CHAPTER 8

Embrace Dynamic Typing

How? Why? These are the two questions that every new Ruby coder—or at least those emigrating from the more traditional programming languages—eventually gets around to asking. How can you possibly write reliable programs without some kind of static type checking? And why? Why would you even want to try? Figure out the answer to those two questions and you're on your way to becoming a seasoned Ruby programmer. In this chapter we will look at how dynamic typing allows you to build programs that are simultaneously compact, flexible, and readable. Unfortunately, nothing comes for free, so we will also look at the downsides of dynamic typing and at how the wise Ruby programmer works hard to make sure the good outweighs the bad.

This is a lot for one chapter, so let's get started.

Shorter Programs, But Not the Way You Think

One of the oft-repeated advantages of dynamic typing is that it allows you to write more compact code. For example, our `Document` class would certainly be longer if we needed to state—and possibly repeat here and there—that `@author`, `@title`, and `@content` are all strings and that the `words` method returns an array. What is not quite so obvious is that the simple "every declaration you leave out is one bit less code" is just the down payment on the code you save with dynamic typing. Much more significant savings comes from the classes, modules, and methods that you never write at all.

To see what I mean, let's imagine that one of your users has a large number of documents stored in files. This user would like to have a class that looks just like a

`Document`,[1] but that will delay reading the contents of the file until the last possible moment: In short, the user wants a lazy document. You think about this new requirement for a bit and come up with the following: First you build an abstract class that will serve as the superclass for both the regular and lazy flavors of documents:

```
class BaseDocument

  def title
    raise "Not Implemented"
  end

  def title=
    raise "Not Implemented"
  end

  def author
    raise "Not Implemented"
  end

  def author=
    raise "Not Implemented"
  end

  def content
    raise "Not Implemented"
  end

  # And so on for the content=
  # words and word_count methods...

end
```

Then you recast `Document` as a subclass of `BaseDocument`:

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

1. Again, to keep things simple we are going to start over here with the very minimal functionality of the original Document class of Chapter 1.

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

    def words
      @content.split
    end

    def word_count
      words.size
    end
  end
```

Finally, you write the `LazyDocument` class, which is also a subclass of `BaseDocument`:

```
  class LazyDocument < BaseDocument

    attr_writer :title, :author, :content

    def initialize( path )
      @path = path
      @document_read = false
    end

    def read_document
      return if @document_read
      File.open( @path ) do | f |
        @title = f.readline.chomp
        @author = f.readline.chomp
        @content = f.read
      end
      @document_read = true
    end

    def title
      read_document
      @title
    end
    def title=( new_title )
      read_document
      @title = new_title
    end

    # And so on...
  end
```

The `LazyDocument` class is a typical example of the "leave it to the last minute" technique: It looks like a regular document but doesn't really read anything from the file until it absolutely has to. To keep things simple, `LazyDocument` just assumes that its file will contain the title and author of the document on the first couple of lines, followed by the actual text of the document.

With the classes above, you can now do nice, polymorphic things with instances of `Document` and `LazyDocument`. For example, if you have a reference to one or the other kind of document and are not sure which:

```
doc = get_some_kind_of_document
```

You can still call all of the usual document methods:

```
puts "Title: #{doc.title}"
puts "Author: #{doc.author}"
puts "Content: #{doc.content}"
```

In a technical sense, this combination of `BaseDocument`, `Document`, and `LazyDocument` do work. They fail, however, as good Ruby coding. The problem isn't with the `LazyDocument` class or the `Document` class. The problem lies with `BaseDocument`: It does nothing. Even worse, `BaseDocument` takes more than 30 lines to do nothing. `BaseDocument` only exists as a misguided effort to provide a common interface for the various flavors of documents. The effort is misguided because Ruby does not judge an object by its class hierarchy.

Take another look at the last code example: Nowhere do we say that the variable `doc` needs to be of any particular class. Instead of looking at an object's type to decide whether it is the correct object, Ruby simply assumes that if an object has the right methods, then it is the right kind of object. This philosophy, sometimes called **duck**

typing,[2] means that you can completely dispense with the `BaseDocument` class and redo the two document classes as a couple of completely independent propositions:

```
class Document
  # Body of the class unchanged...
end

class LazyDocument
  # Body of the class unchanged...
end
```

Any code that used the old related versions of `Document` and `LazyDocument` will still work with the new unrelated classes. After all, both classes support the same set of methods and that's what counts.

There are two lessons you can take away from our `BaseDocument` excursion. The first is that the real compactness payoff of dynamic typing comes not from leaving out a few `int` and `string` declarations; it comes instead from all of the `BaseDocument` style abstract classes that you never write, from the interfaces that you never create, from the casts and derived types that are simply irrelevant. The second lesson is that the payoff is not automatic. If you continue to write static type style base classes, your code will continue to be much bulkier than it might be.

Extreme Decoupling

Compact code is a great thing, but compact code is by no means the only advantage of dynamic typing. There is also the free and easy flexibility that flows from writing code sans type declarations. For example, let's imagine that the editorial department of your company also has an enhancement request. It seems that the folks over at editorial are putting in a more formal system to keep track of authors and publications. In particular, they have invented a couple of new classes:

```
class Title
  attr_reader :long_name, :short_name
  attr_reader :isbn
```

2. As in, "If it walks like a duck and quacks like a duck, then it must be a duck."

```
    def initialize(long_name, short_name, isbn)
      @long_name = long_name
      @short_name = short_name
      @isbn = isbn
    end
  end

  class Author
    attr_reader :first_name, :last_name

    def initialize( first_name, last_name )
      @first_name = first_name
      @last_name = last_name
    end
  end
```

The editorial department would like you to change the `Document` class so that they can use `Title` and `Author` instances instead of strings as the `@title` and `@author` values in `Document` instances, like this:

```
two_cities = Title.new( 'A Tale Of Two Cities',
                        '2 Cities', '0-999-99999-9' )
dickens = Author.new( 'Charles', 'Dickens' )
doc = Document.new( two_cities, dickens, 'It was the best...' )
```

Being a nice person and a consummate professional you immediately agree to undertake this task. And then you do nothing. Absolutely nothing. You do nothing because the `Document` class already works with `Title` and `Author` instances. There are no interfaces to extract, no declarations to change, no class hierarchies to adjust, nothing. It just works.

It works because Ruby's dynamic typing means that you don't declare the classes of variables and parameters. That means that your classes are not frozen together in a rigid network of type relationships. In Ruby, any two classes that *can* work together *will* work together. Flexibility is a huge advantage when it comes to constructing programs. In our example, the `Document` class does not really do anything with `@title` and `@author` other than carry them around; the `Document` class therefore has absolutely no opinion as to what the class of these objects should be.

Even if Document did make some demands on @title and @author, perhaps like this:

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

  def description
    "#{@title.long_name} by #{@author.last_name}"
  end
end
```

Then we will have increased the coupling between Document and the @author and @title objects just a bit. With the addition of the description method, Document now expects that @title will have a method called long_name and @author will have a last_name method. But the bump in coupling is as small as it can be. Document will, for example, accept any object that has a long_name method for @title.

Taking advantage of the loose coupling offered by dynamic typing is easy: As you can see from this last example, it is right there for you—unless you go out of your way to mess it up. Programmers new to Ruby will sometimes try to cope with the loss of static typing by adding type-checking code to their methods:

```
  def initialize( title, author, content )
    raise "title isn't a String" unless title.kind_of? String
    raise "author isn't a String" unless author.kind_of? String
    raise "content isn't a String" unless content.kind_of? String
    @title = title
    @author = author
    @content = content
  end
```

This kind of pseudo-static type checking combines all the disadvantages of the two camps: It destroys the wonderful loose coupling of dynamic typing. It also bloats the code while doing little to improve reliability. Don't do this.

This last example illustrates another, more subtle advantage to dynamic typing. Programming is a complex business. Writing a tricky bit of code is like that old circus act where the performer keeps an improbably large number of plates spinning atop vertical sticks, except that here it's the details of your problem that are spinning and it's all happening in your head. When you are coding, anything that reduces the number of revolving mental plates is a win. From this perspective, a typing system that you can sum up in a short phrase, "The method is either there or it is not," has some definite appeal. If the problem is complexity, the solution might just be simplicity.

Required Ceremony Versus Programmer-Driven Clarity

One thing that variable declarations do add to code is a modicum of documentation. Take the `initialize` method of our `Document` class:

```
def initialize( title, author, content )
```

Considerations of code flexibility and compactness aside, there is simply no arguing with the fact that a few type declarations:

```
# Pseudo-Ruby! Don't try this at home!

def initialize( String title, String author, String content )
```

Would make it easier to figure out how to create a `Document` instance. The flip side of this argument is that not all methods benefit—in a documentation sense—from type declarations. Take this hypothetical `Document` method:

```
def is_longer_than?( n )
  @content.length > n
end
```

Even without type declarations, most coders would have no trouble deducing that `is_longer_than?` takes a number and returns a boolean. Unfortunately, when type declarations are required, you need put them in your code whether they make your code more readable or not—that's why they call it *required*. Required type declarations inevitably become a ceremonial part of your code, motions you need to go through just to get your program to work. In contrast, making up for the lost documentation value of declarations in Ruby is easy: You write code that is painfully, blazingly obvious. Start by using nice, full words for class, variable, and method names:

```
def is_longer_than?( number_of_characters )
  @content.length > number_of_characters
end
```

If that doesn't help, you can go all the way and throw in some comments:

```
# Given a number, which needs to be an instance of Numeric,
# return true if the number of characters in the document
# exceeds the number.
def is_longer_than?( number_of_characters )
  @content.length > number_of_characters
end
```

With dynamic typing, it's the programmer who gets to pick the right level of documentation, not the rules of the language. If you are writing simple, obvious code, you can be very minimalistic. Alternatively, if you are building something complex, you can be more elaborate. Dynamic typing allows you to document your code to exactly the level you think is best. It's your job to do the thinking.

Staying Out of Trouble

Engineering is all about trade-offs. Just about every engineering decision involves getting something, but at a price, and there is a price to be paid for dynamic typing. Undeniably, dynamic typing opens us up to dangers that don't exist in statically typed languages. What if we missed the memo saying that the `Document` class now expects the `@title` to have a `long_name` method? We might just end up here:

```
NoMethodError: undefined method `long_name' for "TwoCities":String
```

This is the nightmare scenario that virtually everyone who comes to Ruby from a statically typed language background worries about. You think you have one thing, perhaps an instance of `Author`, when in fact you actually have a reference to a `String` or a `Time` or an `Employee` and you don't even know it. There is just no getting around the fact that this kind of thing can happen in Ruby code.

What's a Ruby programmer to do? My first bit of advice is to simply relax. The experience that has accumulated over the past half century of dynamic language use is that horrible typing disasters are just not all that common. They are, in fact, downright rare in any carefully written program. The best way to avoid mixing your types, like metaphors, is to write the clearest, most concise code you can, which explains why Ruby programmers place such a high premium on (wait for it!) clear and concise code. If it's easy to see what's going on, you will make fewer mistakes.

Fewer mistakes, but not zero mistakes. Inevitably you are going to experience a type-related bug now and then. Unsurprisingly, you are also going to have non-type-related bugs as well. The Ruby answer to both kinds of bugs is to write automated tests, lots and lots of automated tests. In fact, automated tests are such a core part of writing good Ruby code that the next chapter is devoted to them.

You should also keep in mind that there is a difference between concise and cryptic. Ruby allows you to write wonderfully expressive code, code that gets things done with a minimum of noise. Ruby also allows you to write stuff like this:

```
class Doc
  attr_accessor :ttl, :au, :c

  def initialize(ttl, au, c)
    @ttl = ttl; @au = au; @c = c
  end

  def wds;  @c.split; end
end
```

In any language, this kind of "damn the reader" terseness, with its cryptic variable and method names, is bad. In Ruby it's a complete disaster. Since bad Ruby code does not have the last resort crutch of type declarations to lean on, bad Ruby code can be very bad indeed. The only solution is to not write bad Ruby code. Try to make your code speak to the human reader as much as it speaks to the Ruby interpreter. It comes down to this: Ruby is a language for grown-ups; it gives you the tools for writing clear and concise code. It's up to you to use them.

In the Wild

A good example of the Ruby typing philosophy of "if the method is there, it is the right object" is as close as your nearest file and string. Every Ruby programmer knows that if you want to open a Ruby file, you do something like this:[3]

3. Actually, most Ruby programmers would call `File.open` with a block, but that is beside the point here.

```
open_file = File.open( '/etc/passwd' )
```

Sometimes, however, you would like to be able to read from a string in the same way that you read from a file, so we have `StringIO`:

```
require 'stringio'
open_string = StringIO.new( "So say we all!\nSo say we all!\n" )
```

The idea is that you can use `open_file` and `open_string` interchangeably: Call `readchar` on either and you will get the next character, either from the file or the string. Call `readline` and you will get the next line. Calling `open_file.seek(0)` will put you back at the beginning of the file while `open_string.seek(0)` will put you at the beginning of the string.

Surprisingly, the `File` and `StringIO` classes are completely unrelated. The earliest common ancestor of these two classes is `Object`! Apparently reading and writing files and strings is different enough that the authors of `StringIO` (which was presumably written after `File`) decided that there was nothing to gain—in terms of implementation—from inheriting from `File`, so they simply went their own way. This is fairly typical of Ruby code, where subclassing is driven more from practical considerations—"Do I get any free implementation from inheriting from this class?"—than a requirement to make the types match up.

You can find another interesting example of the "don't artificially couple your classes together" thinking in the source code for the `Set` class, which we looked at briefly in Chapter 3. It turns out that you can initialize a `Set` instance with an array, like this:

```
five_even = [ 2, 4, 6, 8, 10 ]
five_even_set = Set.new( five_even )
```

In older versions of `Set`, the code that inserted the initial values into the new `Set` instance looked like this:[4]

```
enum.is_a?(Enumerable) or raise ArgumentError, "not enumerable"
enum.each { |o| add(o) }
```

4. I did take some liberties with this code to make it fit within the formatting restrictions of this book.

These early versions of `Set` first checked to see if the variable `enum`, which held the initial members of the set, was an instance of `Enumerable`—arrays and many other Ruby collections are instances of `Enumerable`—and raised an exception if it wasn't. The trouble with this approach is that the requirement that `enum` be `Enumerable` is completely artificial. In the spirit of dynamic typing, all that `Set` should really care about is that `enum` has an `each` method that will iterate through all of the elements. Apparently the maintainers of the `Set` class agree, because the `Enumerable` check has disappeared from the current version of `set.rb`.

Wrapping Up

So how do you take advantage of dynamic typing? First, don't create more infrastructure than you really need. Keep in mind that Ruby classes don't need to be related by inheritance to share a common interface; they only need to support the same methods. Don't obscure your code with pointless checks to see whether *this* really is an instance of *that*. Do take advantage of the terseness provided by dynamic typing to write code that simply gets the job done with as little fuss as possible—but also keep in mind that someone (possibly you!) will need to read and understand the code in the future.

Above all, write tests. . . .