CHAPTER 24

Update Existing Classes with Monkey Patching

I can remember the day I sat down and really understood object oriented programming. It was early in my career. At that point I had probably spent two or three years programming straight procedural code in languages ranging from assembly to the only slightly less primitive FORTRAN. I had heard about this new language called SMALLTALK, which featured things called objects that were somehow grouped together by other things called classes. I spent the better part of a weekend reading all about this new programming paradigm, and on Sunday afternoon I finally got it. I'd like to tell you that it changed my professional life. I'd like to tell you that, but I'm just not that big of a liar. If memory serves, and it usually does when it comes to remembering yourself being stupid, my conclusion was that all this object and class talk was a complete waste of time. It was only much later, after I had lived with object oriented programming for awhile, that I began to see its value.

I mention this because with this chapter we will begin an extended look at one of the most startling aspects of Ruby: open classes. Ruby's open classes means that you can change the behavior of any class at any time. You can add new methods. You can replace the code behind an existing method. You can even delete methods altogether. Of all the features in Ruby, open classes is the one most likely to provoke that "what a stupid idea!" reaction. So if you are already shaking your head, just stay with me. It turns out that open classes—and the monkey patching technique that goes with them—is actually a very practical solution to a number of programming problems.

Not only that, but looking into the mechanisms behind open classes will give us some deep insight into how the whole Ruby class model works.

This is a pretty big bill for one chapter, so let's jump right in.

Wide-Open Classes

The best way to think about Ruby's changeable classes is to start with variables. If we say this:

```
name = 'Issac'
```

We are doing one of two things: If `name` has not already been defined, then the code snippet above will define it and set it to the string `'Issac'`. Alternatively, if `name` is already defined, the line of code will set it to a new value, perhaps like this:

```
name = 'Asimov'
```

Ruby classes work in exactly the same way. The first time you define a new class, you are, well, defining a new class. You might start with a very minimal version of the `Document` class, one that consists of just the attributes and the `initialize` method:

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

  def initialize(title, author, content)
    @title = title
    @author = author
    @content = content
  end
end
```

There's nothing surprising there, but now for the interesting part. If you write another `class` statement for `Document`:

```
class Document
  def words
    @content.split
  end
```

```
  def word_count
    words.size
  end
end
```

Then you are not defining a new class. Rather, you are *modifying* the existing `Document` class, the same one you defined in the first bit of code. The effect of this second chunk of code is to add the `words` and `word_count` methods to `Document`.

Even better, the changes you make to your classes will be felt instantly by all of the instances of the class. Thus, if we made a `Document` instance:

```
cover_letter = Document.new( 'Letter', 'Russ', "Here's my resume" )
```

And then enhance our `Document` class with yet another method:

```
class Document
  def average_word_length
    len = words.inject(0.0){ |total, word| word.size + total }
    len / words.size
  end
end
```

Then the change will show up instantly in the existing instance:

```
cover_letter.average_word_length
```

Fixing a Broken Class

Nor are you limited to simply adding new methods to an existing class. It is possible, and not that unusual, to redefine existing methods on Ruby classes. This works on the "last `def` wins" principal: If you reopen a class and define a method that already exists, the new definition overwrites the old. This sort of thing is handy when you need to fix a broken class. Consider that our `average_word_length` method features a really ugly bug, one that bites when you try to get the average word length of an empty document:

```
empty_doc = Document.new( 'Empty!', 'Russ', '' )
puts empty_doc.average_word_length
```

Run this code and you will see:

```
NaN
```

If you haven't had the pleasure, let me introduce you to `NaN`, which is short for "Not a Number." `NaN` is Ruby's way of telling you that you produced an invalid floating-point number. We managed it here by dividing `0.0` by `0.0`.[1] Ideally, you would fix the `average_word_length` method right in the original `Document` code and re-release the whole thing. But what if you weren't the author of `Document`? What if you need a fix right now to make your application work? If you absolutely, positively must get it working, you can reopen the document class and fix the method directly:

```
require 'document'   # Pull in original, broken class

# And now fix the method.

class Document
  def average_word_length
    return 0.0 if word_count == 0
    total = words.inject(0.0){ |result, word| word.size + result}
    total / word_count
  end
end
```

This technique of modifying existing classes on the fly goes by the name of **monkey patching**.[2]

Improving Existing Classes

Once you become comfortable with monkey patching, many possibilities open up. This is especially true since there is no rule against monkey patching Ruby's built-in

1. In the world of integer arithmetic, division by zero will result in an exception. Things are not quite so clear cut in the fuzzier realm of floating-point numbers, where dividing by zero will either result in `NaN` or `Infinity`, depending on exactly what you divide.

2. Programming lore has it that the original name was guerrilla patching, which then morphed into gorilla patching (perhaps programmers simply can't spell?), which somehow evolved into monkey patching.

classes. If you are programming in Ruby and that thing is a class, you can change it. For example, we can finally fix a problem that has been nagging us since Chapter 11.[3] Recall that back then we tried to make the addition operator work between documents and strings. We had no trouble making `document + string` work, since doing the addition in that order called the `+` method from the `Document` class, and we had added the necessary smarts to the `Document` class. The trouble started when we tried to add in the reverse order: `string + document`. This expression resulted in a call to the `+` method on the string, and then we hit a brick wall. How could we possibly change the `String` class? Like this:

```
class String
  def +( other )
    if other.kind_of? Document
      new_content = self + other.content
      return Document.new(other.title, other.author, new_content)
    end
    result = self.dup
    result << other.to_str
    result
  end
end
```

Let's run through this code step by step. The first thing we do is to say `class String`. This reopens the `String` class for our homegrown improvements. Next we define a new `+` method, one that becomes *the* `String` `+` method. In the new method we handle the new `String + Document` case along with the more mundane `String + String` case.

Renaming Methods with alias_method

One downside to our `String` modification is that we ended up reproducing the guts of the original `String` `+` method—the boring non-`Document` bit that creates a bigger string from the two smaller strings—when all we really wanted was to get the method to do the right thing with `Document` instances. We can avoid this with `alias_method`. Although the name suggests otherwise, `alias_method` actually copies a method

3. Well, it has been nagging me.

implementation, giving it a new name along the way. For example, our original `Document` class had a method called `word_count`. With `alias_method`, we can create a couple more methods that do exactly the same thing as `word_count`:

```
class Document
  # Stuff omitted...

  def word_count
    words.size
  end

  alias_method :number_of_words, :word_count
  alias_method :size_in_words,   :word_count

  # Stuff omitted...
end
```

Run this code and you'll end up with a trio of methods on your `Document` instances that will all return the number of words in the document.

Aside from letting you easily give a method several different names, `alias_method` comes in handy when you are messing with the innards of an existing class. Here's a version of our `String` monkey patch that uses `alias_method` to avoid reproducing the logic of the original + method:

```
class String
  alias_method :old_addition, :+

  def +( other )
    if other.kind_of? Document
      new_content = self + other.content
      return Document.new(other.title, other.author, new_content)
    end
    old_addition(other)
   end
end
```

Something quite subtle is going on in this code: The call to `alias_method` copies the implementation of the original + method, giving the fresh copy the name

`old_addition`. Having done that, we proceed to override the `+` method, but—and here's the important part— `old_addition` continues to refer to the original unmodified implementation. When the new `+` method calls `old_addition`, we are actually invoking the original `+` method, which does all of the boring string addition work.

Do Anything to Any Class, Anytime

Our use of `alias_method` in the last section underscores another important aspect of open classes: When you reopen a class, you can do anything you could have done the first time. In the same way that we aliased an existing method, we can make a public method private:

```
class Document
  private :word_count
end
```

Or a private method public again:

```
class Document
  public :word_count
end
```

We can even get rid of it all together:

```
class Document
  remove_method :word_count
end
```

As I said earlier in this chapter, the Ruby classes are like variables. You can set them and leave them alone, or you can fiddle with them as much as you need.

In the Wild

Although this chapter has concentrated on the showier aspects of monkey patching, you can also use it to solve a very mundane problem. To see how, consider the `Pathname` class, which comes with your Ruby installation. `Pathname` tries to be the go-to class

for all your file system programming needs. Because it does try to be all things to all paths, the `Pathname` class is long—it sports almost one hundred instance methods.

Since it is such a large class, the authors of `Pathname` have divided the class into chunks of related methods. There is, for example, an initial piece that defines the basics, things like the `initialize` method and various operators:

```
class Pathname

  # Bits deleted...

  def initialize(path)
    # Set up Pathname instance...
  end

  # ==, <=>, etc. methods deleted...
end
```

Then, there is a chunk containing code that will help you read and write the files your `Pathname` instance points at:

```
class Pathname    # * IO *
  def each_line(*args, &block) # :yield: line
    # Iterate through each line in the file...
  end

  def read(*args)
    # Read the contents of the file...
  end

  # ...
end
```

All told, `Pathname` is broken into nine separate bits and then put back together courtesy of monkey patching.

Another common motivation for monkey patching is to scratch an itch. Have you ever wished that there was just one extra method on the `String` class, a method that you seem to need all the time but is inexplicably missing? Apparently a lot of Ruby programmers have wished for exactly the same thing given that adding methods to the

`String` class is somewhat of a cottage industry. For example, the ActiveSupport gem, which provides miscellaneous helpful code to Rails, adds a number of methods to the `String` class. Among these is `blank?`, which returns true if the string is all white space:[4]

```
class String #:nodoc:
  def blank?
    self !~ /\S/
  end
end
```

Another ActiveSupport contribution to `String` has one of the best names in all of coding: `squish!`. The `squish!` method compresses all of the stretches of white space in a string down to a single space each. Thus, if you have ActiveSupport loaded, this:

```
s = ' Ruby        Rocks   '
s.squish!
```

Will leave `s` equal to `'Ruby Rocks'`. Here is `squish!` in all of its evocative joy:

```
module ActiveSupport #:nodoc:
  module CoreExtensions #:nodoc:
    module String #:nodoc:
      module Filters

        # Some code deleted...

        def squish!
          strip!
          gsub!(/\s+/, ' ')
          self
        end

      end
    end
  end
end
```

4. Come to think of it, that's one I need a lot!

As you can see, `squish!` is not directly defined on the `String` class—it actually lives inside the `ActiveSupport::CoreExtensions::String::Filters`[5] module. The `squish!` method makes its way into `String` when its module is mixed into the `String` class, with another bit of monkey patching:

```
class String
  # Lots of stuff deleted...
  include ActiveSupport::CoreExtensions::String::Filters
end
```

Monkey patching the built-in classes is by no means limited to `String`. ActiveRecord adds a number of methods of the `Array` class, so that not only can you get the first item in an array by calling first, you can also get the second, third, fourth, and fifth items:

```
require 'active-support'

title =  'Hitch Hikers Guild To The Galaxy By Douglas Adams'
array = title.split(//)   # Make an array, one letter per entry

array.first            # 'H'
array.second           # 'i'
array.third            # 't'
array.fourth           # 'c'
array.fifth            # 'h'
```

The sequence stops at the fifth item, with one exception: You can ask for the forty-second element of the array:

```
array.forty_two        # 'a'
```

Ah, geek humor.

5. When you are tired of contemplating the metaphysics of monkey patching, you might consider whether this is an example of module nesting gone wild.

Staying Out of Trouble

The dangers of monkey patching really depend on what you are doing with it. For example, it's hard to see any real danger in the step-by-step assembly of a long class like `Pathname`.

A bit riskier is adding a brand new method to a class, along the lines of adding a `blank?` or `squish!` method on `String`. This is still a reasonably safe thing to do. After all, how much damage can a single additional method do? None, unless the class (or its superclass) already had a `blank?` or a `squish!` method, in which case you have just overwritten that original method. There is, of course, the chance that someone else might be patching in their own `squish!` method, overwriting your masterpiece in the process. Not much of a chance, but something to consider.

Next in our lineup of escalating danger is patching some application class, the way we did when we fixed the `average_word_length` method on `Document`. Here we're getting involved in the central mechanism of the class, although monkey patching a class isn't all that different from making a change to it the traditional way. What it comes down to is that safety lies in knowing what you are doing and in writing tests.

The biggest danger in monkey patching arises when you start messing with the primary mechanism of some critical class. We did exactly this kind of thing when we modified the `String` `+` method earlier in the chapter. It's hard to think of a class more critical than `String`, and by changing the `+` method we ran the risk of breaking a fundamental part of Ruby. The bad news is that there is a very high penalty for screwing up a basic class like `String`: All kinds of things are going to break. The good news is that even the most basic of tests are likely to uncover this kind of error very rapidly. You are writing tests, right?

Wrapping Up

In this chapter we took a very basic look at monkey patching classes. A monkey patch is an on-the-fly modification of an existing class. We saw how you can use monkey patching to add new methods to an existing class, to change methods that are already there, and even do things like aliasing or deleting a method. We saw how monkey patching enables you to fix a broken class, enhance a working one, and assemble large classes bit by bit.

Ironically, although simple monkey patching is the most visible aspect of open classes, it is by no means the only—or even the most important—programming technique to take advantage of this Ruby feature. In the next couple of chapters we will explore two other things we can do with classes that never make us say we're sorry.