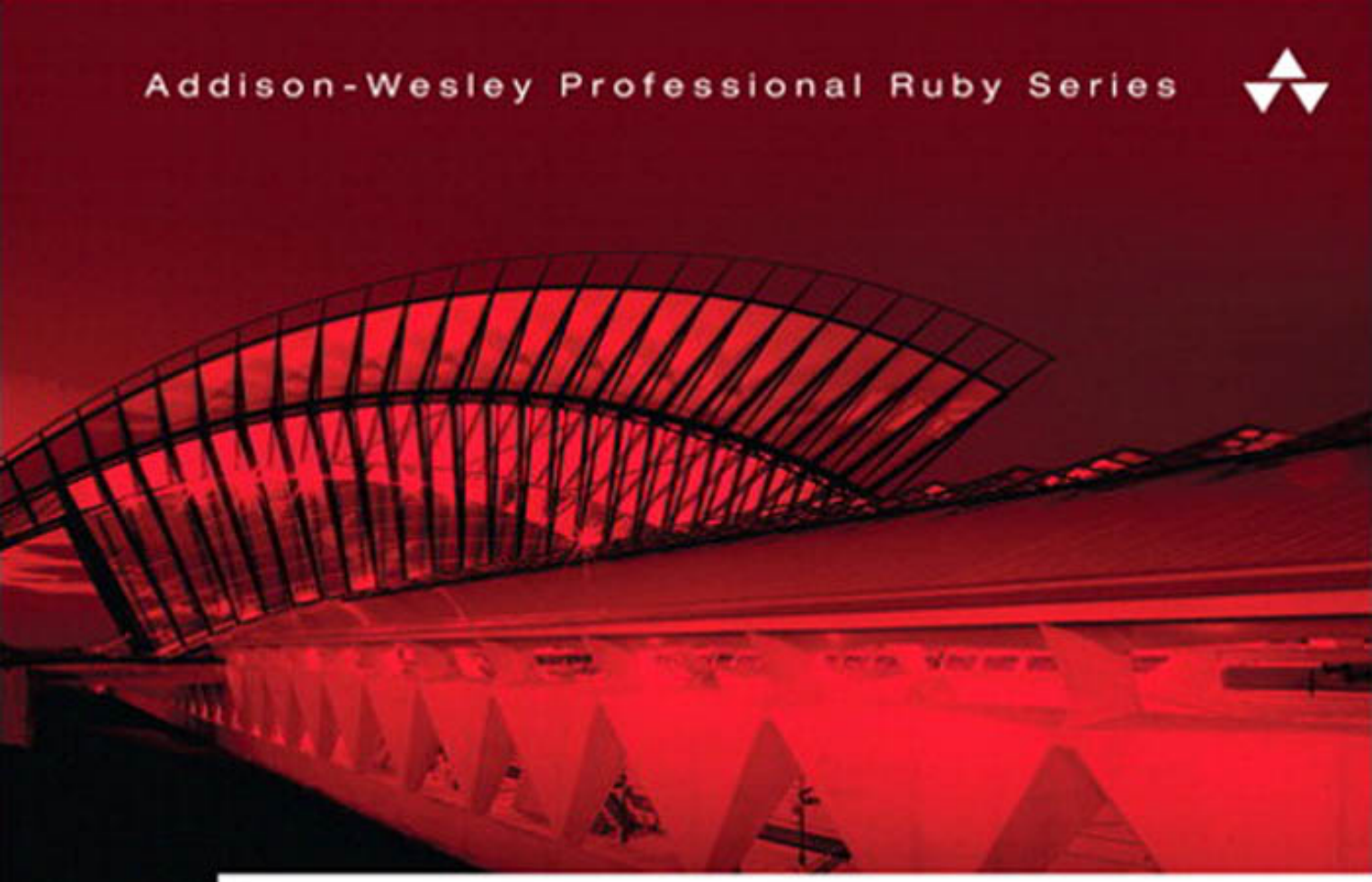
Addison-Wesley Professional Ruby Series



ELOQUENT RUBY

Foreword by **Obie Fernandez,** *Series Editor*

RUSS OLSEN

Praise for *Eloquent Ruby*

"Reading *Eloquent Ruby* is like programming in Ruby itself: fun, surprisingly deep, and you'll find yourself wishing it was always done this way. Wherever you are in your Ruby experience from novice to Rails developer, this book is a must read."

—Ethan Roberts
Owner, Monkey Mind LLC

"*Eloquent Ruby* lives up to its name. It's a smooth introduction to Ruby that's both well organized and enjoyable to read, as it covers all the essential topics in the right order. This is the book I wish I'd learned Ruby from."

—James Kebinger
Senior Software Engineer, PatientsLikeMe
www.monkeyatlarge.com

"Ruby's syntactic and logical aesthetics represent the pinnacle for elegance and beauty in the ALGOL family of programming languages. *Eloquent Ruby* is the perfect book to highlight this masterful language and Russ's blend of wit and wisdom is certain to entertain and inform."

—Michael Fogus
Contributor to the Clojure programming language and author of *The Joy of Clojure*

This page intentionally left blank

Eloquent Ruby

Eloquent Ruby

Russ Olsen

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Olsen, Russ.
Eloquent Ruby / Russ Olsen.
p. cm.
Includes index.
ISBN-13: 978-0-321-58410-6 (pbk. : alk. paper)
ISBN-10: 0-321-58410-4 (pbk. : alk. paper)
1. Ruby (Computer program language) I. Title.
QA76.73.R83O47 2011
005.13'3—dc22

2010048388

ISBN-13: 978-0-321-58410-6
ISBN-10: 0-321-58410-4

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
Second printing, July 2011

To My Dad
Charles J. Olsen
Who never had a chance to write a book of his own,
which is a shame because it would have been
hilarious

This page intentionally left blank

Contents

This page intentionally left blank

Foreword

Do you know why experienced Ruby programmers tend to reach for basic collections and hashes while programmers from other languages go for more specialized classes? Do you know the difference between strip, chop, and chomp, and why there are three such similar methods when apparently one might suffice? (Not to mention lstrip and rstrip!) Do you know the downsides of dynamic typing? Do you know why the differences between strings and symbols get so blurry, even to experienced Ruby developers? How about metaprogramming? What the heck is an eigenclass? How about protected methods? Do you know what they're really about? Really? Are you sure?

Russ knows all that stuff and more. And if books are like babies, then Russ is that experienced mom who pops out her second child after a couple of hours of labor and is back at work a week later in her pre-pregnancy clothes as if nothing out of the ordinary happened. You know: the one all the other moms talk about in hushed tones of disbelief and reverence. That's the way my series authors discuss Russ.

Not that there's anything small or insignificant about Russ' bouncing new baby . . . eh, I mean book. On the contrary, weighing in at just over 400 pages, this tome is slightly larger than its older sibling *Design Patterns in Ruby*. The family resemblance is crystal clear: Russ is first and foremost your friend. His approachable writing style makes even the driest Ruby language topics engaging and funny. Like the way that symbols remind Russ "of the eyes peering out from the tilted head of a confused but friendly dog."

Truth is, we need this kind of book now more than ever. Ruby has hit the mainstream with the force of a Hulk Smash, and the masses are paddling along well-known routes without full (heck, sometimes any) understanding of what makes their favorite

frameworks and library APIs so vibrant and navigable. So for those not content with the basics, those who want to go beyond shallow understanding, this book goes deep. It helps readers achieve true mastery of Ruby, a programming language with some of the deepest, darkest pools of nuance and texture of all the major languages of modern times.

I know you're going to enjoy this book, just like I did. And if you do, please join me in encouraging Russ to get knocked up again soon.

—Obie Fernandez, Professional Ruby Series Editor

Preface

I've taught a fair number of Ruby classes over the years, but one particular class stands out in my mind. Class was over, and as I was going out the door one of my students, an experienced Java programmer, stopped me and voiced a complaint that I have heard many times since. He said that the hardest part of learning Ruby wasn't the syntax or the dynamic typing. Oh, he could write perfectly correct Ruby, sans semicolons and variable declarations. The problem was that something was missing. He constantly found himself falling back into his same old Java habits. Somehow his Ruby code always ended up looking much like what he would have written in Java. My answer to him was not to worry, you haven't missed anything—you just aren't done learning Ruby.

What does it mean to learn a new programming language? Clearly, like my frustrated student, you need to understand the basic rules of the grammar. To learn Ruby you need to be aware that a new line usually starts a new statement, that a class definition starts with the word `class`, and that variable names start with a lowercase letter—unless they start with an `@`. But you can't really stop there. Again, like my erstwhile student you will also need to know what all of that code does. You'll need to know that those statements are really expressions (since they all return a value) and that all of those classes starting with the `class` keyword can change over time. And you'll need to know why those `@variables` are different from the plain vanilla variables.

But the punch line is that even after you master all of this, you are still not quite there. It turns out that computer languages share something fundamental with our everyday order-a-pizza human tongues: Both kinds of languages are embedded in a culture, a way of thinking about the world, an approach to solving problems. A formal

understanding of the mechanics of Ruby isn't the same as really looking at the programming world through Ruby-colored glasses. You need to absorb the cultural part of Ruby, to see how real Rubyists use the language to solve problems.

This is a book about making that final leap, about absorbing the Ruby programming culture, about becoming truly fluent in Ruby. The good news is that for most people the final step is the best part of learning Ruby—a series of "Ah ha!" moments—as it suddenly becomes clear why those funny symbol things exist, why classes are never final, and how this wonderful language works so hard to just stay out of your way.

Who Is This Book For?

This book is for you if you have a basic understanding of Ruby but feel that you haven't quite gotten your arms around the language. If you find yourself wondering what anyone could possibly do with all those odd language features that seem so important to Ruby, keep reading.

This book is also for you if you are a programmer with experience in other object oriented languages, perhaps Java or C# or Python, and you want to see what this Ruby thing is all about. While I'm not going to explain the basic details of Ruby in this book, the basics of Ruby are really very basic indeed. So, if your learning style involves simply jumping into the deep end, welcome to the pool.

How Is This Book Organized?

Mostly, this book works from small to large. We will start with the most tactical questions and work our way up to the grand strategy behind pulling whole Ruby projects together. Thus the first few chapters will concentrate on one-statement, one-method, one-test, and one-bug–sized issues:

- How do you write code that actually looks like Ruby?
- Why does Ruby have such an outsized collection of control structures?
- Why do Ruby programmers use so many hashes and arrays in their code?
- How do I get the most out of Ruby's very powerful strings and regular expressions?
- What are those symbol things, and what do you do with them?

- Is everything in Ruby really an object?
- How do I take advantage of dynamic typing?
- How can I make sure that my code actually works?

From there we will move on to the bigger questions of building methods and classes:

- Why are Ruby classes so full of tiny little methods?
- Why would you overload an operator? And, more importantly, why would you not?
- Do I really need to care about object equality?
- What good is a module?
- Can I really assign a method to an individual object? And what does that have to do with class methods?
- How do I hang some data on a class?
- How do you use blocks to good effect?
- Why would you ever call a method that doesn't actually exist?
- Can I really get notified when a class gets created? Why would I do that?
- Can I really modify classes on the fly? Why would I do that?
- Can I really write code that writes code? Why would I do that?

Finally, we will look at some of the techniques you can use to pull your programming project together into a unified whole:

- Would you really build a whole language simply to solve an ordinary programming problem?
- How do I make it easy for others to use my work?
- How does my Ruby implementation work?
- Where do I go from here?

About the Code Examples

The trouble with writing books about programming is that all the interesting stuff is in a constant state of flux. This is, after all, what makes it interesting. Certainly Ruby is something of a moving target these days: Although the great bulk of the Ruby code base was written for Ruby 1.8.X, version 1.9 has been out for some time and is clearly the future. In the pages that follow I have tried to split the coding difference by writing all of the examples in the 1.9 dialect,[1] taking care to note where Ruby 1.8 would be different. The good news is that there aren't all that many differences.

I have also consistently used the traditional `pp` command to print out more complex objects. However, to keep from driving everyone[2] crazy, I'm not going to endlessly repeat the `require 'pp'` call needed to make `pp` work. Just assume it is there at the top of each example.

1. Specifically, the examples all use Ruby-1.9.1-p430.

2. Especially me!

Acknowledgments

Sometimes I love to write and other times it's like squeezing out that last bit of toothpaste—from the point of view of the tube. At those times the constant support of my friends and family made the difference between a finished book and a smashed computer. In return I would like to say thanks, starting with my lovely wife Karen and my noble son Jackson for their constant support, and for putting up with me when that last sentence would just not settle down. Thanks especially to Karen for sneaking into my office in the middle of the night to remove the extraneous of's and the's from the manuscript.

Thanks to my good friend Bob Kiel for his constant encouragement. Couldn't have done it without you, Bob.

Thanks, too, to Eileen Cross for simply being there for me for all these years.

Thanks to the fine folks at FGM, especially Scott Gessay, Mike Fortier, Mike Morehouse, and Kirk Maskalenko. It really is a great place to work. Also thanks to George Croghan for continuing to speak to me even after I had used the parental voice of death on him.

Thanks to Chris Bailey for keeping me from taking a match to the whole project.

I also owe some serious gratitude to Gene, Laura, and Derek Stokes for their company and cheer as well as occasionally providing me with a quiet place to think and write: I've spent many a happy hour toiling away at the kitchen table of their beach house. I'd especially like to thank Gene for his rocket fuel martinis. I have only myself to blame if Gene's concoctions occasionally enhanced the happiness of the hour at the expense of the toiling. And thanks to Laura for injecting just the right level of zaniness into my life.

Special thanks to Scott Downie (the brightest intern who ever fetched coffee) for introducing me to the TV series *Firefly* and thereby getting me through the dark days of Chapters 15 and 16.[1]

Thanks to everyone behind the Northern Virginia Ruby Users' Group, RubyNation, and the National Capital Area Clojure User Group for their encouragement. Through their efforts hundreds of gallons of beer have found a decent home.

Thanks to everyone who reviewed the early versions of this book, including Rob Sanheim, James Kebinger, and Ethan Roberts.

Special thanks for Beth Gutierrez for providing her unique perspective on the manuscript.

Thanks to Carl Fyffe for helping me find a way out of the dark days of Chapters 15 and 16.

Thanks to Mike Abner and the aforementioned Carl for helping me to settle on a title.

Thanks also to Steve Ingram for starting the e-mail discussion that eventually gave birth to Chapter 6.

Thanks to my friend Diana Greenberg for her constant support, and for not buying a copy of this book before I can give her one.

Special thanks to Diane Freed. If you can imagine trying to correct a manuscript full of technical terms, tortured syntax, and typos (I can't), you have an idea of the job of a copy editor, a job that Diane performed with real finesse.

Thanks also to Rob and Denise Cross for putting up with me over a long Thanksgiving weekend as I went through my end of the copyediting of this book.

Thanks to Raina Chrobak of Addison-Wesley for her help and patience.

Finally special thanks to my editor Chris Guzikowski for putting up with the delays caused by the dark days of Chapters 15 and 16.

P.S. Thanks to Peter Cooper, Sonia Hamilton, John G. Norman, and Bodo Tasche for suggesting corrections to the first printing.

1. Well, originally they were Chapters 11, 12, and 13, and then they became Chapter 10 before settling down as 15 and 16. Now you know why those days were so dark.

About the Author

Russ Olsen's career spans three decades, during which he has written everything from graphics device drivers to document management applications. These days, Russ diligently codes away at GIS systems, network security, and process automation solutions. Otherwise, Russ spends a lot of his free time writing and talking about programming, especially Ruby and Clojure.

Russ' first book is the highly regarded *Design Patterns in Ruby* (Addison-Wesley, 2008). Russ is also the lurking presence behind the Technology As If People Mattered blog (www.russolsen.com). Russ' technical pontifications have been translated into six languages, and Russ is a frequent speaker at technical conferences.

Russ lives in the Washington, D.C., area with his lovely wife, Karen, and noble son, Jackson, both of whom are smarter than he is.

This page intentionally left blank

CHAPTER 6

Use Symbols to Stand for Something

I have to admit that I tend to be a bit anthropomorphic about the technologies I work with. I just can't help but think of all those complex piles of software as somehow alive, each with its own personality—sometimes friendly, sometimes not. Early in my career I imagined FORTRAN as a grouchy old camel—capable of carrying a huge load, but fairly ugly and not a creature you would want to turn your back on. Later on I had this mental image of the -> operator in the C programming language (it dereferences pointers) as an arrow in flight: also very powerful, also nothing to mess with. These days, the colon that precedes every Ruby symbol always makes me think of the eyes peering out from the tilted head of a confused but friendly dog. The key word here is confused—symbols probably have the dubious distinction of being the one bit of syntax that perplexes the greatest number of new Ruby programmers.

In this chapter I am going to try to stamp out all of that confusion and show symbols for what they really are: very simple, useful programming language constructs that are a key part of the Ruby programming style. So let's get started and see why symbols are such handy little mutts to have around.

The Two Faces of Strings

Sometimes a good way to explain a troublesome topic is to engage in a little creative fiction. You start out with an oversimplified explanation and, once that has sunk in a

bit, you work your way from there back to the real world. In this spirit, let's start our exploration of symbols with a slight simplification: Symbols are really just strings. This is not as far fetched as it sounds: Think about the string `"dog"` and its closest symbolic cousin, `:dog`. The thing that hits you in the face about these two objects is that they both are essentially three characters: a "d", an "o", and a "g".

Strings and symbols are also reasonably interchangeable in real life code: Take this familiar example of some ActiveRecord code, which finds all of the records in the books table:[1]

```
book = Book.find(:all)
```

The argument to the `find` method is simply a flag, there to tell `find` that we want all of the records in the `books` table—not just the first record, not just the last record, but all of them. The actual value that we pass into `Book.find` doesn't really matter very much. We might imagine that if we had the time and motivation, we could go into the guts of ActiveRecord and rewrite the code so that we could use a string to signal that we wanted *all* the books:

```
book = Book.find('all')
```

So there is my simplified explanation of symbols: Other than the fact that typing `:all` requires one less keystroke than typing `'all'`, there is not really a lot to distinguish a symbol from a string. So why does Ruby give us both?

Not Quite a String

The answer is that we tend to use strings of characters in our code for two rather different purposes: The first, and most obvious, use for strings is to hold some data that we are processing. Read in those `Book` objects from the database and you will very likely have your hands full of string data, things like the title of the book, the author's name, and the actual text.

The second way that we use strings of characters is to represent things in our programs, things like wanting to find *all* of the records in a table. The key thing about

1. If you are not familiar with ActiveRecord, don't worry. In ActiveRecord there is a class for each database table. In our example we have the (unseen) `Book` class that knows about the `books` table. Every ActiveRecord table class has a class method called `find`, which takes various arguments telling the method for what it should search.

`:all` in our `Book` ActiveRecord example is that ActiveRecord can recognize it when it sees it—the code needs to know which records to return, and `:all` is the flag that says it should return every one. The nice thing about using something like `:all` for this kind of "stands for" duty is that it also makes sense to the humans: You are a lot more likely to recognize what `:all` means when you come across it than `0`, or `-1`, or even (heaven forbid!) `0x29ef`.

These two uses for strings of characters—for regular data processing tasks on the one hand and for internal, symbolic, marker-type jobs on the other—make very different demands on the objects. If you are processing data, you will want to have the whole range of string manipulation tools at your fingertips: You might want the first ten characters of the title, or you might want to get its length or see whether it matches some regular expression. On the other hand, if you are using some characters to stand for something in your code, you probably are not very interested in messing with the actual characters. Instead, in this second case you just need to know whether this thing is the flag that tells you to find all the records or just the first record. Mainly, when you want some characters to stand for something, you simply need to know if *this* is the same as *that*, quickly and reliably.

Optimized to Stand for Something

By now you have probably guessed that the Ruby `String` class is optimized for the data processing side of strings while symbols are meant to take over the "stands for" role—hence the name. Since we don't use symbols for data processing tasks, they lack most of the classic string manipulation methods that we talked about in Chapter 4. Symbols do have some special talents that make them great for being symbols. For example, there can only ever be one instance of any given symbol: If I mention `:all` twice in my code, it is always exactly the same `:all`. So if I have:

```
a = :all
b = a
c = :all
```

I know that `a`, `b`, and `c` all refer to exactly the same object. It turns out that Ruby has a number of different ways to check whether one object is equal to another,[2] but

2. For more on object equality, see Chapter 12.

with symbols it doesn't matter: Since there can only be one instance of any given symbol, `:all` is always equal to itself no matter how you ask:

```
# True! All true!

a == c
a === c
a.eql?(c)
a.equal?(c)
```

In contrast, every time you say `"all"`, you are making a brand new string. So if you say this:

```
x = "all"
y = "all"
```

Then you have manufactured two different strings. Since both the strings happen to contain the same three characters, the two strings are equal in some sense of the word, but they are emphatically not identically the same object. The fact that there can only be one instance of any given symbol means that figuring out whether *this* symbol is the same as *that* symbol is not only foolproof, it also happens at lightning speeds.

Another aspect of symbols that makes them so well suited to their chosen career is that symbols are immutable—once you create that `:all` symbol, it will be `:all` until the end of time.[3] You cannot, for example, make it uppercase or lob off the second `'l'`. This means that you can use a symbol with confidence that it will not change out from under you.

You can see all these issues at play in hashes. Since symbol comparison runs at NASCAR speeds and symbols never change, they make ideal hash keys. Sometimes, however, engineers want to use regular strings as hash keys:

```
author = 'jules verne'
title = 'from earth to the moon'
hash = { author => title }
```

3. Or at least until your Ruby interpreter exits.

So what would happen to the hash if you changed the key out from underneath it?

```
author.upcase!
```

The answer is that nothing will happen to the hash, because the `Hash` class has special defenses built in to guard against just this kind of thing. Inside of `Hash` there is special case code that makes a copy of any keys passed in if the keys happen to be strings. The fact that the `Hash` class needs to go through this ugly bit of special pleading precisely to keep you from coming to grief with string keys is the perfect illustration of the utility of symbols.

In the Wild

In practice, the line between symbols and regular strings is sometimes a bit blurry. It is, for example, trivially easy to turn a symbol into a string: You just use the ubiquitous `to_s` method:

```
the_string = :all.to_s
```

To go in the reverse direction, you can use the `to_sym` method that you find on your strings:

```
the_symbol = 'all'.to_sym
```

The blurriness between symbols and strings sometimes also extends into the minds of Ruby programmers. For example, every object in Ruby has a method called `public_methods`, which returns an array containing the names of all of the public methods on that object. Now, you might argue that method names are the poster children for objects that stand for something (in this case a bit of code), and therefore the `public_methods` method should return an array of symbols. But call `public_methods` in a pre-1.9 version of Ruby, like this:

```
x = Object.new
pp x.public_methods
```

And you will get an array of strings, not symbols:

```
["inspect",
 "pretty_print_cycle",
 "pretty_print_inspect",
 "clone",
 ...
]
```

Is there something wrong with our reasoning? Apparently not, because in Ruby 1.9 `public_methods` does indeed return an array of symbols:

```
[:pretty_print,
 :pretty_print_cycle,
 :pretty_print_instance_variables,
 :pretty_print_inspect,
 :nil?,
 ...
]
```

The lesson here is that if you find symbols a bit confusing, you seem to be in very good company.

Staying Out of Trouble

Given the curious relationship between symbols and strings, it probably will come as no surprise that the best way to screw up with a symbol is to use it when you wanted a string, and vice versa. As we have seen, you want to use strings for data, for things that you might want to truncate, turn to uppercase, or concatenate. Use symbols when you simply want an intelligible thing that stands for something in your code.

The other way to go wrong is to forget which you need at any given time. This seems to happen a lot when using symbols as the keys in hashes. For example, take a look at this code fragment:

```
# Some broken code

person = {}
person[:name] = 'russ'
```

```
person[:eyes] = 'misty blue'

# A little later...

puts "Name: #{person['name']} Eyes: #{person['eyes']}"
```

The code here is broken, but you might have to look at it a couple of times to see that the keys of the `person` hash are symbols, but the `puts` statement tries to use strings. What you really want to say here is:

```
puts "Name: #{person[:name]} Eyes: #{person[:eyes]}"
```

This kind of mistake is common enough that Rails actually provides a Band-Aid for it in the form of the `HashWithIndifferentAccess` class. This convenient, but somewhat dubious bit of code is a subclass of `Hash` that allows you to mix and match strings and symbols with cheerful abandon.

Wrapping Up

In this chapter we have looked at symbols and saw that they exist purely to stand for something in your code. Symbols and garden variety strings have a lot in common—both are mostly just a stretch of characters. Unlike strings, symbols are specially tuned to their "stands for" purpose: Symbols are both unique—there can only ever be one `:all` symbol in your Ruby interpreter—and immutable, so that `:all` will never change. The good news is that once you understand that symbols and strings are like two siblings—related, but with different talents—you will be able to take advantage of the things that each does best.

This page intentionally left blank

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. . . .

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y