this way is that the programmer does not need to maintain the list by hand. You simply make sure that all of the reader classes are subclasses of `DocumentReader` and things take care of themselves.

Modules Want To Be Heard Too

The module analog of `inherited` is `included`. As the name suggests, `included` gets called when a module gets included in a class. So, if we were interested in knowing when our writing-quality module was included in a class, we might add an `included` hook:

```
module WritingQuality
  def self.included(klass)
    puts "Hey, I've been included in #{klass}"
  end

  def number_of_cliches
    # Body of method omitted...
  end
end
```

A common use for the `included` hook is to add some class methods to the including class as your module gets included. Recall from Chapter 16 that when you include a module in your class, all of the module's instance methods suddenly show up as instance methods in the class. So, if you include the `WritingQuality` module in a class, instances of that class will suddenly start sporting the `WritingQuality` method `number_of_cliches`. We've also seen that if you pull a module into a class with `extend`, the module's methods become *class* methods of the class.

A sticky question is this: What should you do if you have a combination of class and instance methods that you want to mix into a class as a unit? You could simply create two modules, one for the instance methods and one for the class methods, and have your host classes do both an `include` and an `extend`, like this:

```
module UsefulInstanceMethods
  def an_instance_method
  end
end

module UsefulClassMethods
  def a_class_method
  end
end

class Host
  include UsefulInstanceMethods
  extend UsefulClassMethods
end
```

Making the class go through both an include and an extend isn't horrible, but it's not elegant either. It would be better if you could get all of the goodness of your modules mixed in, in one go. Fortunately you can. Remember that a module can find out when it is included in a class via the included hook. From there we just need a little bit of ingenuity to get the class methods mixed in:

```
module UsefulMethods
  module ClassMethods
    def a_class_method
    end
  end

  def self.included( host_class )
    host_class.extend( ClassMethods )
  end

  def an_instance_method
  end

  # Rest of the module deleted...
end

class Host
  include UsefulMethods
end
```

Knocking off the extra step required to mix in the class methods may seem like a little thing, and it is. Good code is, however, built from just these tiny bits of courtesy.

Knowing When Your Time Is Up

The at_exit hook is the Ruby's equivalent of the Grim Reaper: It only drops in when you—or rather, your Ruby application—is on its way out. The at_exit hook gets called just before the Ruby interpreter exits, and this is your last chance to get a word in before it's all over. Using at_exit is a bit different from the other hooks we have seen. Instead of overriding something, with at_exit you just *call* at_exit with a block:

```
at_exit do
  puts "Have a nice day."
end
```

The Ruby interpreter will fire off the block just before it expires. An advantage of this code-block approach is that you can call `at_exit` several times, passing in different blocks each time. So along with the `at_exit` above, we might also do:

```
at_exit do
  puts "Goodbye"
end
```

If you do call `at_exit` more than once, then when your application is ready to exit each block will get called in "last in/first out" order. Thus, if we did the two `at_exit` calls in the order shown above, the final words of our program would be:

```
Goodbye
Have a nice day.
```

Such a polite program.

. . . And a Cast of Thousands

While `inherited`, `included`, and `at_exit` are among the most useful—and widely used—hooks that Ruby offers, they are by no means the only ones. The most notable of these remaining hooks is `method_missing`, which we will save for the next three chapters. A less famous, but occasionally useful hook is `method_added` that allows you to listen for new methods being added to a class. You can also listen for changes to global variables with `trace_var`.

The ultimate Ruby hook, however, has got to be `set_trace_func`. With this handy little method you can supply a block that will get called whenever a method gets called or returns, whenever a class definition is opened with the `class` keyword or closed with an `end`, whenever an exception get raised, and whenever—and here's the kicker—a line of code gets executed. This, for example, is one way to find out just how complicated date processing can be:

```
proc_object = proc do |event, file, line, id, binding, klass|
  puts "#{event} in #{file}/#{line} #{id} #{klass}"
end

set_trace_func(proc_object)

require 'date'
```

This code sets up a block to do tracing duty and then requires in `date.rb` from the Ruby standard library. Run the code and you will see something like this:

```
c-return in trace_func_demo.rb/5 set_trace_func Kernel
line in trace_func_demo.rb/7
c-call in trace_func_demo.rb/7 require Kernel
c-call in trace_func_demo.rb/7 set_encoding IO
c-return in trace_func_demo.rb/7 set_encoding IO
c-call in trace_func_demo.rb/7 set_encoding IO
c-return in trace_func_demo.rb/7 set_encoding IO
line in /home/russ/ruby1.9/lib/ruby/1.9.1/date.rb/196
...
```

In full, this output goes on for more than 2,000 lines—and that's just to read in `date.rb`. This loquaciousness underlines the main issue with `set_trace_func`: It's a little too much of a good thing. Turn on `set_trace_func` and prepare to be overwhelmed with data. Still, it's nice to know it's there if you need it.

Staying Out of Trouble

It may seem obvious, but the key to using Ruby hooks is knowing exactly when they will or will not get called. This can be more complicated than it seems. Take, for example, our `DocumentReader` that depends on the `inherited` hook. If all of our reader subclasses are in the same file with the `DocumentReader` file, then it is pretty obvious when the `inherited` method will go off—shortly after the Ruby interpreter reads the end statement of each subclass:

```
class DocumentReader
  # Stuff omitted...
end
```

```
class PlainTextReader < DocumentReader
  # Stuff omitted...
end

# inherited method for PlainTextReader goes off about now...

class YAMLReader < DocumentReader
  # Stuff omitted...
end

# inherited method for YAMLReader goes off about now...
```

Now consider what would happen if we break this code up into several files, with one reader class per file and we require them in:

```
require 'document_reader'

require 'plaintext_reader'  # inherited fires for PlainTextReader
require 'xml_reader'        # inherited fires for XMLReader
require 'yaml_reader'       # inherited fires for YAMLReader
```

The principal remains the same: The `inherited` method will get called just after each subclass is defined, but now it is obscured by the separate files and the `require` statements.

An even bigger surprise is in store if you happen to have a more complex document reader class hierarchy. What if you had some readers that were similar enough that it made sense to build a common subclass:

```
class AsianDocumentReader < DocumentReader
  # Lots of code for dealing with Asian languages...
end

class JapaneseDocumentReader <  AsianDocumentReader
  # Lots of stuff omitted...
end

class ChineseDocumentReader <  AsianDocumentReader
  # Lots of stuff omitted...
end
```

The problem is that this code is going to trigger a `DocumentReader.inherited` call *three* times, once each for the Japanese and Chinese readers and—perhaps unexpectedly—once for the `AsianDocumentReader`. After all, `AsianDocumentReader` is very much a subclass of `DocumentReader`. There are a number of ways to cope with this kind of situation, but in this case it's easiest to just make sure that the `AsianDocumentReader` class never volunteers to read anything:

```
class AsianDocumentReader < DocumentReader
  def self.can_read?(path)
    false
  end

  # Lots of code for dealing with Asian languages...
end
```

The lesson here is that the `inherited` method fires for all of the subclasses, not just the ones you happened to be interested in.

Sometimes the problem is not with hooks getting called too often. Sometimes it's that they don't get called at all. For example, your helpful Ruby interpreter will try to ensure that the `at_exit` blocks do get called right before things shut down. Sometimes, such as during a program or system crash, your Ruby interpreter isn't able to make good on the `at_exit` promise. The bottom line is that to use hooks effectively you need to know exactly when they will be called.

In the Wild

Now let's clear up a mystery that has been with us since way back in Chapter 9. Recall that when we were talking about the `Test::Unit` framework we wondered how, if you had a test in a file, say `simple_test.rb`:

```
require 'test/unit'

class SimpleTest < Test::Unit::TestCase
  def test_addition
    assert_equal 2, 1 + 1
  end
end
```

And you executed that file:

```
$ ruby simple_test.rb
```

The test would run:

```
Loaded suite simple_test
Started
.
Finished in 0.000247 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
```

The question is, how did the test get run? After all, we didn't write a main program into `simple_test.rb`; we just wrote the test class and it seems to run itself. You've probably already guessed the answer: Look inside `Test::Unit` and you will see that it uses `at_exit` to trigger the test just before the Ruby interpreter exits. Here is the actual code:

```
at_exit do
  unless $! || Test::Unit.run?
    exit Test::Unit::AutoRunner.run
  end
end
```

This actually is a fairly sophisticated bit of Ruby: The `unless` statement in the `at_exit` block first looks at the `$!` variable[2] to see whether there has been an error and does nothing if there has been. This check prevents `Test::Unit` from trying to run the tests in the face of gross problems like syntax errors in the test code itself. If `$!` is `nil`, the `unless` statement next checks to see whether the tests have already been run. It is possible, using the `Test::Unit API,` to run the tests manually, and if that is the case `Test::Unit` doesn't want to run them a second time. If neither of these conditions apply, then `Test::Unit` will happily—and automatically—run your tests for you.

2. `$!` is a global variable that Ruby sets to the last exception raised.

Wrapping Up

In this chapter we looked at several of the Ruby hooks that allow you to get some code executed at key moments in the life of your Ruby application. We examined in some detail three of the most common hooks, starting with the `inherited` method that keeps you in the know when a subclass is added to some class. We also looked at module `included` method that lets you know when a module is included in some class, and finally at the `at_exit` hook that lets you get in a word before the Ruby interpreter exits. We saw how the `inherited` hook can be used to allow a class to keep track of its subclasses, how `included` can be used to modify a class as it includes a module, and how `at_exit` is used by `Test::Unit` to run tests automatically. We also saw that our list of three is by no means exhaustive. There are lots of other Ruby hooks, all of them dedicated to letting your code know what is going on.