CHAPTER 14

Use Class Instance Variables

In the last chapter we saw how you can add methods to your classes, methods that concern themselves with the issues that affect a class as a whole. But wherever you have code, you're going to want to have data, which raises the question of where to store your class-level data. In this chapter we will look at the two alternatives that Ruby gives us for storing class-level data, the class variable, and the class instance variable.

We are going to start by looking at class variables, those Ruby things that start with `@@` and seem to be the easy answer to storing your class-related information. Sadly, we will discover that class variables behave in some unfortunate ways, making them less of a solution and more of a problem. For a real solution to the problem of class-level data we will turn to a class instance variable, a slightly less obvious but much more practical solution.

A Quick Review of Class Variables

Imagine that we decide that our `Document` class needs some defaults.[1] We want to be able to set a default paper size, either the U.S. letter-size paper that is popular in the United States or the similar A4 size that is in vogue almost everywhere else. Since any given user will consistently use either U.S. letter or A4, we want to be able to set the

1. Also imagine that, as usual, we have reset the Document class back to its primordial, Chapter 1 form.

paper size at the `Document` class level and have any document created from then on default to using that paper size. So how do you store information associated with a Ruby class?

The obvious answer is to use a class variable. Class variables start with two @'s instead of one and are associated with a class instead of an ordinary instance. So here is `Document` enhanced to use class variables to track its default paper size:

```
class Document

  @@default_paper_size = :a4

  def self.default_paper_size
    @@default_paper_size
  end

  def self.default_paper_size=(new_size)
    @@default_paper_size = new_size
  end

  attr_accessor :title, :author, :content
  attr_accessor :paper_size

  def initialize(title, author, content)
    @title = title
    @author = author
    @content = content
    @paper_size = @@default_paper_size
  end

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

Our new document class starts out by setting the class variable `@@default_paper_size` to `:a4`. Since class variables are not visible to the outside world, we also supply a pair of accessor methods. A nice thing about class variables is that they *are* visible to instances of the class, so inside the `initialize` method we can pick up the value of `@@default_paper_size` as the paper size for the new document. On the surface, the class variable seems like an ideal solution to our problem. But all is definitely not well.

Wandering Variables

To understand the problem with class variables, we need to look at how they are resolved. As the name suggests, a class variable is one that is associated with a class. But which class? Therein lies a tale: When you say `@@default_paper_size = :a4`, Ruby needs to figure out which class will provide the home for the `@@default_paper_size` variable. Ruby starts by looking at the current class. Does the current class already have an `@@default_paper_size` class variable? If it does, then the search is over and the `@@default_paper_size` on the current class becomes `:a4`.

Here's where the story gets interesting: If the class variable is not defined in the current class, Ruby will go looking up the inheritance tree for it. So if there is no `@@default_paper_size` in the current class, Ruby will look for one in the superclass and then the super superclass, until it either finds a `@@default_paper_size` defined on one of those classes or runs out of classes. If Ruby does find `@@default_paper_size` somewhere in the inheritance tree, that's the one that gets set. If Ruby runs out of classes without finding `@@default_paper_size`, then it will create a new class variable in the current class. Looking up the value of a class variable works pretty much the same way. Ruby starts with the current class and looks up the inheritance tree; it either finds the variable or runs out of classes and throws a `NameError` exception.

Superficially, this sounds very object oriented and reasonable. The problem is that this method for resolving class variables means they have a tendency to wander from class to class. To see what I mean, let's play out a scenario very similar to our earlier paper size example, but with a very different—and unpleasant—outcome. Imagine your `Document` class has become so popular that a number of groups are building systems that use it. In particular, one group is building an application that helps people write resumes.

Since resumes are a very specific kind of `Document`, the resume group decides they need their own `Document` subclass:

```
class Resume < Document
  @@default_font = :arial

  def self.default_font=(font)
    @@default_font = font
  end
```

```
  def self.default_font
    @@default_font
  end

  attr_accessor :font


  def initialize
    @font = @@default_font
  end

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

The key feature of the resume class is that it defines a default font as a class variable, along with class methods to set and get the default font. Thinking about how class variables work, we know that when `@@default_font = :arial` gets run, Ruby will look at the `Resume` class, and then at `Document`, and right on up the inheritance tree. Finding no `@@default_font` defined anywhere, Ruby will set `@@default_font` on the `Resume` class. All very reasonable.

Now imagine that there is a second group out there, one that is intent on inflicting yet another PowerPoint or KeyNote clone on the world. They too build a subclass of `Document`, and, since even mundane minds frequently think alike, they also have a default font class variable, although they pick a different font:

```
class Presentation < Document
  @@default_font = :nimbus

  def self.default_font=(font)
    @@default_font = font
  end


  def self.default_font
    @@default_font
  end

  attr_accessor :font
```

```
  def initialize
    @font = @@default_font
  end

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

Again, no problem. When Ruby is confronted with `@@default_font = :nimbus`, it will look at the `Presentation` class (nope), and then at `Resume` (no again), and so on, and eventually decide to attach `@@default_font` to the `Presentation` class. Since there are two `@@default_font`s, one on the `Document` class and one on the `Presentation` class, the two classes can live side by side in the same application with no problem.

Now for the punch line: Ignorant of the goings on in `Resume` and `Presentation`, you decide to add a `@@default_font` class variable to your `Document` class:

```
class Document
  @@default_font = :times

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

The result of this simple change is that all Hell breaks loose. Here's how: The `Document` class needs to be loaded before `Resume` and `Presentation`—it's the superclass after all. This means the `Document` class `@@default_font` will get set first, which means that whenever either of the subclasses goes looking for `@@default_font`, it will find the one in `Document`. Remember, Ruby looks up the inheritance tree *first*. So your seemingly low-impact change to the `Document` class has changed the behavior of both subclasses. Instead of two separate default font variables, one attached to `Presentation` and the other to `Resume`, there is now only one variable, one that lives up in the `Document` class.

To see how nasty that is, consider that this:

```
require 'document'
require 'resume'        # Load Resume first
require 'presentation'  # then Presentation
```

Sets the default font for all documents to the `Presentation` class's favorite font, `:nimbus`. But if you change the order of the `require` statements:

```
require 'document'
require 'presentation' # Load Presentation first
require 'resume'       # then Resume
```

Then the default font for all documents gets set to `:arial`.

If this "It's my variable!" versus "No! It's mine!" argument sounds familiar, it should. This is exactly the kind of situation we run into with global variables—and is also the reason why coders got out of the global variable business a long time ago. The real problem with class variables is that they are not so much variables attached to a specific class as they are global variables with a slightly restricted realm. In fact, Rubyist David Black calls class variables "vertical global variables," vertical in that they are restricted to a single inheritance tree and global in that they are very visible within that tree.

Getting Control of the Data in Your Class

Clearly, we need an alternative way of associating some data with a class, a technique that is more controllable than class variables. The more controllable alternative to the class variable is the class instance variable. The good news is that class instance variables are not really a new thing. They are in fact garden-variety, single `@` instance variables that happen to find themselves attached to a class object.

To see how this might work, recall from the last chapter that inside a class method, `self` is always the class. Now ask yourself this: What would happen if you set an ordinary instance variable inside a class method? Perhaps like this:

```
class Document

  @default_font = :times

  def self.default_font=(font)
    @default_font = font
  end

  def self.default_font
    @default_font
  end
```

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

The answer is that since `@default_font = font` always sets an instance variable on `self`, the `default_font=` method above will set the `@default_font` instance variable on the `Document` object. This means that armed with the code above we can now set our document-wide default font:

```
Document.default_font = :arial
```

To get at the `Document` default font, all you need to do is call the right class method, which is exactly what the `Document initialize` method does:

```
def initialize(title, author)
  @title = title
  @author = author
  @font = Document.default_font
end
```

Class instance variables are a very Ruby solution to the problem holding onto classwide values. There is no extra syntax and no elaborate special case rules: `@default_font` is simply an instance variable on an object. The only remotely interesting thing here is that the object happens to be a class.

Class Instance Variables and Subclasses

To paraphrase my mother, it's all fun and games until someone starts writing subclasses. Recall that we didn't have any problems with the `@@` class variables until we started messing with subclasses. That was when the class variables started wandering from class to class. Do class instance variables fare any better when the subclasses start flying?

The short answer is that class instance variables do just fine with subclasses. To see how, imagine that we have our `Presentation` subclass and, `Presentation` has its own idea of a default font:

```
class Presentation < Document

  @default_font = :nimbus
```

```
  class << self
    attr_accessor :default_font
  end

  def initialize(title, author)
    @title = title
    @author = author
    @font = Presentation.default_font
  end

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

What this code does is create a second class instance variable, this time attached to the `Presentation` class. The `Presentation` `@default_font` is completely separate from the `Document` `@default_font`, and as long as you are careful with which one you are talking about, `Presentation.default_font` or `Document.default_font`, life will be good.

Adding Some Convenience to Your Class Instance Variables

Since class instance variables are just plain old instance variables that happen to be hanging off a class object, we can use all our Ruby trickery to make living with them more pleasant. For example, there is no reason for us to be writing those boring `default_font` getter and setter methods given that Ruby supplies us with the nice `attr_accessor` for just such occasions. So how do you use `attr_accessor` to get at a class instance variable? After all, if you just do this:

```
class Document
  attr_accessor :default_font
end
```

You end up being able to get and set an *instance* variable called `default_font`. Fortunately, we already have the answer. Remember that class methods are just singleton methods on a class object. The trick to defining class-level attributes is to make `self` be the `Document` singleton class first:

```
class Document

  @default_font = :times

  class << self
    attr_accessor :default_font
  end

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

Run this code and you will end up with a `Document` class that has a couple of class methods, one to get the default font and the other to set it.

In the Wild

Examples of class instance variables are easy to find in the Ruby code base. For example, most Rails programmers know how to set up callbacks, methods that get called at particular moments in the life of their ActiveRecord objects. Here, for example, is a class that has arranged for the `handle_after_save` method to get called just after a `Person` instance is saved into the database:

```
class Person < ActiveRecord::Base
  after_save :handle_after_save

  def handle_after_save
    # Do something after the record is saved...
  end
end
```

Clearly, the fact that we do want `handle_after_save` called after each record save is an attribute of the `Person` class, and that is where ActiveRecord saves this information. Poke around in ActiveRecord and you will find a class instance variable called `@after_save_callbacks`. Nor is `@after_save_callbacks` lonely. Dig into ActiveRecord a bit more and you will find a number of similar class instance variables with names like `@after_update_callbacks` and `@after_validation_callbacks`.

Despite the amount of space I've used in this chapter to cast aspersions on class variables, people do use them successfully. For example, the URI class that comes with

your Ruby install and enables you to make sense of things like "http://russolsen.com" and "sendto:russ@russolsen.com" uses an `@@` variable quite happily. When you say something like this:

```
my_uri = URI.parse('http://www.russolsen.com')
```

The `parse` method will consult `@@schemes`, which is a hash that maps each URI scheme—`'http'` in this case—to the class that knows how to parse URIs with that scheme. The interesting part of `@@schemes` is the way it gets filled in. Initially, the basic URI code simply sets `@@schemes` to an empty hash:

```
@@schemes = {}
```

Then the classes that know how to parse the different flavors of URIs get loaded. The last thing each of these classes does is add itself to the master list of schemes, so that in `uri/http.rb` you will find:

```
class HTTP
  # Lots of code omitted...
end
@@schemes['HTTP'] = HTTP
```

While in `uri/sendto.rb` you will see:

```
class MailTo
  # Lots of code omitted...
end

@@schemes['MAILTO'] = MailTo
```

One other interesting thing about URI is that `@@schemes` is not actually a class variable—it is in fact a module variable. Ruby modules are a bit like stunted classes, but in the same way that you can hang class variables on classes, you can stick module variables on modules. So if we pull back the camera on the URI code, we see:

```
module URI
   # ...
```

```
    @@schemes = {}

  end

  module URI
    class HTTP
       # Lots of code omitted...
    end

    @@schemes['HTTP'] = HTTP
  end
```

We will be seeing a lot more of modules in Chapters 15 and 16.

Staying Out of Trouble

The URI code avoids ending up in class variable Hell by setting `@@schemes` variable very early on, before any of the individual scheme-parsing code gets loaded. This ensures that there is a single `@@schemes` variable and that it is properly attached to the `URI` module where it belongs. If you must use class variables, then this is a good strategy. A better strategy is to reread the title of this chapter and stick to class instance variables.

Wrapping Up

One of the joys of using Ruby is how well the language adheres to the principal of least surprise: If you are wondering how to do something, there's a good chance that your first guess will be correct. Still no language is perfect, and Ruby does harbor a few unfortunate[2] design decisions. The class variable would get my vote as the most unfortunate Ruby feature of all: Class variables seem designed to trip up newcomers to Ruby and even manage to surprise experienced Ruby developers on occasion. The good news is that class instance variables are just waiting there to fill in.

2. Yes, that's the word.