

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](http://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.”

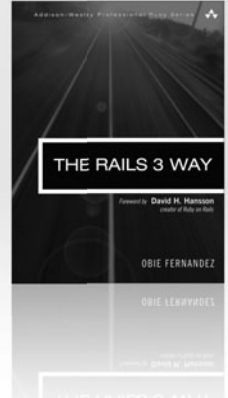
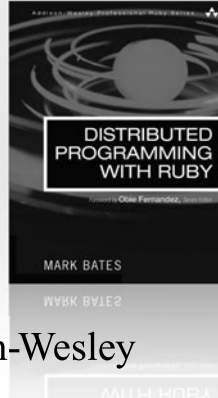
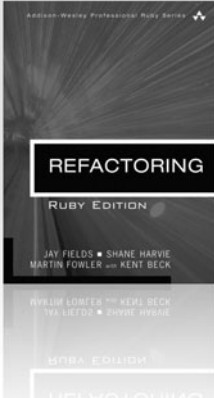
—Michael Fogus  
Contributor to the Clojure programming  
language and author of *The Joy of Clojure*

*This page intentionally left blank*

# ELOQUENT RUBY

# Addison-Wesley Professional Ruby Series

Obie Fernandez, Series Editor



◆◆ Addison-Wesley

Visit [informit.com/ruby](http://informit.com/ruby) for a complete list of available products.

The **Addison-Wesley Professional Ruby Series** provides readers with practical, people-oriented, and in-depth information about applying the Ruby platform to create dynamic technology solutions. The series is based on the premise that the need for expert reference books, written by experienced practitioners, will never be satisfied solely by blogs and the Internet.

PEARSON

◆◆ Addison-Wesley

Cisco Press

EXAM/CRAM

IBM Press

QUE

PRENTICE HALL

SAMS

Safari Books Online

# 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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales  
(800) 382-3419  
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales  
international@pearson.com

Visit us on the Web: [www.informit.com/aw](http://www.informit.com/aw)

*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

Copyright © 2011 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.  
Rights and Contracts Department  
501 Boylston Street, Suite 900  
Boston, MA 02116  
Fax: (617) 671-3447

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

---

*Foreword* xix

*Preface* xxi

*Acknowledgments* xxv

*About the Author* xxvii

## **PART I: The Basics** 1

### **Chapter 1: Write Code That Looks Like Ruby** 3

The Very Basic Basics 4

Go Easy on the Comments 6

Camels for Classes, Snakes Everywhere Else 8

Parentheses Are Optional but Are Occasionally Forbidden 9

Folding Up Those Lines 10

Folding Up Those Code Blocks 11

Staying Out of Trouble 12

In the Wild 13

Wrapping Up 15

### **Chapter 2: Choose the Right Control Structure** 17

If, Unless, While, and Until 17

Use the Modifier Forms Where Appropriate 19

Use each, Not for 20

A Case of Programming Logic 21

Staying Out of Trouble 23

In the Wild 25

Wrapping Up 27

### **Chapter 3: Take Advantage of Ruby's Smart Collections 29**

Literal Shortcuts 29

Instant Arrays and Hashes from Method Calls 30

Running Through Your Collection 33

Beware the Bang! 36

Rely on the Order of Your Hashes 38

In the Wild 38

Staying Out of Trouble 40

Wrapping Up 42

### **Chapter 4: Take Advantage of Ruby's Smart Strings 43**

Coming Up with a String 44

Another API to Master 47

The String: A Place for Your Lines, Characters, and Bytes 49

In the Wild 50

Staying Out of Trouble 51

Wrapping Up 52

### **Chapter 5: Find the Right String with Regular Expressions 53**

Matching One Character at a Time 54

Sets, Ranges, and Alternatives 55

The Regular Expression Star 57

Regular Expressions in Ruby 58

Beginnings and Endings 60

In the Wild 62

Staying Out of Trouble 63

Wrapping Up 64

### **Chapter 6: Use Symbols to Stand for Something 65**

The Two Faces of Strings 65

Not Quite a String 66

Optimized to Stand for Something 67

In the Wild 69  
Staying Out of Trouble 70  
Wrapping Up 71

## **Chapter 7: Treat Everything Like an Object—Because Everything Is** 73

A Quick Review of Classes, Instances, and Methods 74  
Objects All the Way Down 76  
The Importance of Being an Object 77  
Public, Private, and Protected 79  
In the Wild 81  
Staying Out of Trouble 82  
Wrapping Up 84

## **Chapter 8: Embrace Dynamic Typing** 85

Shorter Programs, But Not the Way You Think 85  
Extreme Decoupling 89  
Required Ceremony Versus Programmer-Driven Clarity 92  
Staying Out of Trouble 93  
In the Wild 94  
Wrapping Up 96

## **Chapter 9: Write Specs!** 97

Test::Unit: When Your Documents Just Have to Work 98  
A Plethora of Assertions 101  
Don't Test It, Spec It! 101  
A Tidy Spec Is a Readable Spec 104  
Easy Stubs 105  
. . . And Easy Mocks 107  
In the Wild 108  
Staying Out of Trouble 110  
Wrapping Up 113

## **PART II: Classes, Modules, and Blocks** 115

### **Chapter 10: Construct Your Classes from Short, Focused Methods** 117

Compressing Specifications 117  
Composing Methods for Humans 121

Composing Ruby Methods	122
One Way Out?	123
Staying Out of Trouble	126
In the Wild	127
Wrapping Up	128

## **Chapter 11: Define Operators Respectfully** 129

Defining Operators in Ruby	129
A Sampling of Operators	131
Operating Across Classes	134
Staying Out of Trouble	135
In the Wild	137
Wrapping Up	139

## **Chapter 12: Create Classes That Understand Equality** 141

An Identifier for Your Documents	141
An Embarrassment of Equality	142
Double Equals for Everyday Use	143
Broadening the Appeal of the == Method	145
Well-Behaved Equality	146
Triple Equals for Case Statements	149
Hash Tables and the eql? Method	150
Building a Well-Behaved Hash Key	152
Staying Out of Trouble	153
In the Wild	154
Wrapping Up	156

## **Chapter 13: Get the Behavior You Need with Singleton and Class Methods** 157

A Stubby Puzzle	158
A Hidden, but Real Class	160
Class Methods: Singletons in Plain Sight	162
In the Wild	164
Staying Out of Trouble	165
Wrapping Up	167

**Chapter 14: Use Class Instance Variables 169**

- A Quick Review of Class Variables 169
- Wandering Variables 171
- Getting Control of the Data in Your Class 174
- Class Instance Variables and Subclasses 175
- Adding Some Convenience to Your Class Instance Variables 176
- In the Wild 177
- Staying Out of Trouble 179
- Wrapping Up 179

**Chapter 15: Use Modules as Name Spaces 181**

- A Place for Your Stuff, with a Name 181
- A Home for Those Utility Methods 184
- Building Modules a Little at a Time 185
- Treat Modules Like the Objects That They Are 186
- Staying Out of Trouble 189
- In the Wild 190
- Wrapping Up 191

**Chapter 16: Use Modules as Mixins 193**

- Better Books with Modules 193
- Mixin Modules to the Rescue 195
- Extending a Module 197
- Staying Out of Trouble 198
- In the Wild 202
- Wrapping Up 205

**Chapter 17: Use Blocks to Iterate 207**

- A Quick Review of Code Blocks 207
- One Word after Another 209
- As Many Iterators as You Like 210
- Iterating over the Ethereal 211
- Enumerable: Your Iterator on Steroids 213
- Staying Out of Trouble 215
- In the Wild 217
- Wrapping Up 218

**Chapter 18: Execute Around with a Block 219**

- Add a Little Logging 219
- When It Absolutely Must Happen 224
- Setting Up Objects with an Initialization Block 225
- Dragging Your Scope along with the Block 225
- Carrying the Answers Back 227
- Staying Out of Trouble 228
- In the Wild 229
- Wrapping Up 231

**Chapter 19: Save Blocks to Execute Later 233**

- Explicit Blocks 233
- The Call Back Problem 234
- Banking Blocks 236
- Saving Code Blocks for Lazy Initialization 237
- Instant Block Objects 239
- Staying Out of Trouble 240
- In the Wild 243
- Wrapping Up 244

**PART III: Metaprogramming 247****Chapter 20: Use Hooks to Keep Your Program Informed 249**

- Waking Up to a New Subclass 250
- Modules Want To Be Heard Too 253
- Knowing When Your Time Is Up 255
- . . . And a Cast of Thousands 256
- Staying Out of Trouble 257
- In the Wild 259
- Wrapping Up 261

**Chapter 21: Use `method_missing` for Flexible Error Handling 263**

- Meeting Those Missing Methods 264
- Handling Document Errors 266
- Coping with Constants 267
- In the Wild 268

Staying Out of Trouble 270

Wrapping Up 271

## **Chapter 22: Use `method_missing` for Delegation 273**

The Promise and Pain of Delegation 274

The Trouble with Old-Fashioned Delegation 275

The `method_missing` Method to the Rescue 277

More Discriminating Delegation 278

Staying Out of Trouble 279

In the Wild 281

Wrapping Up 283

## **Chapter 23: Use `method_missing` to Build Flexible APIs 285**

Building Form Letters One Word at a Time 286

Magic Methods from `method_missing` 287

It's the Users That Count—All of Them 289

Staying Out of Trouble 289

In the Wild 290

Wrapping Up 292

## **Chapter 24: Update Existing Classes with Monkey Patching 293**

Wide-Open Classes 294

Fixing a Broken Class 295

Improving Existing Classes 296

Renaming Methods with `alias_method` 297

Do Anything to Any Class, Anytime 299

In the Wild 299

Staying Out of Trouble 303

Wrapping Up 303

## **Chapter 25: Create Self-Modifying Classes 305**

Open Classes, Again 305

Put Programming Logic in Your Classes 308

Class Methods That Change Their Class 309

In the Wild 310



Staying Out of Trouble 314

Wrapping Up 315

## **Chapter 26: Create Classes That Modify Their Subclasses 317**

A Document of Paragraphs 317

Subclassing Is (Sometimes) Hard to Do 319

Class Methods That Build Instance Methods 321

Better Method Creation with `define_method` 324

The Modification Sky Is the Limit 324

In the Wild 327

Staying Out of Trouble 330

Wrapping Up 332

## **PART IV: Pulling It All Together 333**

### **Chapter 27: Invent Internal DSLs 335**

Little Languages for Big Problems 335

Dealing with XML 336

Stepping Over the DSL Line 341

Pulling Out All the Stops 344

In the Wild 345

Staying Out of Trouble 347

Wrapping Up 349

### **Chapter 28: Build External DSLs for Flexible Syntax 351**

The Trouble with the Ripper 352

Internal Is Not the Only DSL 353

Regular Expressions for Heavier Parsing 356

Treetop for Really Big Jobs 358

Staying Out of Trouble 360

In the Wild 362

Wrapping Up 364

### **Chapter 29: Package Your Programs as Gems 367**

Consuming Gems 367

Gem Versions 368

The Nuts and Bolts of Gems	369
Building a Gem	370
Uploading Your Gem to a Repository	374
Automating Gem Creation	375
In the Wild	376
Staying Out of Trouble	377
Wrapping Up	380

## **Chapter 30: Know Your Ruby Implementation 381**

A Fistful of Rubies	381
MRI: An Enlightening Experience for the C Programmer	382
YARV: MRI with a Byte Code Turbocharger	385
JRuby: Bending the “J” in the JVM	387
Rubinius	388
In the Wild	389
Staying Out of Trouble	389
Wrapping Up	390

## **Chapter 31: Keep an Open Mind to Go with Those Open Classes 391**

## **Appendix: Going Further 393**

<i>Index</i>	397
--------------	-----

*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,<sup>1</sup> 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<sup>2</sup> 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 zani-ness 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.<sup>1</sup>

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](http://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:<sup>1</sup>

```
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.

`:a11` 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 `:a11` is the flag that says it should return every one. The nice thing about using something like `:a11` 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 `:a11` 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 `:a11` twice in my code, it is always exactly the same `:a11`. So if I have:

---

```
a = :a11
b = a
c = :a11
```

---

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,<sup>2</sup> 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, `:a11` 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 "a11", you are making a brand new string. So if you say this:

---

```
x = "a11"
y = "a11"
```

---

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 `:a11` symbol, it will be `:a11` until the end of time.<sup>3</sup> You cannot, for example, make it uppercase or lob off the second '1'. 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 `:a11` symbol in your Ruby interpreter—and immutable, so that `:a11` 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,<sup>1</sup> 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
end
```

---

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
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**,<sup>2</sup> 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
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:<sup>3</sup>

---

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:<sup>4</sup>

---

```
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

- " (double quotes), use with string literals, 44–45
- ' (single quotes), use with string literals, 44–45
- (subtraction) operator
  - as binary or unary operator, 132
  - overloading, 131
- . (period)
  - for matching any single character, 54
  - in module syntax, 185
  - using asterisk (\*) in conjunction with, 58
- / (division) operator, 131
- / (forward slashes), in regular expression syntax, 58–59
- : (colon), in symbol syntax, 66–67
- :: (double-colon), in module syntax, 185
- ; (semicolon), for separating statements in Ruby code, 10–11
- \ (backslash)
  - escaping special meanings of punctuation characters in regular expressions, 54
  - escaping strings, 44–45
- | (or) operator, 131
- | (vertical bar), in syntax of alternatives in regular expressions, 56–57
- ||= operator, in expression-based initialization, 26–27
- + (addition) operator
  - as binary or unary operator, 132
  - non-commutative nature of, 137
  - overloading, 131
  - when to use, 136–137
- =- operator, testing if regular expression matches a string, 59–60
- == (double-equals) operator
  - broadening the scope of, 145–146
  - numeric classes accepting `Float` as equals, 154–156
  - overview of, 143–144
  - RSpec and, 138
  - symmetry principal and, 146–147
  - transitive property of, 147–149
- === (triple equals) operator, for `case` statements, 23, 149–150
- => (hash rocket), 30
- ! method names ending with, 48
- ! unary operator, 131–132
- #, in comment syntax, 6
- \$, as string delimiter, 45
- % (formatting operator), strings, 137–138
- % (modulo) operator, 131, 152
- %q, for arbitrarily quoted strings, 45–46
- & (and) operator, 131
- () (parentheses)
  - readability and, 12
  - Ruby conventions for calling defining/ methods, 9–10
- \* (asterisk)
  - in method definition with extra arguments, 31–32
  - in regular expressions, 57–58

- \* (multiplication) operator, 131
- ? (question mark), using with regular expressions, 62–63
- ?: (ternary operator), in expression-based decision making, 26
- @@, in class variable syntax, 169
- [ ] (square brackets)
  - adding to indexing-related class, 135
  - operator-like syntax and, 133
  - as string delimiter, 45
  - using with regular expressions, 55
- [ ]=
  - adding to indexing-related class, 135
  - operator-like syntax and, 133
- ^ (exclusive or) operator, 131
- { } (braces), in code block syntax, 11
- << (left shift operator), 131, 135
- <=> operator
  - Float and Fixnum classes and, 154–156
  - sort method and, 214

## A

- accessor methods, using with class variables, 170
- ActiveRecord
  - callbacks and, 177
  - composed method approach and, 127–128
  - as database interface library, 335
  - DataMapper compared with. *see* DataMapper
  - example of delegation, 282–283
  - example of execute around, 230
  - example of saved code blocks, 243
  - examples of internal DSLs, 346
  - find method, 66–67
  - magic methods, 291–292
  - silence method, 231
- add\_unique\_word method, 120
- addition (+) operator. *see* + (addition) operator
- alias\_method, for renaming methods, 297–299
- alternatives, in regular expressions, 55–57
- ancestors method, for viewing inheritance ancestry, 199
- and (&) operator, 131
- APIs
  - avoiding trouble when using method\_missing for, 289–290

- building form letters one word at a time, 286–288
- building with method\_missing, 292
- examples of use of method\_missing for, 290–292
- review of applying method\_missing to, 292
- supported by strings, 47–49
- transition from API to DSL, 341–344
- user focus in creating easy-to-use APIs, 289
- when to use instead of internal DSLs, 348
- archives, gems and, 370
- arguments
  - code blocks taking, 208
  - execute around methods taking, 226–227
  - methods taking fixed or variable numbers of, 30–31
  - naming conventions, 8
  - singleton methods accepting, 159
- arrays
  - APIs for, 35
  - caution when iterating over, 40–41
  - each method, 34, 217
  - improper use of, 41
  - method-passing with, 30–32
  - monkey patching for adding methods to, 302
  - order of, 38
  - overview of, 29
  - public methods for array instances, 36
  - reverse method, 36–37
  - shortcuts for accessing, 30
  - sort method, 37
- assert method, Test::Unit, 98
- assert\_equal method, Test::Unit, 98
- assert\_match method, Test::Unit, 101
- assert\_nil method, Test::Unit, 101
- assert\_not\_equal method, Test::Unit, 101
- assert\_not\_nil method, Test::Unit, 101
- assertions, in Test::Unit, 101
- asterisk (\*)
  - in method definition with extra arguments, 31–32
  - in regular expressions, 57–58
- asymmetrical equality relationships, 147
- at\_exit hook
  - informing when time is up, 255–256
  - in Test::Unit, 259–260

- attr\_accessor
  - accessing class instance variables, 176
  - in default set of methods in Object class, 82
  - as subclass-changing method, 327–328
- attr\_reader, as subclass-changing method, 327
- attr\_writer, as subclass-changing method, 327–328
- attributes, at class level, 176–177
- automating
  - gem creation, 375–377
  - testing gems, 94
- B**
- backslash (\)
  - escaping special meanings of punctuation characters, 54
  - escaping strings, 44–45
- BasicObject, use in delegation with
  - method\_missing, 280–281
- Bignum class, 154–156
- binary operators
  - operating across classes, 134–135
  - overview of, 131–132
- bitwise operators, 131
- blank? method, adding to String class, 301
- block\_given? passing code blocks in methods, 208, 233
- blocks. *see* code blocks
- boolean logic
  - false and true values in Ruby, 23–25
  - mapping boolean operators to union and intersection operations, 135
- braces ({}), in code block syntax, 11
- break, in code blocks, 216
- bugs, 94. *see also* tests
- bytes, strings as collections of, 49–50
- C**
- C language, 382
- C# language, 336
- call backs
  - ActiveRecord objects and, 177
  - creating listeners for, 234–236
  - using explicit code blocks for, 236–237
- call method
  - calling code blocks explicitly, 234
  - Proc.new and, 241
- camel case, class naming conventions, 8
- Capistrano, 243–244
- Cardinal, 382
- case sensitivity, working with strings, 47
- case statements
  - example of use of, 21–23
  - triple equals operator (===) for, 149–150
- characters
  - matching any one of a bunch of characters, 55
  - matching one character at a time, 54–55
  - strings as collections of, 49
- chomp method, working with strings, 47
- chop method, working with strings, 47
- clarity
  - of code, 94
  - qualities of good code, 4
- class definitions, executable. *see* classes, self modifying
- class instance variables
  - avoiding trouble when using, 179
  - examples of use of, 177–179
  - for holding onto classwide values, 174–175
  - review of, 179
  - singleton class used to add convenience to, 176–177
  - subclasses and, 175–176
- class methods. *see also* singleton methods
  - adding convenience to class instance variables, 176–177
  - avoiding trouble when using, 165–166
  - for building instance methods, 321–323
  - defining, 163–164
  - extending modules and, 197–198
  - handling missing constants. *see* const\_missing
  - included hook used with, 254–255
  - making structural changes to classes, 309–310
  - overview of, 162
  - uses of, 164–165
- class variables
  - avoiding trouble when using, 179
  - example of use of, 170
  - problems associated with global nature of, 171–174

- class variables (*continued*)
  - review of, 179
  - storing class level data with, 169
  - tendency to wander from class to class, 171
  - URI class and, 177–178
- `class_eval`, for creating methods, 322–323, 329
- classes
  - accessing in modules, 182–183
  - adding iterator methods to, 210–211
  - avoiding name collisions, 377–378
  - benefits of dynamic typing, 85, 89
  - binary operators used across, 134–135
  - as both factory and container, 182
  - changing class definition, 305–308
  - class/instance approximation in defining methods, 157
  - composed method for building, 122–123
    - as container for methods, 74
  - defining, 294
  - do anything to any class, anytime, 297–299
  - as factory for creating instances, 74–75
  - fixing broken, 295–296
  - flexibility resulting from decoupling, 90–91
  - holding onto classwide values, 174–175
  - hook for informing when a class gains a subclass, 250–253, 257–259
  - hook for informing when a module gets included in a class, 253–255
  - mixins for sharing code between unrelated classes, 195–197
  - modifying, 295–297
  - modules for grouping related, 182
  - modules for organizing into hierarchies, 181
  - modules for swapping groups of related classes at runtime, 186–187
  - naming conventions, 8
  - open nature in Ruby. *see* open classes
  - preference for bare collections over specialized classes, 38–40
  - renaming methods using `alias_method`, 297–299
  - storing class level data, 169
  - superclasses, 75–76
  - when to use modules vs. naked classes, 189
  - writing methods for. *see* methods, writing
- classes, self modifying
  - adding programming logic to classes, 308–309
  - avoiding trouble when using, 314–315
  - class methods that change class, 309–310
  - defining classes and, 305–308
  - examples of use of, 310–313
  - overview of, 305
  - review of, 315–316
- classes, that modify subclasses
  - avoiding trouble when using, 330–332
  - class methods that build instance methods, 321–323
  - `define_method` for creating methods, 324
  - difficulty of subclassing and, 319–321
  - example of paragraph subclass of document class, 317–319
  - examples of use of, 327–329
  - no limits on modifying subclasses from superclass methods, 324–326
  - overview of, 317
  - review of, 332
- closure (scope)
  - avoiding trouble when using, 241–242
  - code blocks drag scope along to wherever they are applied, 225–227
- code
  - clarity and conciseness of, 94
  - concise vs. cryptic, 94
  - dynamic typing increasing compactness of, 85–89
  - embedding in classes, 308
  - format of. *see* code format
  - less code, less likelihood of error, 84
  - qualities of good code, 4
  - readability of, 12–13
  - sharing between unrelated classes, 195
- code blocks
  - `at_exit` hook, 255–256
  - multiline vs. single line, 12
  - Ruby conventions, 11
- code blocks, as iterators
  - adding multiple iterators, 210–211
  - adding single iterator, 209–210
  - avoiding trouble when using, 215–216
  - creating by tacking on to the end of method calls, 207–208

- Enumerable module and, 213–215
  - overview of, 207
  - returning values, 208–209
  - review of, 218
  - spectrum of iterator types, 217–218
  - taking arguments, 208
  - writing iterators for collections that do not yet exist, 211–213
- code blocks, saving for later use
  - applying to call backs, 234–237
  - applying to lazy initialization, 237–239
  - avoiding trouble when using, 240–242
  - examples of use of, 243–244
  - explicit vs. implicit approaches to passing blocks, 233–234
  - overview of, 233
  - producing instant block objects, 239–240
  - review of, 244–245
- code blocks, using execute around
  - applying to logging, 222–224
  - applying to object initialization, 225, 229–230
  - avoiding trouble when using, 228–229
  - delivering code where needed, 219
  - dragging scope along to wherever they are applied, 225–227, 241–242
  - for functions that must happen before or after operations, 224
  - returning something from, 227–228
  - silence method for turning logging off, 231
- code format
  - breaking rules and, 14–15
  - code blocks, 11
  - indentation, 5–6
  - naming conventions, 8–9
  - "one statement per line" convention, 10–11
  - parentheses in calling/defining methods, 9–10
  - qualities of good code, 4
  - readability and, 12–13
  - review of conventions, 15
- collections
  - adding left shift operator to collection class, 135
  - caution when iterating over, 40–41
  - collection-related methods in Enumerable class, 213
  - improper use of arrays and hashes, 41–42
  - iterating through, 33–36
  - knowing which methods change and which leave as is, 36–38
  - method calls for accessing, 30–33
  - order of hashes, 38
  - overview of, 29
  - preference for bare collections over specialized classes, 38–40
  - review of, 42
  - shortcuts for accessing, 29–30
- colon (:), in symbol syntax, 66–67
- comments
  - dynamic typing and, 93
  - example in `set.rb` class, 13–14
  - when and how often to use, 6–8
- comparison operator, 23
- complexity, simplicity as solution to, 92
- composed method
  - `ActiveRecord::Base` class example, 127–128
  - applying to `TextCompressor` class, 121
  - for building classes, 122–123
  - characteristics of, 121–122
- compression algorithm, 117–118
- conciseness, of code, 4, 94
- conditions, syntax in control statements, 10
- consistency, of Ruby object system, 76–77
- `const_missing`
  - avoiding trouble when using, 270–271
  - examples of use of, 269–270
  - handling missing constants, 267–268
  - review of, 271
- constants
  - accessing in modules, 183
  - handling missing. *see* `const_missing`
  - modules for organizing into hierarchies, 181–182
  - modules for swapping groups of related constants at runtime, 186–187
  - naming conventions, 8–9
  - stashing in mixins, 204–205
- containers
  - modules as, 181–182
  - treating modules as object rather than static containers, 186
- control structures
  - `||=` in expression-based initialization, 26–27
  - boolean logic and, 23–25

control structures (*continued*)

- case statement, 21–23
  - code capturing values of `while` or `if` statements, 25
  - each method preferred over `for` loops, 20–21
  - `if`, `unless`, `while`, and `until` statements, 17–19
  - modifier forms, 19–20
  - overview of, 17
  - review of, 27
  - syntax for conditions in, 10
  - ternary operator (`?:`) in decision making, 26
- Cucumber testing tool, 363–364

**D**

## data

- storing class level, 169, 174
  - using strings for processing, 66–67
- data types
- built-in, 58–60
  - disadvantages of adding type checking code, 91
  - dynamic. *see* dynamic typing
  - static. *see* static typing
  - type documentation, 92

## DataMapper

- example of use of modules in, 190–191
- mixins used by, 202–203

## debugging, logging for, 219

## decomposing classes

- into small methods, 123
- troubles arising from, 126–127

## decoupling, with dynamic typing, 89–92

`def`

- class methods that build instance methods, 322
- last `def` principle, 295

`define_method`, for creating methods, 324, 328`defined?` boolean logic and, 24`delegate.rb` file, 281–282

## delegation

- avoiding trouble when using, 279–281
- example of use by `ActiveRecord`, 282–283
- `method_missing` applied to, 277–278
- overview of, 273
- problems with traditional style of, 275
- pros/cons of, 274–275

## review of, 283

- selective approach to, 278–279
- `SimpleDelegator` class, 281–282
- `delete` method, for arrays, 37

`Dir` class, 217

## directories

- generating directory structure of gems, 377
- organizing for gems packaging, 370–372

division (`/`) operator, 131

## DLL Hell, 370

`do` keyword, in code block syntax, 11

## documentation

- compensating for lost documentation due to required type declarations, 92–93
- Ruby implementations, 389

`DocumentIdentifier` class, 142

## documents

- compressing specification documents, 117–118
- creating identifier, 142
- handling document errors, 266–267
- lazy documents, 86–89
- paragraph subclass of document class, 317–319
- Ruby coding conventions illustrated in `Document` class, 5

Domain Specific Languages, external. *see* DSLs (Domain Specific Languages), externalDomain Specific Languages, internal. *see* DSLs (Domain Specific Languages), internaldouble quotes ("`"`), use with string literals, 44–45double-colon (`::`), in module syntax, 185double-equals (`==`) operator. *see* `==` (double-equals) operator`downcase` method, working with strings, 47

## DSLs (Domain Specific Languages), external

- avoiding trouble when using, 360–362
- building parser for XML processing language, 353–356

## examples of use of, 362–364

## overview of, 336, 351–352

## regular expressions for parsing, 356–358

## review of, 364–365

## Treetop parsing tool, 358–360

## when to use as alternative to internal DSL, 352

## DSLs (Domain Specific Languages), internal

- avoiding trouble when using, 347–349
- based on Ruby code, 352

- dealing with XML, 336–341
- examples of use of, 345–346
- `method_missing` used with, 344
- narrow focus of, 336
- overview of, 335
- review of, 349
- transition from API to DSL, 341–344
- when to use as alternative to external DSL, 352
- duck typing, 88–89
- dynamic typing
  - compactness of code and, 85–89
  - comparing `File` and `StringIO` classes, 94–95
  - compensating for lost documentation due to
    - required type declarations, 92–93
  - extreme decoupling with, 89–92
  - overview of, 85
  - set class and, 95–96
  - trade offs in use of, 93–94

## E

- each method
  - adding iterator methods to classes, 210–212
  - avoiding trouble when iterating arrays, 40
  - iteration with, 34
  - preferred over `for` loops, 20–21
  - types of iterators and, 217
- `each_address` method, `Resolv` class, 217
- `each_cons` method, `Enumerable` module and, 213–214
- `each_object` method, `ObjectSpace` class, 217–218
- `each_splice` method, `Enumerable` module and, 214
- eigenclasses. *see* singleton classes
- encryption
  - managing with class methods, 309–310
  - managing with programming logic in classes, 308–309
- `end` keyword, in code block syntax, 11
- `Enumerable` module, 213–215
- `Enumerator` class, 214
- `eq1?` method
  - `Hash` class using, 152–153
  - overview of, 150–152
  - restrictive view of equality in, 153

- `equal?` method, for testing object identity, 143
- equality
  - avoiding trouble when using, 153–154
  - broadening the scope of double-equals operator, 145–146
  - double-equals (`==`) operator, 143–144
  - `eq1?` method, 150–153
  - `equal?` method, 143
  - `Float` and `Fixnum` classes and `<=>` operator, 154–156
  - identifiers and, 142
  - methods for, 142–143
  - overview of, 141
  - review of, 154–156
  - symbols and, 67–68
  - symmetry principal and, 146–147
  - transitive property and, 147–149
  - triple equals operator (`===`), 149–150
- ERB, 362–363
- `eval` method, `Object` class, 78
- exception handling. *see also* `method_missing`,
  - error handling with
    - with `execute around`, 228
  - handling document errors, 266–267
  - internal DSLs and, 347
  - logging and, 222, 224
- exclusive or (`^`) operator, 131
- executable class definitions. *see* classes, self
  - modifying
- `execute around`
  - avoiding trouble when using, 228–229
  - for functions that must happen before or after operations, 224
  - initializing objects with, 225, 229–230
  - passing arguments and, 226–227
  - returning something from code blocks, 227–228
- external DSLs. *see* DSLs (Domain Specific Languages), external

## F

- false
  - in boolean logic, 23–24
  - `false` as an object, 76
- `File` class, comparing with `StringIO` class, 94–95



filenames, avoiding name collisions, 378  
 find method, ActiveRecord, 66–67  
 find\_index, map method compared with, 35  
 Fixnum class, 154–156  
 Float class, 154–156  
 floating point numbers, 296  
 for loops, 20–21  
 formatting operator (%), for strings, 137–138  
 forward slashes (/), in regular expression syntax, 58–59  
 forwardable.rb, 328–329  
 Fowler, Martin, 336

## G

gem files, 370  
 gem install command, 374  
 gem list command, 368–369  
 Gemcutter, adding gems to Gemcutter repository, 375–376  
 gems  
   automating creation of, 375–376  
   avoiding trouble when using, 377–380  
   building, 370–374  
   creating, 378–379  
   examples of use of, 376–377  
   installing and consuming, 367–368  
   nuts and bolts of, 369–370  
   packaging programs as, 367  
   review of, 380  
   shoulda gem, 108  
   uploading to repository, 374–375  
   versioning support, 368–369  
 gemspec file, 373–374  
 GEM::Specification instances, 229–230  
 gets method, Object class, 78  
 global variables, class variables compared with, 174  
 gsub  
   inflection rules based on, 50–51  
   passing regular expressions into, 60

## H

HAML, 361  
 hash rocket (=>), 30

hashes  
   APIs for, 35  
   caution when iterating over, 40–41  
   each method, 34, 217  
   Hash class, 69  
   hash tables and eql? method, 150–153  
   hash values, 152  
   improper use of, 41–42  
   method-passing with, 33  
   order of, 38  
   overview of, 29  
   public methods, 36  
   shortcut for accessing, 30  
   symbols as hash keys, 68–71  
 HashWithIndifferenceAccess class, 71  
 helper methods, Rails, 203–204  
 hoe, for automating creation of gems, 376–377  
 hooks  
   avoiding trouble when using, 257–259  
   examples of use of, 259–260  
   informing when a class gains a subclass, 250–253  
   informing when a module gets included in a class, 253–255  
   informing when time is up, 255–256  
   method\_missing. *see* method\_missing  
   overview of, 249  
   review of, 261  
   set\_trace\_func, 256–257  
   value of, 332  
 HTML  
   HAML for HTML templating, 361  
   Rails helper methods for creating, 203–204

## I

identifiers  
   creating document identifier, 142  
   testing object identity, 143  
 if statements  
   case statement compared with, 23  
   code capturing values of, 25  
   example of use of, 17–18  
   modifier forms of, 20  
 included method, informing when a module gets included in a class, 253–255

- indentation, Ruby conventions, 5–6
  - indexing strings, 52
  - inflection rules, for strings, 50–51
  - inheritance
    - ancestors method, 199
    - class variables searching for associated classes, 171, 173
    - mixin modules and, 201–202
    - superclasses in inheritance tree, 193
  - inherited method
    - avoiding trouble when using, 257–259
    - hook for informing when a class gains a subclass, 250–253
  - initialization
    - defining classes, 294
    - of objects using execute around, 225
    - saved code blocks used for lazy initialization, 237–239
    - of variables, 26
  - initialize method, for defining classes, 294
  - inject method, collection methods, 35–36
  - instance\_of?, 145
  - instance methods
    - class methods that build, 321–323
    - instance\_methods method, 307
    - instance.method\_name, 74
  - instance variables
    - attaching to class objects, 174
    - instance\_variables method, 79
    - naming conventions, 8
  - instances
    - classes as factory for creating, 74–75
    - class/instance approximation in defining methods, 157
    - inheriting methods of Object class, 78
    - singleton methods defined for single object instance, 158–159
  - integers, 154–156
  - interfaces, 285. *see also* APIs
  - internal DSLs. *see* DSLs (Domain Specific Languages), internal
  - intersection operations, mapping boolean operators to, 135
  - IronRuby implementation, 382
  - iteration
    - adding an iterator, 209–210
    - adding multiple iterators, 210–211
    - avoiding trouble when using, 215–216
    - caution when iterating over arrays and hashes, 40–41
    - code blocks used as iterators, 207
    - Enumerable module and, 213–215
    - spectrum of iterator types, 217–218
    - through collections, 33–36
    - writing iterators for collections that do not yet exist, 211–213
- J**
- JAR file Hell, 370
  - Java
    - examples of general purpose languages, 336
    - JRuby and, 387
  - Java Virtual Machine (JVM), 387
  - JRuby
    - overview of, 382, 387–388
    - support and documentation, 389
  - JVM (Java Virtual Machine), 387
- K**
- kind\_of? method
    - double-equals (==) operator and, 146
    - locating modules in classes with, 199
- L**
- lambda method, creating default Proc object using, 239–241
  - lazy initialization, 237–239
  - "leave it to the last minute" technique, 88
  - left shift operator (<<), 131, 135
  - lib directory
    - organizing for gems packaging, 372
    - sow command generating, 377
  - lines, strings as collections of, 50
  - listeners, for call backs, 234–236
  - literals, shortcuts for accessing collections, 29–30
  - load methods, managing logging with, 221–222
  - logging
    - adding to database interactions, 220
    - capturing return values, 228

logging (*continued*)  
 for debugging, 219  
 load and save methods for managing, 221–222  
 passing arguments and, 227  
 silence method for turning off, 231  
 using code blocks for, 222–223  
 using explicit log messages, 220–221  
 long running tests, 110  
 lstrip method, for strings, 47

## M

magic methods. *see also* `method_missing`,  
 building APIs with  
 example in ActiveRecord, 291–292  
 example in ActiveSupport class, 290–291  
 overview of, 288

map method, for collections, 35

Matsumoto, Yukihiro, 382

Matz's Ruby Interpreter. *see* MRI (Matz's Ruby  
 Interpreter)

metaclasses. *see* singleton classes

metadata, gems and, 370, 373

metaprogramming

hooks. *see* hooks

monkey patching. *see* monkey patching

need for testing in, 315–316

overview of, 249

self modifying classes. *see* classes, self modifying  
 superclasses as basis for class modifying code.

*see* classes, that modify subclasses

when to use, 331–332

`method_added`, 256

`method_missing`

types of hooks, 256

used in conjunction with internal DSL, 344

value of, 332

`method_missing`, building APIs with

avoiding trouble when using, 289–290

building form letters one word at a time,  
 286–288

examples of use of, 290–292

overview of, 285

review of, 292

user focus in creating easy-to-use APIs, 289

`method_missing`, delegation with

avoiding trouble when using, 279–281

example of use by ActiveRecord, 282–283

overview of, 273

problems with traditional style of delegation,  
 275

process of applying delegation, 277–278

pros/cons of delegation, 274–275

review of, 283

selective approach to delegation, 278–279

SimpleDelegator class, 281–282

`method_missing`, error handling with, 263–264

avoiding trouble when using, 270–271

handling document errors, 266–267

overriding, 265

review of, 271

what occurs when Ruby fails to find a method,  
 264–265

whiny nil facility in Rails as example of use of,  
 268–269

methods

array method-passing feature, 30–32

calling on object instances, 74–75

class methods that build instance methods,  
 321–323

classes as container for, 74

class/instance approximation in defining, 157

creating code blocks by tacking on to end of  
 method calls, 207–208

`define_method` for creating, 324

defining module-level, 189

defining operators vs. using methods, 135

determining when methods are defined, 307

dynamic typing and, 85

for equality, 142–143

fundamental nature of method calls in Ruby,  
 81–82

handling missing. *see* `method_missing`

hash method-passing feature, 33

“if the method is there, it is the right object,” 94

inheriting default set from `Object` class,  
 77–78

looking for in superclasses, 75–76

mixing instance methods with class methods,  
 254–255

- modifying classes and, 295
  - modifying subclasses from superclass methods, 324–326
- modules as container for, 182, 184–185
- naming conventions, 8
- operator-to-method translation, 130
- parentheses in calling/defining, 9–10
- public, private, and protected, 79–81
- public methods for arrays and hashes, 36
- redefining on broken classes, 295–296
- reflection-oriented, 79
- renaming using `alias_method`, 297–299
- singleton methods overriding class-defined methods, 159–160
- that take code blocks, 223–224
- methods, writing
  - `ActiveRecord::Base` class example, 127–128
  - composed method way of building classes, 122–123
  - compressing specifications, 117–121
  - overview of, 117
  - qualities of good methods, 121–122
  - review of, 128
  - single-exit approach, 123–126
  - troubles arising from decomposing methods, 126–127
- MiniSpec, 110
- MiniTest, 110
- mixin modules
  - as alternative to superclasses, 193–195
  - avoiding trouble when using, 198–202
  - constants stored in, 204–205
  - `Mapper` example of use of, 202–203
  - for extending modules, 197–198
  - inheritance relationships and, 201–202
  - overview of, 193
  - Rails helper methods using, 203–204
  - review of, 205
  - as solution for sharing code between unrelated classes, 195–197
- mocha
  - singleton methods and, 165
  - utilities for `Test::Unit`, 109
- mocks
  - `RSpec`, 107–108
  - singleton methods and, 165
- models, object oriented programming as support system for, 157
- modifier forms, of control structures, 19–20
- modifiers, strings, 48
- module variables, 178–179
- `module_eval`, for creating methods, 329
- modules
  - accessing classes in, 182–183
  - accessing constants in, 183
  - adding module variables to, 178–179
  - avoiding name collisions, 377–378
  - avoiding trouble when using, 189–190
  - benefits of dynamic typing, 85
  - building incrementally, 185
  - class hierarchy and, 201
  - as containers, 181–182
  - economical use of, 190–191
  - extending, 197–198
  - grouping related classes in, 182
  - grouping utility methods in, 184–185
  - hook for informing when a module gets included in a class, 253–255
  - including in classes, 195–196
  - mixing into class. *see* mixin modules
  - nesting, 183–184
  - review of, 191
  - treating as objects, 186–189
- modulo (%) operator, 131, 152
- monkey patching. *see also* open classes
  - do anything to any class, anytime, 297–299
  - examples of use of, 299–302
  - how it works, 307–308
  - modifying existing classes, 296–297
  - renaming methods using `alias_method`, 297–299
- MRI (Matz's Ruby Interpreter)
  - overview of, 382–385
  - support and documentation, 389
  - YARV as next generation implementation of, 385
- multiline strings, 46, 61–62

multiplication (\*) operator, 131  
mutability, of strings, 51–52

## N

names

accessing classes in modules by, 182–183  
`alias_method` for renaming methods,  
297–299

avoiding collisions, 377–378  
example in `set.rb` class, 14  
execute around and, 228–229

gems and, 371

method, 122

objects and name collisions, 82–83

Ruby conventions, 8–9

variable, 8

namespaces, creating name-space modules, 189

`NaN` (Not a Number), 296

nesting modules, 183–184

`nil`

boolean logic and, 23–25

initializing variables and, 26

as an object, 77, 84

whiny `nil` facility in Rails, 268–269

Not a Number (`NaN`), 296

`not` operator, 132

numeric classes

accepting `Float` as equals, 154–156

not supporting singleton methods, 159

## O

object oriented programming

Ruby as OO programming language, 73

as support system, for models, 157

object relational mappers

`ActiveRecord`. *see* `ActiveRecord`

`DataMapper`. *see* `DataMapper`

objects

avoiding trouble when using, 82–84

`BasicObject`, 280–281

classes, instances, and methods, 74–76

consistency of Ruby object system, 76–77

dynamic typing. *see* dynamic typing

equality. *see* equality

fundamental nature of method calls in Ruby,  
81–82

“if the method is there, it is the right object,” 94  
initializing using `execute around`, 225,  
229–230

methods, 77–79

modules as, 186–189

name collisions and, 82–83

object class, 77

overview of, 73–74

public, private, and protected methods, 79–81

referencing with variables, 77

review of, 84

singleton methods, 158–159

`ObjectSpace` class, 217–218

open classes. *see also* monkey patching

avoiding trouble when using, 303

creating self-modifying classes, 305

defining classes, 294

examples of use of, 299–302

fixing broken classes, 295–296

improving existing classes, 296–297

modifying classes, 295

overview of, 293–294

renaming methods using `alias_method`,  
297–299

review of, 303–304

value of, 332

`OpenStruct` class, 290–291

operators

cases/situations calling for, 135–137

commutative, 137

defining, 129–131

overview of, 129

review of, 139

string formatting, 137–138

types in Ruby, 131–133

using across classes, 134–135

or (!) operator, 131

order, of arrays and hashes, 38

overloading operators, 129

overriding methods

errors and, 83

`method_missing`, 265

methods in superclass unable to override

methods in subclasses, 200

## P

packaging programs, as gems. *see* gems

parentheses ( )

- readability and, 12
- Ruby conventions for calling defining/methods, 9–10

`parse_statement` method, 357

parsers

- based on regular expressions, 356–358
- building for XML processing language, 353–356
- examples of external DSLs, 364
- HAML and, 361–362
- Treetop for building, 358–360

`Pathname` class, 299–300

pattern matching, 150

period (.)

- for matching any single character, 54
- in module syntax, 185
- using asterisk (\*) in conjunction with, 58

polymorphism, 88

`pop` method, for arrays, 37

`print` method, `Object` class, 78

private methods, 79–81

`Proc` class, 239–241

`Proc.new`, 240–241

programming

- metaprogramming. *see* metaprogramming
- object oriented, 73, 157
- trade offs in programming languages, 336

programming logic, adding to classes, 308–309, 314

programs, packaging as gems, 367

protected methods, 81

public methods

- overview of, 79
- returning all public methods of an object, 69

`public_methods`, `Object` class, 79

`push` method, for arrays, 37

`puts` method, `Object` class, 78

## Q

question mark (?), using with regular expressions, 62–63

## R

RACC, for building parsers, 359

Rails

- example of `const_missing` hook, 270
- example of on-the-fly class modification, 312–313
- example of saved code blocks, 243
- helper methods using mixins, 203–204
- whiny nil facility, 268–269

Rake

- as build tool, 335
- example of `const_missing` hook, 269–270
- example of saved code blocks, 243–244
- examples of internal DSLs, 345–346
- specifying executable scripts in gems, 374

`rake` command, 374

`rake push` command, 376

Rakefiles

- automating creation of gems, 375–376
- `sow` command generating, 377

ranges

- of characters in regular expressions, 56
- indexing strings and, 52

readability, of code, 12–13

reflection-oriented methods, 79

`Regexp` data type, 58

regular expressions

- asterisk (\*) symbol in, 57–58
- `case` statement detecting match, 23
- HAML and, 361
- matching beginnings and endings of strings, 60–62
- matching one character at a time, 54–55
- mistakes to avoid, 63
- as objects, 76
- overview of, 53
- parser based on, 356–358
- pattern matching against strings, 150
- resources for use of, 394
- review of, 64
- sets, ranges, and alternatives, 55–57
- `time.rb` example, 62–63

repository, uploading gems to, 374–375

`require` method, `Object` class, 82

- required type declarations, compensating for lost documentation due to, 92–93
  - `Resolve` class, 217
  - resources, for Ruby, 393–395
  - `respond_to` method, 146–147
  - return, in code blocks, 216
  - reverse method, for arrays, 36–37
  - REXML XML parsing library, 338–339
  - Ripper DSL, 352–353
  - RSpec
    - double-equals (`==`) operator, 138
    - examples, 104
    - independence of test, 111
    - internal DSLs and, 345–346
    - MiniSpec, 110
    - mocks, 107–109
    - overview of, 102–104
    - parameters, 105
    - saved code blocks and, 243
    - shoulda gem providing RSpec-like example, 108
    - singleton methods and, 165
    - specifying executable scripts in gems, 374
    - stubs, 106–107
    - as testing utility, 335
    - tidy and readable specs, 104–105
  - `rstrip` method, for strings, 47
  - Rubinius, 382, 388–389
  - Ruby implementations
    - avoiding trouble when using, 389
    - extending, 389
    - JRuby, 387–388
    - MRI, 382–385
    - overview of, 381
    - review of, 390
    - Rubinius, 388–389
    - versions and, 381–382
    - YARV, 385–387
  - Ruby versions
    - comparing Ruby versions, 381–382
    - managing transition between, 311–312
    - MRI supporting Ruby 1.8, 383
    - YARV supporting Ruby 1.9, 381–382
  - RubyForge, 375
  - RubyGems. *see* gems
  - `ruby-mp3info`, 368
  - RubySpec project, 109–110
  - run-time decisions, putting programming logic in classes and, 308
- ## S
- save methods, 221–222
  - scope (closure)
    - avoiding trouble when using, 241–242
    - code blocks drag scope along to wherever they are applied, 225–227
  - scope, of class methods, 165–166
  - scripts, specifying executable scripts in gems, 374
  - `self`
    - class methods and, 309
    - as default object in method calls, 75
    - knowing value of during class definition, 330–331
  - semicolon (`;`), for separating statements in Ruby code, 10–11
  - set
    - regular expression for matching any one of a bunch of characters, 55–56
    - using asterisk (`*`) in conjunction with, 58
  - set class
    - dynamic typing and, 95–96
    - mapping boolean operators to union and intersection operations, 135
  - `set_trace_func` hook, 256–257
  - `setup` method, `Test::Unit`, 100
  - `shift` method, for arrays, 37
  - shoulda gem, utilities for `Test::Unit`, 108
  - `silence` method, for turning logging off, 231
  - `SimpleDelegator` class, 281–282
  - simplicity, as solution to code complexity, 92
  - single quotes (`'`), use with string literals, 44–45
  - single-exit approach, to writing methods, 123–126
  - singleton classes
    - adding convenience to class instance variables, 176–177
    - class methods, 162–165
    - visibility of, 160–161
  - singleton methods
    - alternative syntax for, 160
    - avoiding trouble when using, 165–167
    - class methods, 162–165

- defining, 158, 163–164
- extending modules and, 198
- invisibility of singleton class, 160–161
- overriding class-defined methods, 159–160
- overview of, 157–158
- review of, 167
- software
  - resources for building software with Ruby, 394
  - trade offs in software engineering, 335
- sort method
  - <=> operator and, 214
  - for arrays, 37
- source code, for Ruby projects, 394
- sow command, generating directory structure of
  - gems, 377
- spec command
  - running specifications with, 103–104
  - specifying executable scripts in gems, 374
- specs. *see also* tests
  - MiniSpec, 110
  - mocks and, 107–108
  - overview of, 103
  - RubySpec project, 109–110
  - running with spec command, 103–104
  - stubs and, 105–107
  - tidy and readable, 104–105
  - when to write, 113
- splat, for star jargon, 22
- split method, working with strings, 48
- square brackets. *see* [ ] (square brackets)
- squish! method, adding to String class, 301–302
- static typing
  - adding type-checking code to methods and, 91
  - bulkier code with, 89
  - dangers of dynamic typing and, 93
  - overview of, 85
- StringIO class, comparing with File class,
  - 94–95
- strings. *see also* regular expressions
  - adding methods to string class, 300–302
  - APIs supported, 47–49
  - converting symbols to/from, 69
  - formatting operator (%) for, 137–138
  - indexing, 52
  - inflection rules based on gsub, 50–51
  - mutability of, 51–52
  - as objects, 76
  - optimizing string class for data processing, 67
  - options for writing, 44–46
  - overview of, 43
  - pattern matching regular expressions against
    - strings, 150
  - review of, 52
  - string class, 43
  - symbols as, 65–66
  - types of thing collected in, 49–50
  - uses of, 66–67
  - when to use symbols vs. when to use strings,
    - 70–71
- strip method, 47
- stubs
  - RSpec, 105–107
  - singleton methods and, 165
- sub method, working with strings, 47–48
- subclasses
  - calling private methods from, 80
  - class instance variables and, 175–176
  - difficulty of subclassing, 319–321
  - example of paragraph subclass, 317–319
  - examples of subclass-changing methods, 327
  - hook for informing when a class gains a
    - subclass, 250–253, 257–259
  - methods in superclass unable to override
    - methods in subclasses, 200
  - no limit to modifying from superclass method,
    - 324–326
  - practical basis of, 95
- subtraction (-) operator
  - as binary or unary operator, 132
  - overloading, 131
- sudo, for running gems, 368
- superclasses
  - in inheritance tree, 193
  - methods in superclass that can add methods to
    - subclasses, 324
  - methods in superclass unable to override
    - methods in subclasses, 200
  - mixins as alternative to, 193–195
  - modules and, 198
  - no limit to modifying subclasses from
    - superclass method, 324–326
  - overview of, 75–76



swapcase method, working with strings, 47  
 switch statement, case statement compared with, 21  
 symbols  
   compared with strings, 65–66  
   confusing nature of, 69–70  
   converting strings to/from, 69  
   as hash keys, 68–69  
   immutability of, 68  
   not supporting singleton methods, 159  
   as objects, 76  
   overview of, 65  
   review of, 71  
   single instance of, 67–68  
   using strings as symbolic markers, 66–67  
   when to use symbols vs. when to use strings, 70–71  
 symmetry principal, double-equals (==) operator and, 146–147

## T

tabs, Ruby indentation conventions and, 5–6  
 TAR files, 370  
 teardown method, Test::Unit, 100  
 ternary operator (:), in expression-based decision making, 26  
 test directory, sow command generating, 377  
 test-first development, 113  
 tests  
   applying to gems, 380  
   assertions in Test::Unit, 101  
   automated testing for resolving bugs, 94  
   limitations of Test::Unit, 101–102  
   MiniTest, 110  
   mocha utilities for Test::Unit, 109  
   mocks and, 107–108  
   overview of, 97  
   qualities of good tests, 110–113  
   review of, 113  
   RSpec testing framework, 102–104  
   RSpec project, 109–110  
   shoulda gem utilities for Test::Unit, 108  
   stubs and, 105–107

  tidy and readable specs, 104–105  
   when to write, 113  
 Test::Unit  
   at\_exit hook used in, 259–260  
   assertions in, 101  
   limitations of, 101–102  
   mocha utilities for, 109  
   overview of, 98–100  
   shoulda gem utilities for, 108  
 text processing, strings and, 43  
 TextCompressor class, 119  
 time zones, regular expression for offsetting, 62–63  
 time.rb, regular expressions and, 62–63  
 times method, iterators, 211–212  
 to\_s method  
   of Object class, 77–78  
   turning symbols into strings, 69  
 to\_sym method, turning strings into symbols, 69  
 transitive property, of double-equals (==) operator, 147–149  
 Treetop  
   for building parsers, 358–360  
   examples of external DSLs, 364  
 triple equals operator (===), for case statements, 23, 149–150  
 true, as an object, 76  
 two space rule, Ruby indentation convention, 5–6  
 type declaration  
   documentation and, 92  
   dynamic typing. *see* dynamic typing  
   static typing. *see* static typing  
 type-checking code, disadvantages of adding, 91

## U

unary operators, 131–132, 134  
 union operations, mapping boolean operators to, 135  
 unique\_index\_of method, 120  
 unit tests. *see also* Test::Unit  
   minimum tests, 112–113  
   speed as factor in, 110

Unix, 370

unless statements

- example of use of, 18–19
- modifier forms of, 20

until statements

- comments, 6
- example of use of, 19
- modifier forms of, 20

uppercase method, working with strings, 47

URIs

- using class variables with, 177–178
- using modules with, 191

user interfaces, 285. *see also* APIs

## V

values

- case statement returning, 22
- code blocks returning, 208–209

variables

- adding module variables to modules, 178–179
- attaching instance variable to class objects (class instance variables), 174–175
- class variables. *see* class variables
- documenting declaration of, 92
- initializing, 26
- modules and, 186
- naming, 8
- open classes and, 294
- referencing objects with, 77

VCIS (Version Conflict Induced Insanity), 370

versions

- Ruby implementations and, 381–382
- versioning support in gems, 368–369

vertical bar (|), in syntax of alternatives in regular expressions, 56–57

visibility, of methods, 79–81

## W

while statements

- code capturing values of, 25
- example of use of, 19
- modifier forms of, 20

whiny nil facility, Rails, 268–269

white space, managing in strings, 47

with\_logging methods

- capturing return values, 228
- managing logging with, 222
- passing arguments and, 227

## X

XML

- accessing/manipulating data in, 336–337
- building parser for, 353–356
- creating reader for, 251–252
- processing in Ruby with REXML, 337–339
- xmlRipper class for writing XML processing scripts, 340–341

xmlRipper class

- building parser for XML processing language, 354
- transition from API to DSL and, 341–344
- for writing XML processing scripts, 340

XPath, 338–339, 344

XSLT, 337

XUnit testing frameworks, 98

## Y

YAML

- compared with XML, 250
- example of use of modules in, 191

YARV

- overview of, 385–387
- support and documentation, 389

yield, firing code blocks, 233