CHAPTER 13

Get the Behavior You Need with Singleton and Class Methods

Much of programming is about building models of the world. Social networking systems model the relationships between people. Accounting systems model the flow of money. Flight simulations model airplanes in flight (or slamming into the ground if I'm at the controls). You can, in fact, look at all of object oriented programming as a support system for this kind of modeling. We build classes to describe groups of similar things, and we have instances to represent the things themselves.

Usually, the class/instance approximation works fine: You define a class called `American`, and you define methods that indicate instances of this class like hamburgers and baseball. The problem is that the class/instance approximation is exactly that, an approximation. I am, for example, an American who does indeed like the occasional burger, but I can't tell you the last time I willingly sat through a baseball game.

What do you do when the central approximation of object oriented programming breaks down, when your instance does not want to follow the rules laid down by its class? In this chapter we will look at Ruby's answer to this question: the singleton method. We will see how singleton methods allow you to produce objects with an independent streak, objects whose behavior is not completely controlled by their class. We will also see how class methods are actually just singleton methods by another name.

A Stubby Puzzle

Let's start with a question. Recall that back in Chapter 9 we saw how RSpec made it easy to create stub objects, those stand-ins for real objects that make writing tests less of a pain. We saw that if, inside of a spec, you did something like this:

```
stub_printer = stub :available? => true, :render => nil
```

You would end up, in the form of `stub_printer`, with an object having two methods, `available?` and `render`:

```
stub_printer.available?  # Always returns true
stub_printer.render      # Always returns nil
```

We also saw in Chapter 9 that RSpec doesn't limit you to one stub object at a time, so you could easily conjure up a second stub, perhaps for a font:

```
stub_font = stub :size => 14, :name => 'Courier'
```

Now for the question: If you look at the classes of these two stub objects:

```
puts stub_printer.class
puts stub_font.class
```

You will discover that they are one and the same:

```
Spec::Mocks::Mock
Spec::Mocks::Mock
```

How is this possible? After all, the printer object supports different methods than the font object. How could they both be of the same class?

The answer is that the `available?` and `render` methods on the printer instance, as well as the `size` and `name` methods on the font instance, are singleton methods. In Ruby, a **singleton method** is a method that is defined for exactly one object instance.[1]

1. Let me also hasten to add that the term "singleton" as it is used here has nothing to do with the Singleton Pattern of design patterns fame. It's just an unfortunate collision of terminology. If you feel a bit put out by this, consider the plight of the guy who wrote the Ruby design patterns book.

It's as though Ruby objects can declare independence from their class and say, "Yeah, I know that no other `Spec::Mocks::Mock` instance has an `available?` method, but I'm special."

You can hang a singleton method on just about any object at any time.[2] The mechanics of defining singleton methods are really pretty simple: Instead of saying `def method_name` as you would to define a regular garden-variety method, you define a singleton method with `def instance.method_name`. If, for example, you wanted to create your own stub printer by hand, you could say this:

```
hand_built_stub_printer = Object.new

def hand_built_stub_printer.available?
  true
end

def hand_built_stub_printer.render( content )
  nil
end
```

You could then call `hand_built_stub_printer.available?` and `render`. Singleton methods are in all respects ordinary methods: They can accept arguments, return values, and do anything else that a regular method can do. The only difference is that singleton methods are stuck to a single object instance.

Singleton methods override any regular, class-defined methods. For example, if you run the (admittedly more fun than useful) code shown here:

```
uncooperative = "Don't ask my class"

def uncooperative.class
  "I'm not telling"
end

puts uncooperative.class
```

2. Well, any object except for instances of the numeric classes and symbols, neither of which supports singleton methods.

You will see:

```
I'm not telling
```

There is also an alternative syntax for defining singleton methods, one that can be less bulky if you are creating a lot of methods:

```
hand_built_stub_printer = Object.new

class << hand_built_stub_printer
  def available?          # A singleton method
    true
  end

  def render              # Another one
    nil
  end
end
```

Either way you say it, you end up with the same singleton methods.

A Hidden, but Real Class

So how does Ruby pull off the singleton method trick? The key is that every Ruby object carries around an additional, somewhat shadowy class of its own. As you can see in Figure 13-1, this more or less secret class—the **singleton** class—sits between every object and its regular class.[3] The singleton class starts out as just a methodless shell and is therefore pretty invisible.[4] It's only when you add something to it that the singleton class steps out of the shadows and makes its existence felt.

Since it sits directly above the instance, the singleton class has the first say on how the object is going to behave, which is why methods defined in the singleton class will win out over methods defined in the object's regular class, and in the superclasses.

3. Singleton classes are also known as metaclasses or eigenclasses. Although terminology is mostly in the ear of the beholder, I like the more descriptive and less pretentious “'singleton.”

4. In fact, since the average Ruby object never uses its singleton class, Ruby implementations will typically delay creating the singleton classes until they are actually needed. This is, however, just an implementation issue. If you do look, the singleton class will always be there for you.

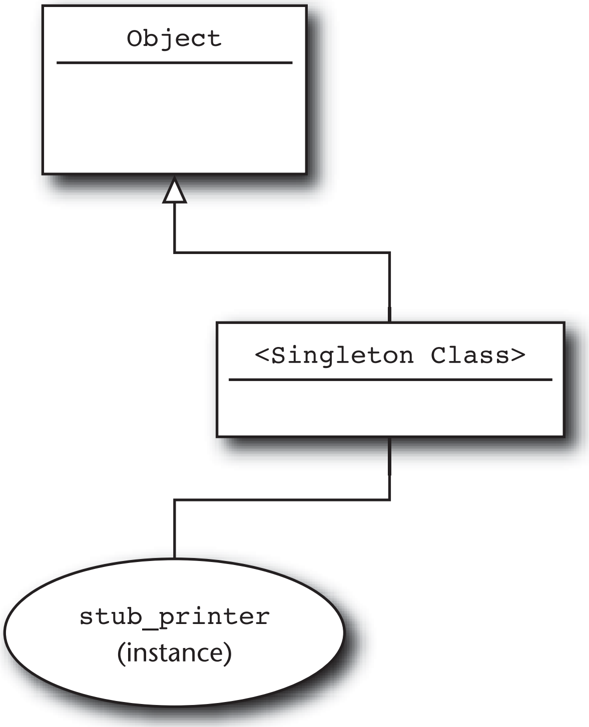

Figure 13-1 The singleton class

Don't think that the singleton class is just a convenient fiction, either. There really is an actual Ruby class hidden in there. You can even get hold of the singleton class, like this:

```
singleton_class = class << hand_built_stub_printer
  self
end
```

The code shown here may look bizarre, but it is straightforward: When you do the `class << hand_built_stub_printer`, you change context so that `self` is the singleton class. Since class definitions, like most Ruby expressions, return the last thing they evaluate, simply sticking `self` inside the class definition causes the whole class statement to return the singleton class.

Class Methods: Singletons in Plain Sight

A common programmer reaction when bumping into singleton methods for the first time is to dismiss them as an interesting but mostly useless feature. After all, outside of a few specialized cases such as stubs and mocks, why bother building classes of consistent behavior only to override that behavior with a singleton method? In fact, this sensible, pragmatic assessment turns out to be completely mistaken. There is one particular application of singleton methods that is so pervasive that it is practically impossible to build a Ruby program without it. We even have a special name for these ubiquitous singleton methods, **class methods**.

To see how class methods are actually singleton methods in disguise, let's build another singleton method, this time one that prints out some interesting information about its host object:

```
my_object = Document.new('War and Peace', 'Tolstoy',
                         'All happy families...')

def my_object.explain
  puts "self is #{self}"
  puts "and its class is #{self.class}"
end

my_object.explain
```

Run the code above and you will get output that looks something like this:

```
self is #<Document:0xb7bc2ca0>
and its class is Document
```

No surprise here, we are just traveling over ground that we covered earlier. Next, however, comes the twist: What if, instead of an instance of `Document`, we defined the `explain` method on the `Document` class itself?

```
my_object = Document

def my_object.explain
  puts "self is #{self}"
  puts "and its class is #{self.class}"
end

my_object.explain
```

Do that and you will get the following output:

```
self is Document
and its class is Class
```

Since `my_object` is just a reference to `Document`, we can also call the `explain` method like this:

```
Document.explain
```

We can also define the `explain` method on `Document` explicitly:

```
def Document.explain
  puts "self is #{self}"
  puts "and its class is #{self.class}"
end
```

If the code above looks familiar, it should, since it is a typical Ruby class method definition. It is also a singleton method definition! If you think about it, this all makes sense: Any given class, say, `Document`, is an instance of `Class`. This means that it inherits all kinds of methods from `Class`, methods like `name` and `superclass`. When we want to add a class method, we want that new method to exist only on the one class (`Document` in the example), not on *all* classes. Since the object that goes by the name of `Document` is an instance of `Class`, we need to create a method that exists only on the one object (`Document`) and not on any of the other instances of the same class. What we need is a singleton method.

Once you get used to the idea that a class method is just a singleton method on an instance of `Class`, a lot of things that you learned in Ruby 101 start to make sense. For example, you will sometimes see the following syntax used to define class methods:

```
class Document
  class << self
    def find_by_name( name )
```

```
      # Find a document by name...
    end

    def find_by_id( doc_id )
      # Find a document by id
    end
  end
end
```

This code only makes sense in light of the *class method = singleton method* equation: It is simply the `class << some_object` syntax applied to the `Document` class.

In the Wild

Class methods abound in real Ruby programs. Class methods are the perfect home for the code that is related to a class but independent of any given instance of the class. For example, we've seen that each ActiveRecord model class is associated with a particular table in the database. So that if you had this:

```
class Author < ActiveRecord::Base
end
```

You would have a class that is associated with a database table. Which table? For that, you need to ask the *class*, not an instance:

```
my_table_name = Author.table_name
```

A common use for class methods is to provide alternative methods for constructing new instances. The Ruby library `Date` class, for example, comes with a whole raft of class methods that create new instances. You can, for example, get a date from the year, the month, and the day:

```
require 'date'
xmas = Date.civil( 2010, 12, 25 )
```

Or by the year and the day of that year:

```
xmas = Date.ordinal( 2010, 359 )
```

Or by the day, the week number, and the day of the week:

```
xmas = Date.commercial( 2010, 51, 6 )
```

If you have many different ways that you might create an object, a set of well-named class methods is generally clearer than making the user supply all sorts of clever arguments to the `new` method.

Plain, nonclass singleton methods are as rare as class methods are common. In fact, their main use in real code is the one we explored earlier in this chapter, building mocks and stubs for testing frameworks. Both RSpec and the Mocha framework that we looked at briefly in Chapter 9 use singleton methods to do their mocking magic. But don't take my word for it, look for yourself:

```
describe "Singleton methods in stubs" do
  it "is just a demonstration of stubs as singleton methods" do
    stub_printer = stub :available? => true, :render => nil
    pp stub_printer.singleton_methods
  end
end
```

This RSpec example creates our familiar stubbed-out printer and then uses `singleton_methods`—part of the arsenal of every Ruby object—to print out a list of the names of all of the singleton methods. Run this spec and you will see:

```
[:available?, :render]
```

If you do the same kind of thing with the mocks and stubs created by the Mocha gem, you will discover that they too rely on singleton methods.

Staying Out of Trouble

Most of the problems you are likely to encounter with singleton methods, particularly in their role as class methods, will likely stem from simple confusion. Easiest to deal with is confusion over scope. Remember, when you define a class method, it is a method attached to a class. The instances of the class will not know anything about that method. Thus, if you define a class method on `Document`:

```
class Document
  def self.create_test_document( length )
    Document.new( 'test', 'test', 'test ' * length )
  end

  # ...
end
```

Then you can call that method via the class:

```
book = Document.create_test_document( 10000 )
```

But `Document` instances are completely ignorant of the `Document` class methods, so that this:

```
longer_doc = book.create_test_document( 20000 )
```

Will give you this:

```
NoMethodError: undefined method `create_test_document'
 for #<Document:0xb7cd7c6c>
```

Well, perhaps not completely ignorant, since instances do know all about their classes:

```
longer_doc = book.class.create_test_document( 20000 )
```

A bit more subtle is the confusion over the value of `self` during the execution of a class method when you mix classes, subclasses, and class methods. To see what I mean, consider this simple pair of classes:

```
class Parent
  def self.who_am_i
    puts "The value of self is #{self}"
  end
end

class Child < Parent
end
```

Now, clearly, if you run `Parent.who_am_i` you would expect the following output:

```
The value of self is Parent
```

But what happens if you run `Child.who_am_i`? The answer is that `self` is always the thing before the period when you called the class method:

```
The value of self is Child
```

While this behavior can be a bit unsettling, it is actually the secret behind some very powerful Ruby metaprogramming techniques, techniques that we will take up in Chapter 26.

Wrapping Up

In this chapter we took a tour of singleton methods and learned how they work. We saw that singleton methods are great for those times when you need an object with some unique behavior. We saw that singleton methods are hidden inside the more or less secret singleton class that is part of most Ruby objects. We saw that the singleton class is a real class, one that you can actually get hold of. We saw how singleton methods can really shine in testing, where they make it easy to construct stub and mock objects. But we also saw that the most common use of singleton methods is for class methods, which are just singleton methods defined on the instances of `Class`.