CHAPTER 7

Treat Everything Like an Object—Because Everything Is

Early in my career I worked with a mainframe operating system that had a variety of file types. That old OS supported text files for your documentation, source files for your code, executable files for your programs, and data files for your output. Actually, "supported" is not really the right word. "Imposed with an iron fist" would be more accurate. Creating new files involved specifying the file type using a complex and arcane syntax. You needed to use one command for copying data files and a similar, but subtlety different, command for copying source files. Converting from one file type to another was a task worthy of a Ph.D. I'm sure that the designers of that ancient system had originally set out to make life easier for their users by imposing some order on the everyday complexities of computing. Along the way, however, they lost sight of something important: Sometimes—like when you are trying to make a copy—a file is just a file.

Ruby is an object oriented programming language, which means that the world of Ruby is a world of objects, instances of `Date` and `String` and `Document` and a thousand other classes. As different as all these objects are bound to be, at some level they are all just objects. In the pages that follow we will focus on the things that all Ruby objects have in common. We will start by doing a quick tour of the Ruby object system and then look at how pervasive objects are in Ruby. Next we will look at the methods common to all Ruby objects, the basic toolkit that every object inherits, and at how you can control access to methods. We will round out the chapter by looking at how classes and methods play a key role in the infrastructure of Ruby itself and at some avoidable potholes lurking in the land of the object.

A Quick Review of Classes, Instances, and Methods

On the surface, Ruby is a very conventional, almost boring, object oriented language. Every Ruby object is an instance of some class. Classes mainly earn their keep by providing two key things: First, classes act as containers for methods:

```
class Document

  # Most of the class omitted...

  # A method

  def words
    @content.split
  end

  # And another one

  def word_count
    words.size
  end
end
```

Second, classes are also factories, factories for making instances:

```
doc = Document.new( 'Ethics', 'Spinoza', 'By that which is...' )
```

Once you have an instance, you can call methods on it using the more or less universal object oriented syntax of `instance.method_name`:

```
doc.word_count
```

During a method call, Ruby sets `self` to the instance that you called the method on, so that if you added this to `Document`:

```
class Document
  # Most of the class on holiday...

  def about_me
    puts "I am #{self}"
    puts "My title is #{self.title}"
    puts "I have #{self.word_count} words"
  end
end
```

And then ran `doc.about_me`, you would see something like:

```
I am #<Document:0x8766ed4>
My title is Ethics
I have 4 words
```

Ruby treats `self` as a sort of default object: When you call a method without an explicit object reference, Ruby assumes that you meant to call the method on `self`, so that a call to plain old `word_count` gets translated to `self.word_count`. You should rely on this assumption in your code: Don't write `self.word_count` when a plain `word_count` will do.[1]

Every class—except one—has a superclass, someplace that the class can turn to when someone calls a method that the class doesn't recognize. If you don't specify a superclass when you are defining a new class, the new class automatically becomes a direct subclass of `Object`. Our `Document` class, for example, is a direct subclass of `Object`. Alternatively, you can specify a different superclass:

```
# RomanceNovel is a subclass of Document,
# which is a subclass of Object

class RomanceNovel < Document
  # Lot's of steamy stuff omitted...
end
```

1. You Python programmers know who I'm talking to!

The Ruby technique for resolving methods is straight out of the object oriented handbook: Look for the method in the class of the object. If it is there, call it and you are done. If not, move on to the superclass and try there. Repeat until you either find the method or run out of superclasses.[2]

Objects All the Way Down

While Ruby's basic object system is not the most original bit of technology, it does have something very powerful going for it: consistency. You can see just how consistent Ruby's object oriented philosophy is by computing the absolute value of a number:

```
-3.abs            # Returns 3
```

Nothing very exciting there … except for the syntax. Why is it `-3.abs` and not `abs(-3)`? The answer is both simple and profound: In Ruby the number `-3` is an object. When you say `-3.abs`, you are calling the `abs` method on an object, an object that goes by the name `-3`. It's not just the numbers either. In Ruby, strings and symbols and regular expressions are all objects, objects that come equipped with their very own methods:

```
# Call some methods on some objects

"abc".upcase
:abc.length
/abc/.class
```

In fact, virtually everything you come across in Ruby is an object. You might, for example, think that `true` and `false` are a couple of special Ruby language constructs. Not so—they're just objects:

```
# Call some methods on a couple of familiar objects

true.class          # Returns Trueclass
false.nil?          # False is close, but not nil
```

2. Ruby does add some cool twists to its fairly vanilla object model, twists that we are going to explore in Chapters 13, 16, and 24.

So are the classes:

```
true.class.class # Returns Class
```

Even our old buddy—and sometimes nemesis—`nil` is an object:

```
nil.class             # Returns NilClass
nil.nil?              # Yes, finally true!
```

In Ruby, if you can reference it with a variable, it's an object.

The Importance of Being an Object

Actually, if you can reference it with a variable it's probably not just an object, but an *Object*, an instance of the `Object` class.[3] Since virtually all Ruby objects can trace their ancestry back to `Object`, virtually all Ruby objects have a set of methods in common: the ones they inherit from `Object`. So the next time you call `class` or `instance_of?` to see just what sort of object you have, thank the `Object` class for supplying those methods. The `Object` class is also the source of the well-worn `to_s` method, which returns a string representation of your object. It's the `to_s` method that `puts` relies on to turn its arguments into printable strings.[4] Like a lot of `Object` methods, you can rely on the default implementation of `to_s`. So if you ran this:

```
doc = Document.new( 'Emma', 'Austin', 'Emma Woodhouse, ...' )
puts doc
```

You would see something like:

```
#<Document:0x8767120>
```

3. The reason I hedge is that Ruby 1.9 added the `BasicObject` class, as the new superclass of `Object`. `BasicObject` instances are, however, few and far between and tend to be used in a way that masks their true identity. All these mysteries will be explained in Chapter 22.

4. This is a bit of a simplification, but not much. Many of the `Object` methods actually live in the `Kernel` module, which is mixed into the `Object` class, a process that we will explore in Chapter 16. For most practical purposes this is a distinction without a difference.

Alternatively, you can override `to_s` for your own purposes:

```
class Document
  # Mostly omitted...

  def to_s
    "Document: #{title} by #{author}"
  end
end
```

With the class above, our documents would print out something like:

```
Document: Emma by Austin
```

Instances of `Object` also inherit some more esoteric talents. For example, the `eval` method, defined by `Object`, takes a string and executes the string *as if it were Ruby code*. The possibilities with `eval` are literally limitless: Having `eval` around means that every Ruby programmer has the entire Ruby language available at a moment's notice. You can, to take an easy example, create a quick riff on `irb` with nothing more than `eval` and a handful of other `Object` supplied methods:[5]

```
while true
  print "Cmd> "
  cmd = gets
  puts( eval( cmd ) )
end
```

Run the code above and you will find yourself in a very `irb`-like loop:

```
Cmd> 2 + 2
4
Cmd> puts "hello world"
hello world
```

5. The `print` method prints what you tell it to, without mixing in any additional newline characters the way that `puts` does. The `gets` method is the inverse of `puts`: It reads a string.

The Object class also supplies a set of reflection-oriented methods, methods that let you dig into the internals of an object. We met one of these in the last chapter: the public_methods method, which returns an array of all the method names available on the object. There is also instance_variables, which will pull out the names of any instance variables buried in the object. So, if you run this:

```
pp doc.instance_variables
```

You will see:

```
[:@title, :@author, :@content]
```

In all, Object bestows about fifty methods on its children.

Public, Private, and Protected

Like a lot of object oriented programming languages, Ruby lets you control the visibility of your methods. Methods can either be public—callable by any code anywhere, or private, or protected. Ruby methods are public by default, so up to now all of our Document methods have been public.

You can make your methods private by adding private before the method definition:

```
class Document
  # Most of the class omitted

  private  # Methods are private starting here

  def word_count
    return words.size
  end
end
```

Or by making them private after the fact:

```
class Document
  # Most of the class omitted
```

```
    def word_count
      return words.size
    end

    private :word_count
  end
```

Ruby's treatment of private methods is a bit idiosyncratic. The rule is that you cannot call a private method with an explicit object reference. So if `word_count` was indeed private, then this:

```
n = doc.word_count
```

Will throw an exception, since we tried to use an explicit object reference (`doc`) in the call. By restricting the way that private methods can be called, Ruby ensures that private methods can only be called from inside the class that defined them. Thus the call to `word_count` in the `print_word_count` method that follows will work:

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

  def word_count
    return words.size
  end

  private :word_count

  # This method works because self is the right thing,
  # the document instance, when you call it.
  def print_word_count
    n = word_count
    puts "The number of words is #{word_count}"
  end
end
```

Note that in Ruby, private methods are callable from subclasses. Think about it: You don't need an explicit object reference to call a superclass method from a subclass. Thus, this is a perfectly functional bit of code:

```
# RomanceNovel is a subclass of Document,
# which is a subclass of Object

class RomanceNovel < Document
  def number_of_steamy_words
    word_count / 4     # Works: self is a Document instance!
  end
end
```

The rules for protected methods are looser and a bit more complex: Any instance of a class can call a protected method on any other instance of the class. Thus, if we made `word_count` protected, any instance of `Document` could call `word_count` on any other instance of `Document`, including instances of subclasses like `RomanceNovel`.

There are a couple of things to keep in mind about private and protected methods in Ruby. The first is that while Ruby's system of controlling method visibility is perfectly respectable, it doesn't get a lot of use. For example, if you look at the Ruby standard library you will find nearly 200,000 lines of code. In that huge pile of software, `private` appears just over 1000 times and `protected` only about fifty times. Second, remember that just because the rules say that you can't call some private or protected method, well, you can still call it. Among the methods that every object inherits from `Object` is `send`. If you supply `send` with the name of a method and any arguments the method might need, `send` will call the method, visibility be damned:

```
n = doc.send( :word_count )
```

The Ruby philosophy is that the programmer is in charge. If you want to declare some method private, fine. Later, if someone, perhaps you, wants to violate that privacy, fine again. You are in charge and presumably you know what you are doing.

In the Wild

One thing that takes a while to sink in when learning Ruby is just how central the idea of methods and method calls are to the fundamental infrastructure of the language. Certainly there are lots of of things that go on in Ruby that are *not* method calls. Assigning a value to a local variable is not a method call. An `if` statement is not a method call. Neither is a `while` loop. Once, however, you get beyond the lowest layer of basic infrastructure, a surprising number of things you might consider to be part of the language are, in fact, just calls to methods. For example, you can't get much more fundamental than method visibility. Nevertheless, `private`, the magic word that makes Ruby methods private, is not really magic at all: It's just a method, albeit one that is implemented inside the Ruby interpreter. The same is true of `private`'s buddies, `public` and `protected`.

The Ruby interpreter also defines another method, one that takes the name of a file, reads the contents of the file, and executes those contents as Ruby code. This method maintains a list of the files it has already processed and won't re-execute a file that it has already seen. We call that *method* `require`:[6]

```
require 'date'    # A Call to a method
```

Another set of methods that play the part of language keywords are the `attr_accessor` family:

```
class Person
  attr_accessor :salary  # A method call
  attr_reader :name      # Another method call
  attr_writer :password  # And another
end
```

Not only are `attr_accessor` and friends just methods, but they are within the reach of mortal programmers. In Chapter 26 we are going to build our own versions of `attr_reader` and `attr_writer`. Stay tuned!

Staying Out of Trouble

The good news is that virtually all Ruby objects[7] inherit about fifty methods from the `Object` class, methods that allow you to do all sorts of useful things. The bad news is that those inherited methods also represent about fifty opportunities to have a name collision. Take this innocent-looking addition to the `Document` class:

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

6. The `require` method can also handle loading native libraries.

7. Except for those `BasicObject` oddballs.

```
  # Send this document to off via email
  def send( recipient )
    # Do some interesting SMTP stuff...
  end
end
```

Innocent, except that we have just overridden the `Object` supplied `send` method. Know your `Object` class, if for no other reason than to stay out of its way.

Another way to go wrong is to intentionally override a method from `Object`—and to get it wrong. Recall that earlier in the chapter we overrode the `to_s` method so that it would produce a better description of our document. Make a mistake in the `to_s` method, perhaps like this:

```
class Document
  # Mostly omitted...

  def to_s
    "#{title} by #{aothor}"    # oops!
  end
end
```

And you will no longer be able to print `Document` instances via `puts`.

Finally, programmers new to the language will sometimes forget how uniform the Ruby object model is and invent special cases where none actually exist, perhaps like this:

```
if the_object.nil?
  puts 'The object is nil'
elsif the_object.instance_of?( Numeric )
  puts 'The object is a number'
else
  puts "The object is an instance of #{the_object.class}"
end
```

When they could get away with the much simpler:

```
puts "The object is an instance of #{the_object.class}"
```

The single line of code here will work with virtually any object in your Ruby interpreter: Feed it a string and it will tell you that you have a `String`. Feed it a number and you might see that you have an instance of `Fixnum` or `Float`. Feed it `nil` and it will tell you that you have an instance[8] of `NilClass`. This example illustrates a more general and very pleasant side effect of Ruby's uniform object system: Since Ruby's `nil` is a real object, it is generally much less toxic than the equivalent constructs in other programming languages. When I'm coding in Ruby I don't have to be quite as paranoid that this object I have might just be `nil`. Remember, a great way to avoid broken code is to have less of it. The code that you never write will work forever.

Wrapping Up

If you remember anything from this chapter, remember this: Virtually everything in Ruby is an object, and virtually all of those objects inherit a basic set of methods from the `Object` class. In a very real way, the `Object` class is the glue that binds Ruby together and lends the language its simple elegance. Given how fundamental the `Object` class is to Ruby, it shouldn't come as much of a surprise that we aren't done with it. In chapters to come, we will look at how you can use the framework provided by the `Object` class to define your own operators, to create objects with their own ideas of equality, to rewire your inheritance tree, and to handle calls to methods that you haven't written.

For the moment, however, we will turn our attention to a different question. In this chapter we have focused on the things that all Ruby objects have in common: At their core, the dates and strings and documents floating around in our programs are all the same. But they are also all different: Outside of its `Object` core, a `Date` instance is very different from a `String` instance. The question for the next chapter is simple: How do you keep all of these different objects straight?

8. The only instance!