Chapter 1

Write Code That Looks Like Ruby

Some years ago I did a long stint working on a huge document management system. The interesting thing about this job was that it was a joint development project: Part of the system was developed by my group here in the United States and part was built by a team of engineers in Tokyo. I started out doing straight programming, but slowly my job changed—I found myself doing less and less programming and more and more translating. Whenever a Japanese–American meeting collided with the language barrier my phone would ring, and I would spend the rest of the afternoon explaining to Mr. Smith just exactly what Hosokawa-San was trying to say, and vice versa. What is remarkable is that my command of Japanese is practically nil and my Japanese colleagues actually spoke English fairly well. My special ability[1] was that I could understand the very correct, but very unidiomatic classroom English spoken by our Japanese friends as well as the slangy, no-holds-barred Americanisms of my U.S. coworkers.

You see the same kind of thing in programming languages. The parser for any given programming language will accept any valid program—that's what makes it a valid program—but every programming community quickly converges on a style, an accepted set of idioms. Knowing those idioms is as much a part of learning the language as knowing what the parser will accept. After all, programming is as much about communicating with your coworkers as writing code that will run.

1. I had developed this talent over numerous lunches, happy hours, and late-night bull sessions. Oh, how I do devote myself to the cause.

In this chapter we'll kick off our adventures in writing good Ruby with the very smallest idioms of the language. How do you format Ruby code? What are the rules that the Ruby community (not the parser) have adopted for the names of variables and methods and classes? As we tour the little things that make a Ruby program stylistically correct, we will glimpse at the thinking behind the Ruby programming style, thinking that goes to the heart of what makes Ruby such a lovely, eloquent programming language. Let's get started.

The Very Basic Basics

At its core, the Ruby style of programming is built on a couple of very simple ideas: The first is that code should be crystal clear—good code tells the reader exactly what it is trying to do. Great code shouts its intent. The second idea is related to the first: Since there is a limit to how much information you can keep in your head at any given moment, good code is not just clear, it is also concise. It's much easier to understand what a method or a class is doing if you can take it all in at a glance.

To see this in practice, take a look at this code:

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

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

  def words
    @content.split
  end

  def word_count
    words.size
  end
end
```

The `Document` class is nothing special technically—just a field for an author, one for a title, and one for the document content along with a few simple methods.[2] The thing to note about the code above is that it follows the Ruby indentation convention: In Ruby you indent your code with two spaces per level. This means that the first level of indentation gets two spaces, followed by four, then six, and then eight. The two space rule may sound a little arbitrary, but it is actually rooted in very mundane practicality: A couple of spaces is about the smallest amount of indentation that you can easily see, so using two spaces leaves you with the maximum amount of space to the right of the indentation for important stuff like actual code.

Note that the rule is to use two *spaces* per indent: The Ruby convention is to never use tabs to indent. Ever. Like the two space rule, the ban on tabs also has a very prosaic motivation: The trouble with tabs is that the exchange rate between tabs and spaces is about as stable as the price of pork belly futures. Depending on who and when you ask, a tab is worth either eight spaces, four spaces, two spaces, or, in the case of one of my more eccentric former colleagues, three. Mixing tabs and spaces without any agreement on the exchange rate between the two can result in code that is less than readable:

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

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

  def words
      @content.split
    end

    def word_count
    words.size
        end
end
```

2. But do take a good long look at the `Document` class because it's going to be with us for much of the book.

Life, not to mention the schedule, is too short to deal with problems like this,[3] so idiomatic Ruby should be serenely tab free.

Go Easy on the Comments

The mechanics of Ruby comments are simple enough: Anything following a # in the code is a comment.[4] The real questions regarding comments are when and how much? When is it a good idea to insert a comment into your code? And how much commenting is enough?

Good Ruby code should speak for itself, and most of the time you should let it do its own talking. If it's obvious how someone would use your method—if the class or program needs no explanation—then don't explain it. Above all, avoid boilerplate comments: Never put in a comment simply because you always put a comment there.

There are good reasons for adding comments to your code, the best being to explain how to use your software masterpiece. These kinds of "how to" comments should focus on exactly that: how to use the thing. Don't explain why you wrote it, the algorithm that it uses, or how you got it to run faster than fast. Just tell me how to use the thing and remember that examples are always welcome:

```
# Class that models a plain text document, complete with title
# and author:
#
# doc = Document.new( 'Hamlet', 'Shakespeare', 'To be or...' )
# puts doc.title
# puts doc.author
# puts doc.content
#
# Document instances know how to parse their content into words:
#
# puts doc.words
# puts doc.word_count
#
```

3. The "problem like this" in the example is that it was written with randomly mixed tabs and spaces, with each tab worth two spaces. Now expand the tabs to four spaces each and you have the formatting chaos that we see.

4. Ruby also supports multiline comments delimited by =begin and =end, but Ruby programmers tend to stick with the # style of comments most of the time.

```
class Document
  # class omitted...
end
```

This is not to say that you shouldn't have comments that explain some of the background of the code. Just keep the background separate from the instructions:

```
# Author: Russ Olsen
# Copyright 2010 Russ Olsen
#
# Document: An example Ruby class
```

Sometimes it's also wise to include a "how it works" explanation of particularly complicated bits of code. Again, keep this kind of explanation separate from the "how to":

```
  # Using ngram analysis, compute the probability
  # that this document and the one passed in were
  # written by the same person. This algorithm is
  # known to be valid for American English and will
  # probably work for British and Canadian English.
  #
  def same_author_probability( other_document )
    # Implementation left as an exercise for the reader...
  end
```

The occasional in-line comment can also help:

```
    return 0 if divisor == 0   # Avoid division by zero
```

Whatever you do, don't fall into the trap of sprinkling those "pesky younger sibling" comments throughout your code, comments that follow on the heels of each line, repeating everything it says:

```
    count += 1  # Add one to count
```

The danger in comments that explain how the code works is that they can easily slide off into the worst reason for adding comments: to make a badly written program

somewhat comprehensible. That voice you hear in your head, the one whispering that you need to add some comments, may just be your program crying out to be rewritten. Can you improve the class, method, and variable names so that the code itself tells you what it is doing? Can you rearrange things, perhaps by breaking up a long method or collapsing two classes together? Can you rethink the algorithm? Is there anything you can do to let the code speak for itself instead of needing subtitles?

Remember, good code is like a good joke: It needs no explanation.

Camels for Classes, Snakes Everywhere Else

Once we get past the relatively easy issues of indentation and comments, we come face to face with the question of names. Although this isn't the place to talk about the exact words you would use to describe your variables, classes and methods, this is the place to talk about how those words go together. It's really very simple: With a few notable exceptions, you should use `lowercase_words_separated_by_underscores`.[5] Almost everything in this context means methods, arguments, and variables, including instance variables:

```
def count_words_in( the_string )
  the_words = the_string.split
  the_words.size
end
```

Class names are an exception to the rule: Class names are camel case, so `Document` and `LegalDocument` are good but `Movie_script` is not. If all of this seems a bit doctrinaire for you, there is a place where you can exercise some creativity: constants. Ruby programmers seem divided about whether constants should be rendered in camel case like classes:

```
FurlongsPerFortnight = 0.0001663
```

Or all uppercase, punctuated by underscores:

```
ANTLERS_PER_MALE_MOOSE = 2
```

5. Because of its low, streamlined look, this naming style is known as "snake case."

My own preference—and the one you will see throughout this book—is the `ALL_UPPERCASE` flavor of constant.

Parentheses Are Optional but Are Occasionally Forbidden

Ruby tries hard not to require any syntax it can do without and a great example of this is its treatment of parentheses. When you define or call a method, you are free to add or omit the parentheses around the arguments. So if we were going to write a method to find a document, we might write it with the parentheses:

```
def find_document( title, author )
  # Body omitted...
end

# ...

find_document( 'Frankenstein', 'Shelley' )
```

Or leave them out:

```
def find_document title, author
  # Body omitted...
end

# ...

find_document 'Frankenstein', 'Shelley'
```

So, do you put them in or leave them out? With some notable exceptions, Ruby programmers generally vote *for* the parentheses: Although this isn't a hard and fast rule, most Ruby programmers surround the things that get passed into a method with parentheses, both in method definitions and calls. Somehow, having those parentheses there makes the code seem just a bit clearer.

As I say, this is not a rigid law, so Ruby programmers do tend to dispense with the parentheses when calling a method that is familiar, stands alone on its own line, and

whose arguments are few in number. Our old friend `puts` fits this description to a tee, and so we tend to leave the parentheses off of calls to `puts`:

```
puts 'Look Ma, no parentheses!'
```

The other main exception to the "vote yes for parentheses" rule is that we don't do empty argument lists. If you are defining—or calling—a method with no parameters, leave the parentheses off, so that it is:

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

And not:

```
def words()
  @content.split()
end
```

Finally, keep in mind that the conditions in control statements don't require parentheses—and we generally leave them off. So don't say this:

```
if ( words.size < 100 )
  puts 'The document is not very long.'
end
```

When you can say this:

```
if words.size < 100
  puts 'The document is not very long.'
end
```

Folding Up Those Lines

Although most Ruby code sticks to the "one statement per line" format, it is possible to cram several Ruby statements onto a single line: All you need to do is to insert a semicolon between the statements:

```
puts doc.title; puts doc.author
```

As I say, mostly we don't. There are a few exceptions to this rule. For example, if you are defining an empty, or perhaps a very, very simple class, you might fold the definition onto a single line:

```
class DocumentException < Exception; end
```

You might also do the same thing with a trivial method:

```
def method_to_be_overriden; end
```

Keep in mind that a little bit of this kind of thing goes a long way. The goal is code that is clear as well as concise. Nothing ruins readability like simply jamming a bunch of statements together because you can.

Folding Up Those Code Blocks

Ruby programs are full of code blocks, chunks of code delimited by either a pair of braces, like this:

```
10.times { |n| puts "The number is #{n}" }
```

Or, by the `do` and `end` keywords:

```
10.times do |n|
  puts "The number is #{n}"
  puts "Twice the number is #{n*2}"
end
```

The two forms of code block are essentially identical: Ruby doesn't really care which you use. Ruby programmers have, however, a simple rule for formatting of code blocks: If your block consists of a single statement, fold the whole statement into a single line and delimit the block with braces. Alternatively, if you have a multi-statement block, spread the block out over a number of lines, and use the `do/end` form.

Staying Out of Trouble

More than anything else, code that looks like Ruby looks *readable*. This means that although Ruby programmers generally follow the coding conventions that we have covered in this chapter, sometimes circumstances—and readability—call for the unconventional. Take the rule about folding up a one line code block, so that instead of this:

```
doc.words.each do |word|
  puts word
end
```

You would write this:

```
doc.words.each { |word| puts word }
```

The time to break this convention is when it would make your single line of code too long:

```
doc.words.each { |word| some_really_really_long_expression( ... with
lots of args ... ) }
```

Although coders differ about how long is too long,[6] at some point you're going to confront a block that might live on a single line, but shouldn't.

This kind of thinking should also go into the question of parentheses. There are times when, according to the "rules," you might omit the parentheses, but readability says that you should leave them in. For example, we have seen that `puts` generally goes sans parentheses:

```
puts doc.author
```

Another method that is frequently without parentheses is `instance_of?`, which tells you whether an object is an instance of some class:

```
doc.instance_of? Document
```

6. In fact, given the formatting limitations of this book, a good number of the blocks you'll see in this book will be multiline rather than single line, simply because the longer line will not fit on the page.

If, however, you assemble these two methods into a more complex expression, like this:

```
puts doc.instance_of? self.class.superclass.class
```

Then perhaps it is time for some parentheses:

```
puts doc.instance_of?( self.class.superclass.class )
```

Thus, the final code formatting rule is to always mix in a pinch of pragmatism.

In the Wild

Absolutely the best way to learn to *write* idiomatic Ruby code is to *read* idiomatic Ruby code. The Ruby standard library, the lump of Ruby code that came with your Ruby interpreter, is a great place to start. Just pick a class that interests you; perhaps you have always been fascinated by the `Set` class.[7] Find `set.rb` in your Ruby install and settle in for some interesting reading.

If you do go looking at `set.rb`, you will find, along with two-space indentation and well-formed variable names, that the file starts with some comments[8] explaining the background of the class:

```
# Copyright (c) 2002-2008 Akinori MUSHA <knu@iDaemons.org>
#
# Documentation by Akinori MUSHA and Gavin Sinclair.
#
# All rights reserved. You can redistribute and/or modify it
# under the same terms as Ruby.
```

This is followed by a quick explanation of what the class does:

```
# This library provides the Set class, which deals with a
# collection of unordered values with no duplicates.  It
# is a hybrid of Array's intuitive inter-operation facilities
```

7. Perhaps you should get out more often.

8. If you do look you will discover that I reformatted the comments a bit to make them fit on the page.

```
# and Hash's fast lookup.  If you need to keep values ordered,
# use the SortedSet class.
```

And then a few examples:

```
#   require 'set'
#   s1 = Set.new [1, 2]                   # -> #<Set: {1, 2}>
#   s2 = [1, 2].to_set                    # -> #<Set: {1, 2}>
#   s1 == s2                              # -> true
```

If you want to add useful comments to your own code, you could do worse than follow the model of `set.rb`.

If you look closely at the `Set` class you will see a couple of additional method name conventions at work. The first is that Ruby programmers will usually end the name of a method that answers a yes/no or true/false question with a question mark. So if you do peek into `Set` you will find the `include?` method as well as `superset?` and `empty?`. Don't be fooled by that exotic-looking question mark: It's just an ordinary part of the method name, not some special Ruby syntax. The same thing is true of exclamation points at the end of a method name: The Ruby rules say that `!` is a fine character with which to end a method name. In practice, Ruby programmers reserve `!` to adorn the names of methods that do something unexpected, or perhaps a bit dangerous. So the `Set` class has `flatten!` and `map!`, both of which change the class in place instead of returning a modified copy.

Although `set.rb` is a model of Ruby decorum, there are also notable spots where the code that comes built into Ruby breaks some of the Ruby conventions. Exhibit one: the `Float` method. Shock! Imagine a method name that begins with an uppercase letter. [Cue ominous music.]

Actually, there is some excuse for this momentous breach of manners: The `Float` method turns its argument—usually a string—into a floating point number.[9] Thus you can use `Float` as a kind of stand-in for the class name:

```
pi = Float('3.14159')
```

9. Note that the `Float('3.14159')` is not quite the same as `'3.14159'.to_f`. The Float method will throw an exception if you pass it bad input, while `to_f` will quietly return 0.

The best part of having rules is that they inevitably create rule breakers—and it's even better when the rule breakers are the authorities.

Wrapping Up

So there we have the very basics of the Ruby programming style. Shallow, space-based indentation. A definite set of rules for formatting names. Optional parentheses, mostly supplied. Comments that tell you how to use it, or how it works, or who wrote it, but in stingy moderation. Above all else, pragmatism: You cannot make readable code by blindly following some rules.

This is, however, just the beginning: In a very real sense, this whole book is devoted to Ruby conventions of one kind or other. In particular, in Chapters 10 and 18 we will return to the subject of method names, while the last chapter is all about breaking the rules.

For the moment, however, we're going to stick to the basics, so in the next chapter we'll look at the Ruby control structures and how they contribute to clear and concise code.