

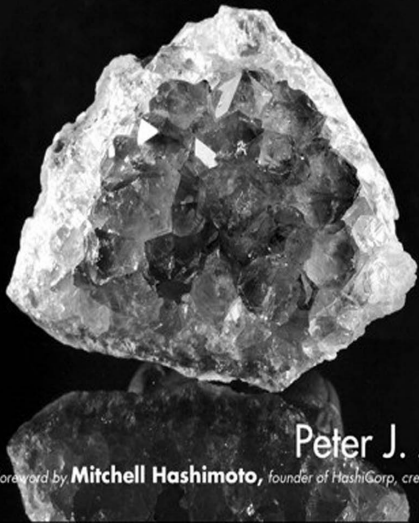
Effective SOFTWARE DEVELOPMENT SERIES

Scott Meyers, Consulting Editor



Effective RUBY

48 Specific Ways to Write Better Ruby



Peter J. Jones

Foreword by **Mitchell Hashimoto**, founder of HashiCorp, creator of Vagrant

FREE SAMPLE CHAPTER

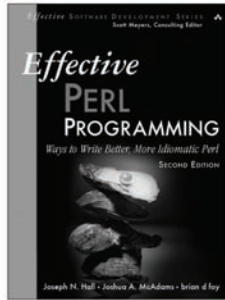
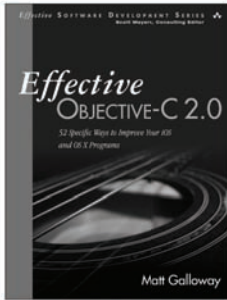
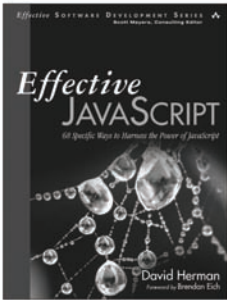
SHARE WITH OTHERS



Effective Ruby

The Effective Software Development Series

Scott Meyers, Consulting Editor



◆◆ Addison-Wesley

Visit informit.com/esds for a complete list of available publications.

The **Effective Software Development Series** provides expert advice on all aspects of modern software development. Titles in the series are well written, technically sound, and of lasting value. Each describes the critical things experts always do — or always avoid — to produce outstanding software.

Scott Meyers, author of the best-selling books *Effective C++* (now in its third edition), *More Effective C++*, and *Effective STL* (all available in both print and electronic versions), conceived of the series and acts as its consulting editor. Authors in the series work with Meyers to create essential reading in a format that is familiar and accessible for software developers of every stripe.



Make sure to connect with us!
informit.com/socialconnect

informit.com
the trusted technology learning source

◆◆ Addison-Wesley

Safari
Books Online

Effective Ruby

48 Specific Ways to Write Better Ruby

Peter J. Jones

◆ 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.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the United States, please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Jones, Peter J., author.

Effective Ruby : 48 specific ways to write better Ruby / Peter J. Jones.

Pages cm

Includes index.

ISBN 978-0-13-384697-3 (pbk. : alk. paper)

1. Ruby (Computer program language) 2. Object-oriented programming (Computer science) I. Title.

QA76.73.R83J66 2015

005.13'3—dc23

2014026572

Copyright © 2015 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. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-13-384697-3

ISBN-10: 0-13-384697-0

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

First printing, September 2014

Editor-in-Chief

Mark L. Taub

Senior Acquisitions Editor

Trina MacDonald

Development Editor

Songlin Qiu

Managing Editor

John Fuller

Full-Service Production Manager

Julie B. Nahil

Copy Editor

Christina Edwards

Indexer

Jack Lewis

Proofreader

Andrea Fox

Technical Reviewers

Bruce Williams

Bobby Wilson

Editorial Assistant

Olivia Basegio

Cover Designer

Chuti Prasertsith

Compositor

LaurelTech

For Shanna, the reason my life has balance and purpose.

This page intentionally left blank

Contents

Foreword	xi
Preface	xiii
Acknowledgments	xvii
About the Author	xix
Chapter 1: Accustoming Yourself to Ruby	1
Item 1: Understand What Ruby Considers to Be True	1
Item 2: Treat All Objects as If They Could Be nil	3
Item 3: Avoid Ruby’s Cryptic Perlisms	6
Item 4: Be Aware That Constants Are Mutable	9
Item 5: Pay Attention to Run-Time Warnings	12
Chapter 2: Classes, Objects, and Modules	17
Item 6: Know How Ruby Builds Inheritance Hierarchies	17
Item 7: Be Aware of the Different Behaviors of super	24
Item 8: Invoke super When Initializing Subclasses	28
Item 9: Be Alert for Ruby’s Most Vexing Parse	31
Item 10: Prefer Struct to Hash for Structured Data	35
Item 11: Create Namespaces by Nesting Code in Modules	38
Item 12: Understand the Different Flavors of Equality	43
Item 13: Implement Comparison via “<=>” and the Comparable Module	49
Item 14: Share Private State through Protected Methods	53
Item 15: Prefer Class Instance Variables to Class Variables	55

Chapter 3: Collections	59
Item 16: Duplicate Collections Passed as Arguments before Mutating Them	59
Item 17: Use the Array Method to Convert nil and Scalar Objects into Arrays	63
Item 18: Consider Set for Efficient Element Inclusion Checking	66
Item 19: Know How to Fold Collections with reduce	70
Item 20: Consider Using a Default Hash Value	74
Item 21: Prefer Delegation to Inheriting from Collection Classes	79
Chapter 4: Exceptions	85
Item 22: Prefer Custom Exceptions to Raising Strings	85
Item 23: Rescue the Most Specific Exception Possible	90
Item 24: Manage Resources with Blocks and ensure	94
Item 25: Exit ensure Clauses by Flowing Off the End	97
Item 26: Bound retry Attempts, Vary Their Frequency, and Keep an Audit Trail	100
Item 27: Prefer throw to raise for Jumping Out of Scope	104
Chapter 5: Metaprogramming	107
Item 28: Familiarize Yourself with Module and Class Hooks	107
Item 29: Invoke super from within Class Hooks	114
Item 30: Prefer define_method to method_missing	115
Item 31: Know the Difference between the Variants of eval	122
Item 32: Consider Alternatives to Monkey Patching	127
Item 33: Invoke Modified Methods with Alias Chaining	133
Item 34: Consider Supporting Differences in Proc Arity	136
Item 35: Think Carefully Before Using Module Prepending	141
Chapter 6: Testing	145
Item 36: Familiarize Yourself with MiniTest Unit Testing	145
Item 37: Familiarize Yourself with MiniTest Spec Testing	149
Item 38: Simulate Determinism with Mock Objects	152
Item 39: Strive for Effectively Tested Code	156

Chapter 7: Tools and Libraries	163
Item 40: Know How to Work with Ruby Documentation	163
Item 41: Be Aware of IRB's Advanced Features	166
Item 42: Manage Gem Dependencies with Bundler	170
Item 43: Specify an Upper Bound for Gem Dependencies	175
Chapter 8: Memory Management and Performance	179
Item 44: Familiarize Yourself with Ruby's Garbage Collector	179
Item 45: Create Resource Safety Nets with Finalizers	185
Item 46: Be Aware of Ruby Profiling Tools	189
Item 47: Avoid Object Literals in Loops	195
Item 48: Consider Memoizing Expensive Computations	197
Epilogue	201
Index	203

This page intentionally left blank

Foreword

When I was asked to do a technical review and write the foreword for a book on Ruby, I was skeptical. Several Ruby books already exist that run the gamut from beginner to advanced Ruby VM internals. I thought, “What could *another* Ruby book offer?” But I agreed to look over the text, and to my surprise a great and novel book about Ruby was laid out before my eyes. This book is quite unlike any other Ruby book, and in a couple hundred pages, I imagine anyone who reads this—novice or expert—will emerge a better Ruby programmer.

Ruby as a language has matured a lot in the past decade, when I got started with it. Early on, there was the hype phase, when Ruby was touted as the end-all and be-all of languages. Then emerged the proliferation of libraries, when it felt like libraries were being abandoned and re-created daily, and none could be trusted to remain up to date. Then other “new hotness” languages started emerging, and Ruby went through a phase of being seen as the language of yesteryear. But now, finally, Ruby is seen as a practical, effective language for solving many problems, although it understandably won’t solve all of them. (You’re not going to be writing the next big operating system in Ruby.)

Instead of being a book that covers basic syntax or advanced practices, this book masterfully walks the line of introducing real-world best practices for writing Ruby applications that won’t crash, will be maintainable, and will be fast. It is a book that every Ruby programmer should read. Beginners should learn best practices to better understand the language, and experienced developers should double-check their own practices and maybe learn a couple of new ones as well.

The book is written in my favorite way: lots of examples, and the examples don’t just explain “What?” and “How?”, but also “Why?” Although these are best practices curated from the Ruby community

over years of maturation, it is important to always remain skeptical, ask questions, and perhaps find new best practices that improve on the old.

With that, I wish you a fun journey through this book, and fully expect you to grow as a Ruby programmer in just a couple hundred pages.

—*Mitchell Hashimoto, founder and CEO of HashiCorp, creator of Vagrant*

Preface

Learning a new programming language usually happens in two phases. During the first phase you spend time learning the syntax and structure of the language. This phase is often short when you have previous experience learning new programming languages. In the case of Ruby, the syntax is very familiar to those who have experience with other object-oriented languages. The structure of the language—how you build programs out of the syntax—should also be very familiar to experienced programmers.

The second phase, on the other hand, can take a bit more work. This is when you dig deeper into the language and learn its idioms. Most languages have a unique way of solving common problems, and Ruby is no different. For example, Ruby uses blocks along with the iterator pattern instead of explicit looping. Learning how to solve problems “the Ruby way” while avoiding any sharp edges is what this phase is all about.

And that’s what this book is all about, too. But it’s not an introductory book. I assume you’ve already completed the first phase of learning Ruby—that is, learning its syntax and structure. My goal with this book is to take you deeper into Ruby. I want to show you how to get the most out of the language and how to write effective code that is more reliable and easier to maintain. Along the way we’ll also explore how parts of the Ruby interpreter work internally, knowledge that will allow you to write more efficient programs.

Ruby Implementations and Versions

As you know, Ruby has a very active community of contributors. They work on all sorts of projects, including different implementations of the Ruby interpreter. Besides the official Ruby implementation (colloquially known as MRI), there are several others to choose from. Need to deploy your Ruby application to production servers that

are already configured for running Java applications? No problem, that's what JRuby is for. How about the opposite end of the spectrum, Ruby applications targeting smartphones and tablets? There's a Ruby implementation for that, too.

Having several Ruby implementations to choose from is a good sign that Ruby is alive and well. Obviously, each of them has a unique, internal implementation. But from the perspective of a programmer writing Ruby code, the various interpreters behave closely enough to MRI that you won't have to worry much.

The vast majority of this book applies equally to all of the various Ruby implementations. The only exceptions are the items that describe Ruby internal details such as how the garbage collector works. In those cases, I will narrow my focus to the official Ruby implementation, MRI. You'll know when we're talking about MRI specifically when I mention specific Ruby versions in the text.

Speaking of specific versions, all of the code in this book has been tested with Ruby 1.9.3 and later. At the time of writing, Ruby 2.1 was the most recent version, with 2.2 just around the corner. When I don't mention a specific version of Ruby in the text, the example code will work on all supported versions.

A Note about Style

Ruby programmers have, for the most part, converged on a single way of formatting their Ruby code. There are even a handful of Ruby-Gems that can inspect your code and scold you when it isn't formatted according to a set of predetermined styling rules. I bring this up because the style I've chosen for the example code in this book deviates slightly from what might be considered normal.

When I call a method and supply arguments to it, I use parentheses around the arguments, without any space between the opening parenthesis and the name of the method. Out in the wild, it's common to see method calls *without* parentheses, probably because Ruby doesn't require them. But as we'll see in Chapter 1, omitting parentheses in some situations can cause ambiguities in your code, which in turn forces Ruby to guess what you mean. Because of these ambiguities, I think it's a bad habit to omit parentheses and one the community needs to wean itself from.

The other reason I use parentheses is to clearly show when an identifier is a method call versus a keyword. You might be surprised to learn that things you thought were keywords are actually method calls (e.g., `require`). Using parentheses helps illustrate this.

While we're on the topic of style, I should mention that throughout this book, when I mention methods, I use RI notation. If you're not familiar with RI notation it's easy to learn and can be very helpful. Its primary purpose is to differentiate between class and instance methods. When referring to class methods I'll write the name of the class and the method separated by two colons ("::"). For example, `File::open` is the `open` class method from the `File` class. Likewise, instance methods are written with a number sign ("#") between the class name and the name of the instance method (e.g., `Array#each`). The same is true of module methods (e.g., `GC::stat`) and module instance methods (e.g., `Enumerable#grep`). Item 40 goes into more detail about RI notation and how to use it to look up method documentation. But this little primer is enough to get you started.

Where to Get the Source Code

Throughout the book we'll examine many listings of example source code. To make it easier to digest, code will often be broken up into small chunks that we'll work through one at a time. There are even times when unimportant details are omitted. Sometimes it's nice to see all the code at once, to get the bigger picture, so to speak. All of the code shown can be found at the website for the book at <http://effectiveruby.com>.

This page intentionally left blank

Acknowledgments

Writing something that people are willing to spend their time reading, and something worthy of spending their money on, isn't a solo endeavor. As a matter of fact, beyond the close group of people who were directly involved in this book, many others unknowingly contributed in one way or another. For example, my friend Michael Garriss had no idea what the consequences of his actions would be when he sweet-talked me into learning Ruby. He certainly didn't expect for me to drag him from company to company, introducing Ruby at each step of the way. Nevertheless, that's what happened.

It might be a bit unorthodox (and vague) but I want to thank everyone who's ever contributed their time and creativity to the open-source community. Every single tool I used while writing this book, even those I created specifically for it, are open-source projects. There's no way I could have written this book without being able to review the source code to the Ruby interpreter and the handful of gems that are discussed. I spent many hours curled up with code, dissecting, experimenting, and sometimes crying. The fact that I was able to do that is wonderful all by itself.

Of course, without the generosity of those who volunteered to work with me, this book wouldn't be worthy of your attention. Several people gave up their free time to review early drafts of the chapters and provided me with immensely useful feedback. Isaac Foraker, Timothy Clayton, and my wife, Shanna Jones, spent many hours reading text and experimenting with code. Thank you so much for your time.

Probably not realizing what they were getting themselves into, Bruce Williams and Bobby Wilson agreed to be technical reviewers. They both helped me improve the examples in the book and the explanations that go along with them. They also encouraged me when the anxiety of having someone else poking around my work was a bit overwhelming.

xviii Acknowledgments

Everyone at Pearson made things as easy for me as they could. Trina MacDonald, Olivia Basegio, and Songlin Qiu were extremely patient with me and shaped this book into what it is now. I've grown so much during this project and a large part of that is due to them.

Scott Meyers is a hero of mine and being able to work with him is like having a dream come true. In the late 1990s I came across Scott's book, *Effective C++*, and it changed the way I approached programming. It also inspired me to teach others the things I have learned. Sending my work to Scott for review was terrifying, but Scott was nothing less than encouraging and extremely helpful. Thank you, Scott.

My wife, Shanna Jones, was a huge source of encouragement and understanding. Knowing that it would take me away from her, she pushed me to write this book anyway. Shanna, you've taught me more than you understand. Thank you for always supporting me.

About the Author

Peter J. Jones has been working professionally with Ruby since 2005. He began programming before he learned how to use a keyboard properly after stumbling upon a Commodore 64, a few code listings, and some cassette tapes. Peter is a freelance software engineer and a senior instructor for programming related workshops taught by Devalot.com.

This page intentionally left blank

1

Accustoming Yourself to Ruby

With each programming language you learn, it's important to dig in and discover its idiosyncrasies. Ruby is no different. While it borrows heavily from the languages that preceded it, Ruby certainly has its own way of doing things. And sometimes those ways will surprise you.

We begin our journey through Ruby's many features by examining its unique take on common programming ideas. That is, those that impact every part of your program. With these items mastered, you'll be prepared to tackle the chapters that follow.

Item 1: Understand What Ruby Considers to Be True

Every programming language seems to have its own way of dealing with Boolean values. Some languages only have a single representation of true or false. Others have a confusing blend of types that are sometimes true and sometimes false. Failure to understand which values are true and which are false can lead to bugs in conditional expressions. For example, how many languages do you know where the number zero is false? What about those where zero is true?

Ruby has its own way of doing things, Boolean values included. Thankfully, the rule for figuring out if a value is true or false is pretty simple. It's different than other languages, which is the whole reason this item exists, so make sure you understand what follows. In Ruby, every value is true *except* false and nil.

It's worth taking a moment and thinking about what this means. While it's a simple rule, it has some strange consequences when compared with other mainstream languages. In a lot of programming languages the number zero is false, with all other numbers being true. Using the rule just given for Ruby, zero is *true*. That's probably one of the biggest gotchas for programmers coming to Ruby from other languages.

Another trick that Ruby plays on you if you're coming from another language is the assumption that `true` and `false` are keywords. They're not. In fact, they're best described as global variables that don't follow the naming and assignment rules. What I mean by this is that they don't begin with a "\$" character, like most global variables, and they can't be used as the left-hand side of an assignment. But in all other regards they're global variables. See for yourself:

```
irb> true.class
--> TrueClass
```

```
irb> false.class
--> FalseClass
```

As you can see, `true` and `false` act like global objects, and like any object, you can call methods on them. (Ruby also defines `TRUE` and `FALSE` constants that reference these `true` and `false` objects.) They also come from two different classes: `TrueClass` and `FalseClass`. Neither of these classes allows you to create new objects from them; `true` and `false` are all we get. Knowing the rule Ruby uses for conditional expressions, you can see that the `true` object only exists for convenience. Since `false` and `nil` are the only false values, the `true` object is superfluous for representing a true value. Any non-`false`, non-`nil` object can do that for you.

Having two values to represent false and all others to represent true can sometimes get in your way. One common example is when you need to differentiate between `false` and `nil`. This comes up all the time in objects that represent configuration information. In those objects, a `false` value means that something should be disabled, while a `nil` value means an option wasn't explicitly specified and the default value should be used instead. The easiest way to tell them apart is by using the `nil?` method, which is described further in Item 2. Another way is by using the `==` operator with `false` used as the left operand:

```
if false == x
  ...
end
```

With some languages there's a stylistic rule that says you should always use immutable constants as the left-hand side of an equality operator. That's not why I'm recommending `false` as the left operand to the `==` operator. In this case, it's important for a functional reason. Placing `false` on the left-hand side means that Ruby parses the expression as a call to the `FalseClass#==` method (which comes from the `Object` class). We can rest safely knowing this method only returns `true` if the right operand is also the `false` object. On the other

hand, using `false` as the *right* operand may not work as expected since other classes can override the `Object#==` method and loosen the comparison:

```
irb> class Bad
      def == (other)
        true
      end
    end
```

```
irb> false == Bad.new
---> false
irb> Bad.new == false
---> true
```

Of course, something like this would be pretty silly. But in my experience, that means it's more likely to happen. (By the way, we'll cover the “`==`” operator more in Item 12.)

Things to Remember

- ◆ Every value is true *except* `false` and `nil`.
- ◆ Unlike in a lot of languages, the number zero is true in Ruby.
- ◆ If you need to differentiate between `false` and `nil`, either use the `nil?` method or use the “`==`” operator with `false` as the left operand.

Item 2: Treat All Objects as If They Could Be nil

Every object in a running Ruby program comes from a class that, in one way or another, inherits from the `BasicObject` class. Imagining how all these objects relate to one another should conjure up the familiar tree diagram with `BasicObject` at the root. What this means in practice is that an object of one class can be substituted for an object of another (thanks to polymorphism). That's why we can pass an object that *behaves* like an array—but is not actually an array—to a method that expects an `Array` object. Ruby programmers like to call this “duck typing.” Instead of requiring that an object be an instance of a specific class, duck typing shifts the focus to what the object can do; in other words, interface over type. In Ruby terms, duck typing means you should prefer using the `respond_to?` method over the `is_a?` method.

But in reality, it's rare to see a method inspect its arguments using `respond_to?` to make sure it supports the correct interface. Instead, we tend to just invoke methods on an object and if the object doesn't respond to a particular method, we leave it up to Ruby to raise a

`NoMethodError` exception at run time. On the surface, it seems like this could be a real problem for Ruby programmers. Well, just between you and me, it is. It's one of the core reasons testing is so very important. There's nothing stopping you from accidentally passing a `Time` object to a method expecting a `Date` object. These are the kinds of mistakes we have to tease out with good tests. And thanks to testing, these types of problems can be avoided. But one of these polymorphic substitutions plagues even well-tested applications:

```
undefined method 'fubar' for nil:NilClass (NoMethodError)
```

This is what happens when you call a method on an object and it turns out to be that pesky `nil` object...the one and only object from the `NilClass` class. Errors like this tend to slip through testing only to show up in production when a user does something out of the ordinary. Another situation where this can occur is when a method returns `nil` and then that return value gets passed directly into another method as an argument. There's a surprisingly large number of ways `nil` can unexpectedly get introduced into your running program. The best defense is to assume that any object might actually be the `nil` object. This includes arguments passed to methods and return values from them.

One of the easiest ways to avoid invoking methods on the `nil` object is by using the `nil?` method. It returns `true` if the receiver is `nil` and `false` otherwise. Of course, `nil` objects are always `false` in a Boolean context, so the `if` and `unless` expressions work as expected. All of the following lines are equivalent to one another:

```
person.save if person
person.save if !person.nil?
person.save unless person.nil?
```

It's often easier to explicitly convert a variable into the expected type rather than worry about `nil` all the time. This is especially true when a method should produce a result even if some of its inputs are `nil`. The `Object` class defines several conversion methods that can come in handy in this case. For example, the `to_s` method converts the receiver into a string:

```
irb> 13.to_s
----> "13"

irb> nil.to_s
----> ""
```

As you can see, `NilClass#to_s` returns an empty string. What makes `to_s` really nice is that `String#to_s` simply returns `self` without performing any conversion or copying. If a variable is already a string then using `to_s` will have minimal overhead. But

if nil somehow winds up where a string is expected, `to_s` can save the day. As an example, suppose a method expects one of its arguments to be a string. Using `to_s`, you can hedge against that argument being nil:

```
def fix_title (title)
  title.to_s.capitalize
end
```

The fun doesn't stop there. As you'd expect, there's a matching conversion method for almost all of the built-in classes. Here are some of the more useful ones as they apply to nil:

```
irb> nil.to_a
--> []
```

```
irb> nil.to_i
--> 0
```

```
irb> nil.to_f
--> 0.0
```

When multiple values are being considered at the same time, you can make use of a neat trick from the Array class. The `Array#compact` method returns a copy of the receiver with all nil elements removed. It's common to use it for constructing a string out of a set of variables that might be nil. For example, if a person's name is made up of first, middle, and last components—any of which might be nil—you can construct a complete full name with the following code:

```
name = [first, middle, last].compact.join(" ")
```

The nil object has a tendency to sneak into your running programs when you least expect it. Whether it's from user input, an unconstrained database, or methods that return nil to signal failure, always assume that every variable could be nil.

Things to Remember

- ◆ Due to the way Ruby's type system works, any object can be nil.
- ◆ The `nil?` method returns true if its receiver is nil and false otherwise.
- ◆ When appropriate, use conversion methods such as `to_s` and `to_i` to coerce nil objects into the expected type.
- ◆ The `Array#compact` method returns a copy of the receiver with all nil elements removed.

Item 3: Avoid Ruby's Cryptic Perlisms

If you've ever used the Perl programming language then you undoubtedly recognize its influence on Ruby. The majority of Ruby's perlisms have been adopted in such a way that they blend perfectly with the rest of the ecosystem. But others either stick out like an unnecessary semicolon or are so obscure that they leave you scratching your head trying to figure out how a particular piece of code works.

Over the years, as Ruby matured, alternatives to some of the more cryptic perlisms were added. As more time went on, some of these holdovers from Perl were deprecated or even completely removed from Ruby. Yet, a few still remain, and you're likely to come across them in the wild. This item can be used as a guide to deciphering those perlisms while acting as a warning to avoid introducing them into your own code.

The corner of Ruby where you're most likely to encounter features borrowed from Perl is a set of cryptic global variables. In fact, Ruby has some pretty liberal naming rules when it comes to global variables. Unlike with local variables, instance variables, or even constants, you're allowed to use all sorts of characters as variable names. Recalling that global variables begin with a "\$" character, consider this:

```
def extract_error (message)
  if message =~ /^ERROR:\s+(.+)$/
    $1
  else
    "no error"
  end
end
```

There are two perlisms packed into this code example. The first is the use of the "=~" operator from the String class. It returns the position within the string where the right operand (usually a regular expression) matches, or nil if no match can be found. When the regular expression matches, several global variables will be set so you can extract information from the string. In this example, I'm extracting the contents of the first capture group using the \$1 global variable. And this is where things get a bit weird. That variable might look and smell like a global variable, but it surely doesn't act like one.

The variables created by the "=~" operator are called *special* global variables. That's because they're scoped *locally* to the current thread and method. Essentially, they're local values with global names. Outside of the extract_error method from the previous example, the \$1 "global" variable is nil, even after using the "=~" operator. In the example,

returning the value of the \$1 variable is just like returning the value of a local variable. The whole situation can be confusing. The good news is that it's completely unnecessary. Consider this alternative:

```
def extract_error (message)
  if m = message.match(/^ERROR:\s+(.+)$/ )
    m[1]
  else
    "no error"
  end
end
```

Using `String#match` is much more idiomatic and doesn't use any of the special global variables set by the `"=~"` operator. That's because the `match` method returns a `MatchData` object (when the regular expression matches) and it contains all of the same information that was previously available in those special global variables. In this version of the `extract_error` method, you can see that using the index operator with a value of 1 gives you the same string that \$1 would have given you in the previous example. The bonus feature is that the `MatchData` object is a plain old local variable and you get to choose the name of it. (It's fairly common to make an assignment inside the conditional part of an `if` expression like this. That said, it's all too easy to use `"="` when you really meant `"=="`. Watch out for these kinds of mistakes.)

Besides those set by the `"=~"` operator, there are other global variables borrowed from Perl. The one you're most likely to see is `$:`, which is an array of strings representing the directories where Ruby will search for libraries that are loaded with the `require` method. Instead of using the `$:` global variable, you should use its more descriptive alias: `$LOAD_PATH`. As a matter of fact, there are more descriptive versions for all of the other cryptic global variables such as `$;` and `$/`. But there's a catch. Unlike with `$LOAD_PATH`, you have to load a library to access the other global variables' aliases:

```
require('English')
```

Once the `English` library is loaded, you can replace all those strange global variables by their longer, more descriptive aliases. For a full list of these aliases, take a look at the documentation for the `English` module.

There's one last perlism you should be aware of. Not surprisingly, it also has something to do with a global variable. Consider this:

```
while readline
  print if ~ /^ERROR:/
end
```

If you think this code is a bit obfuscated, then congratulations, you're in good company. You might be wondering what the `print` method is actually printing and what that regular expression is matching against. It just so happens that all of the methods in this example are working with a global variable—the `$_` variable to be more precise.

So, what's going on here? It all starts with the `readline` method. More specifically, it's the `Kernel#readline` method. (In Item 6, we'll dig more into how Ruby determines that, in this context, `readline` comes from the `Kernel` module.) This version of `readline` is a little different from its counterpart in the `IO` class. You can probably gather that it reads a line from standard input and returns it. The subtle part is that it also stores that line of input in the `$_` variable. (`Kernel#gets` does the same thing but doesn't raise an exception when the end-of-file marker is reached.) In a similar fashion, if `Kernel#print` is called without any arguments, it will print the contents of the `$_` variable to standard output.

You can probably guess what that unary `~` operator and the regular expression are doing. The `Regexp#~` operator tries to match the contents of the `$_` variable against the regular expression to its right. If there's a match, it returns the position of the match; otherwise, it returns `nil`. While all these methods might look like they are somehow magically working together, you now know that it's all thanks to the `$_` global variable. But why does Ruby even support this?

The only legitimate use for these methods (and the `$_` variable) is for writing short, simple scripts on the command line, so-called “one liners.” This allows Ruby to compete with tools such as Perl, `awk`, and `sed`. When you're writing real code you should avoid methods that implicitly read from, or write to, the `$_` global variable. These include other similar `Kernel` methods I haven't listed here such as `chomp`, `sub`, and `gsub`. The difference with those is that they can no longer be used in recent versions of Ruby without using either the `-n` or the `-p` command-line option to the Ruby interpreter. That is, it's like these methods don't even exist without one of those command-line options. That's a good thing.

Now you can see how some of the more cryptic perlisms can affect the readability, and thus maintainability, of your code. Especially those obscure global variables and the ones that are global in name only. It is best to use the more Ruby-like methods (`String#match` vs. `String#=~`) and the longer, more descriptive names for global variables (`$LOAD_PATH` vs. `$:`).

Things to Remember

- ◆ Prefer `String#match` to `String#=~`. The former returns all the match information in a `MatchData` object instead of several special global variables.
- ◆ Use the longer, more descriptive global variable aliases as opposed to their short cryptic names (e.g., `$LOAD_PATH` instead of `$:`). Most of the longer names are only available after loading the English library.
- ◆ Avoid methods that implicitly read from, or write to, the `$_` global variable (e.g., `Kernel#print`, `Regexp#=~`, etc.).

Item 4: Be Aware That Constants Are Mutable

If you're coming to Ruby from another programming language, there's a good chance that constants don't behave the way you expect them to. But before we dig into that let's review what Ruby considers to be a constant.

When you first learned Ruby you were probably taught that constants are identifiers that are made up of uppercase alphanumeric characters and underscores. Some examples include `STDIN`, `ARGV`, and `RUBY_VERSION`. But that's not the entire story. In reality, a constant is any identifier that begins with an uppercase letter. This means that identifiers like `String` and `Array` are also constants. That's right...the names of classes and modules are actually constants in Ruby. With that in mind, let's take a closer look at how constants differ from other variable-like things in Ruby.

As their name suggests, constants are meant to remain unchanged during the lifetime of a program. You might assume, therefore, that Ruby would prevent you from altering the value stored in a constant. Well, that assumption would be wrong. Consider this:

```
module Defaults
  NETWORKS = ["192.168.1", "192.168.2"]
end

def purge_unreachable (networks=Defaults::NETWORKS)
  networks.delete_if do |net|
    !ping(net + ".1")
  end
end
```

If you invoke the `purge_unreachable` method without an argument, it will accidentally mutate a constant. It will do this without so much

as a warning from Ruby. Essentially, constants are more like global variables than unchanging values. If you think about it, since class and module names are constants, and you can change a class at anytime (e.g., add methods), then the objects referenced by constants need to be mutable in Ruby. That's fine for classes and modules, but not so great for the values we actually want to be constant and immutable. Thankfully, there's a solution to this problem—the freeze method:

```
module Defaults
  NETWORKS = ["192.168.1", "192.168.2"].freeze
end
```

With this change in place, the `purge_unreachable` method will raise a `RuntimeError` exception if it tries to alter the array referenced by the `NETWORKS` constant. As a general rule of thumb, always freeze constants to prevent them from being mutated. Unfortunately, freezing the `NETWORKS` array isn't quite enough. Consider this:

```
def host_addresses (host, networks=Defaults::NETWORKS)
  networks.map {|net| net << ".#{host}"}
end
```

The `host_addresses` method will modify the elements of the `NETWORKS` array if it isn't given a second argument. While the `NETWORKS` array itself is frozen, its *elements* are still mutable. You might not be able to add or remove elements from the array, but you can surely make changes to the existing elements. So, if a constant references a collection object such as an array or hash, freeze the collection *and* its elements:

```
module Defaults
  NETWORKS = [
    "192.168.1",
    "192.168.2",
  ].map!(&:freeze).freeze
end
```

(If you happen to be using Ruby 2.1 or later you can make use of a trick from Item 47 and freeze the string literals directly. This can save you a bit of memory while keeping the elements from accidentally being mutated.)

Freezing a constant will change an obscure, hard-to-track-down bug into an exception. That's an obvious win. Unfortunately, it's still not enough. Even if you freeze the object a constant refers to, you can still

cause problems by assigning a *new* value to an existing constant. See for yourself:

```
irb> TIMEOUT = 5
---> 5
```

```
irb> TIMEOUT += 5
(irb):2: warning: already initialized constant TIMEOUT
(irb):1: warning: previous definition of TIMEOUT was here
---> 10
```

As you can see, assigning a new value to an existing constant is perfectly legal in Ruby. You can also see that Ruby produces a warning telling us that we're redefining a constant. But that's it, just a warning. Thankfully, if we take things into our own hands, we can make Ruby raise an exception if we accidentally redefine a constant. The solution is a bit clumsy, and may be too heavy-handed for some situations, but it's simple. To prevent Ruby from assigning new values to existing constants, freeze the class or module they're defined in. You may even want to structure your code so that all constants are defined in their own module, isolating the effects of the freeze method:

```
module Defaults
  TIMEOUT = 5
end
```

```
Defaults.freeze
```

There are three levels of freezing you should consider when defining constants. The first two are easy: freeze the object that the constant references and the module the constant is defined in. Those two steps prevent the constant from being mutated or assigned to. The third is a bit more complicated. We saw that if a constant references an array of strings, we need to freeze the array *and* the elements. In other words, you need to deeply freeze the object the constant refers to. Each constant will be different, just make sure it's completely frozen.

Things to Remember

- ◆ Always freeze constants to prevent them from being mutated.
- ◆ If a constant references a collection object such as an array or hash, freeze the collection *and* its elements.
- ◆ To prevent assignment of new values to existing constants, freeze the module they're defined in.

Item 5: Pay Attention to Run-Time Warnings

Ruby programmers enjoy a shortened feedback loop while writing, executing, and testing code. Being interpreted, the compilation phase isn't present in Ruby. Or is it? If you think about it, Ruby must do some of the same things a compiler does, such as parsing our source code. When you give your Ruby code to the interpreter, it has to perform some compiler-like tasks before it starts to execute the code. It's useful to think about Ruby working with our code in two phases: compile time and run time.

Parsing and making sense of our code happens at compile time. Executing that code happens at run time. This distinction is especially important when you consider the various types of warnings that Ruby can produce. Warnings emitted during the compilation phase usually have something to do with syntax problems that Ruby was able to work around. Run-time warnings, on the other hand, can indicate sloppy programming that might be the source of potential bugs. Paying attention to these warnings can help you fix mistakes before they become real problems. Before we talk about how to enable the various warnings in Ruby, let's explore a few of the common warning messages and what causes them.

Warnings emitted during the compilation phase are especially important to pay attention to. The majority of them are generated when Ruby encounters ambiguous syntax and proceeds by choosing one of many possible interpretations. You obviously don't want Ruby guessing what you really meant. Imagine what would happen if a future version of Ruby changed its interpretation of ambiguous code and your program started behaving differently! By paying attention to these types of warnings you can make the necessary changes to your code and completely avoid the ambiguity in the first place. Here's an example of where the code isn't completely clear and Ruby produces a warning:

```
irb> "808".split /0/
warning: ambiguous first argument; put parentheses or even spaces
```

When Ruby's parser reaches the first forward slash, it has to decide if it's the beginning of a regular expression literal, or if it's the division operator. In this case, it makes the reasonable assumption that the slash starts a regular expression and should be the first argument to the `split` method. But it's not hard to see how it could also be interpreted as the division operator with the output of the `split` command being its left operand. The warning itself is generic, and only half of it is helpful. But the fix is simple enough—use parentheses:

```
irb> "808".split(/0/)
--> ["8", "8"]
```

If you send your code through Ruby with warnings enabled you're likely to see other warnings related to operators and parentheses. The reason is nearly always the same. Ruby isn't 100% sure what you mean and picks the most reasonable interpretation. But again, do you really want Ruby guessing what you mean or would you rather be completely clear from the start? Here are two more examples of ambiguous method calls that are fixed by adding parentheses around the arguments:

```

irb> dirs = ['usr', 'local', 'bin']

irb> File.join *dirs
warning: '*' interpreted as argument prefix

irb> File.join(*dirs)
--> "usr/local/bin"

irb> dirs.map &:length
warning: '&' interpreted as argument prefix

irb> dirs.map(&:length)
--> [3, 5, 3]

```

Other useful warnings during the compilation phase have to do with variables. For example, Ruby will warn you if you assign a value to a variable, but then never end up using it. This might mean you're wasting a bit of memory but could also mean you've forgotten to include a value in your calculation. You'll also receive a warning if you create two variables with the same name in the same scope, so-called *variable shadowing*. This can happen if you accidentally specify a block argument with the same name as a variable that's already in scope. Both types of variable warnings can be seen in this example:

```

irb> def add (x, y)
      z = 1
      x + y
    end
warning: assigned but unused variable - z

irb> def repeat (n, &block)
      n.times {|n| block.call(n)}
    end

warning: shadowing outer local variable - n

```

As you can see, these compile-time warnings don't necessarily mean that you've done anything wrong, but they certainly *could* mean that.

So the best course of action is to review the warnings and make changes to your source code accordingly. The same can also be said of warnings generated while your code is executing, or what I call run-time warnings. These are warnings that can only be detected after your code has done something suspicious such as accessing an uninitialized instance variable or redefining an existing method. Both of which could have been done on purpose or by accident. Like the other warnings we've seen, these are easy to remedy.

I think you get the point, so I won't enumerate a bunch of descriptive, easy-to-fix run-time warnings for you. Instead, I'd rather show you how to enable warnings in the first place. Here again, it becomes important to distinguish between compile time and run time. If you want Ruby to produce warnings about your code as it's being parsed, you need to make sure the interpreter's warning flag is enabled. That might be as easy as passing the `-w` command-line option to Ruby:

```
ruby -w script.rb
```

For some types of applications, it's not that simple. Perhaps your Ruby program is being started automatically by a web server or a background job processing server. More commonly, you're using something like Rake to run your tests and you want warnings enabled. When you can't enable warnings by giving the interpreter the `-w` command-line option, you can do it indirectly by setting the `RUBYOPT` environment variable. How you set this variable will depend on the operating system and how your application is being started. What's most important is that the `RUBYOPT` environment variable be set to `-w` within the environment where your application is going to run *before* Ruby starts.

(I should also mention that if you're using Rake to run your tests you have another option available for enabling warnings. Item 36 includes an example Rakefile that does just that.)

Now, there's one last way to enable warnings. It's poorly documented and as a result often causes a lot of confusion. Within your program you can inspect and manipulate the `$VERBOSE` global variable (and its alias, `$-w`). If you want all possible warning messages you should set this variable to true. Setting it to false lowers the verbosity (producing fewer warnings) and setting it to nil disables warnings altogether. You might be thinking to yourself, "Hey, if I can set `$VERBOSE` to true, then I don't need to mess around with this `-w` business." This is where the distinction between compile time and run time really helps.

If you don't use the `-w` command-line option with the Ruby interpreter, but instead rely upon the `$VERBOSE` variable, you won't be able

to see compile-time warnings. That's because setting the `$VERBOSE` global variable doesn't happen until your program is *running*. By that time, the parsing phase is over and you've missed all the compile-time warnings. So, there are two guidelines to follow. First, enable compile-time warnings by using the `-w` command-line option to the Ruby interpreter or by setting the `RUBYOPT` environment variable to `-w`. Second, control run-time warnings using the `$VERBOSE` global variable.

My advice is to always enable compile-time *and* run-time warnings during application development and while tests are running. If you absolutely must disable run-time warnings, do so by temporarily setting the `$VERBOSE` global variable to `nil`.

Unfortunately, enabling warning messages comes with a warning of its own. I'm disappointed to report that it's not common practice to enable warnings. So, if you're using any RubyGems and enable warnings, you're likely to get *a lot* of warnings originating from within them. This may strongly tempt you to subsequently disable warnings. Thankfully, when Ruby prints warnings to the terminal it includes the file name and line number corresponding to the warning. It shouldn't be too hard for you to write a script to filter out unwanted warnings. Even better, become a good open-source citizen and contribute fixes for any gems that are being a little sloppy and producing warnings.

Things to Remember

- ◆ Use the `-w` command-line option to the Ruby interpreter to enable compile-time and run-time warnings. You can also set the `RUBYOPT` environment variable to `-w`.
- ◆ If you must disable run-time warnings, do so by temporarily setting the `$VERBOSE` global variable to `nil`.

This page intentionally left blank

This page intentionally left blank

Index

Symbols

- “::” class path separator, 40–42
- “_” (underscore) variable feature, 168
- “||=” operator, 74–75, 197–198
- “=” (equal sign) operator, 31–34
- “=~” operator, 6–7
- “==” operator, 3, 43–48, 53
- “===” operator, 46–48
- \$: global variable, 6
- \$_ global variable, 6–9
- \$LOAD_PATH global variable, 7–9
- \$VERBOSE global variable, 14–15
- %w operator, 52, 195
- “&” operator, 72
- () (parentheses), 25–28
- “@” (at) instance variables
 - @current_user, 197–198
 - @hash. *See* @hash
 - @object, 120
 - @permissions, 63
 - @readings, 36
 - @status, 198–199
- introduction to, 55–58
- “@@” class variable, 56–58
- @hash
 - delegation and, 81–84
 - HashProxy and, 116–118
 - hook methods and, 108–110
- “~” (unary complement) operator, 140
- “<<” operator, 76–77
- “<=>” operator, 49–53

A

- Accessor method, 54
- Accumulators, 70–74
- Active Support, 128
- ActiveModel::Validations, 115
- add_dependency method, 174–178

Aliasing

- alias chaining in, 134–136
- alias_method for, 143
- eq!? in, 53
- allocate_resource method, 187
- ancestors method, 24, 142–143
- Anonymous objects, 167
- Arguments, collections passed as, 59–63
- Array, 70
- Array class
 - in global namespace, 39
 - RI for, 164
- Array methods
 - Array#compact, 5
 - Array#reject, 61
 - Array#reverse, 79–80
 - for collection classes, 63–66
- assert method, 147–148
- Assertions, 147–152
- Assignments, 32–33
- at (“@”) instance variables. *See* “@” (at) instance variables
- Audit trails, 100–103
- AuditDecorator class, 118–120
- Automation of tests, 161

B

- BasicObject#initialize, 29
- begin
 - for exceptions, 90, 94–95
 - retry and, 102–103
- Binding, 39–40, 122–123
- Blacklisting, 91
- Blocks, 71–74, 94–97
- Bound retry attempts, 100–103
- Bounds on version requirements, 175–178
- break, 100
- Bundler, 170–175

C

Caching variables, 199–200
 Callbacks, 107. *See also* Hook methods
 can? method, 67–68
 case expressions, 43–48
 catch, 105–106
 Child#initialize, 29
 Class hooks
 invoking super in, 114–115
 in metaprogramming, 107–113
 Class methods, defined, 18
 Class objects, defined, 18
 Class variables
 class instance variables vs., 55–58
 definition of, 18
 in superclasses, 57–58
 Classes. *See also specific classes*
 case equality operators in, 48
 class_eval for, 124–127
 class_exec for, 125–127
 collections. *See* Collection classes
 comparisons in, 49–53
 definition of, 17–18
 equality in, 43–48
 hooks and. *See* Class hooks
 in inheritance hierarchies, 17–24
 initializing subclasses in, 28–31
 instance variables in, 55–58
 methods in, 18
 namespaces and, 38–42
 objects in, 18
 parsing in, 31–34
 protected methods and, 53–55
 setter methods in, 31–34
 struct vs. hash in, 35–38
 super. *See* Superclasses
 super and, 24–31
 variables in. *See* Class variables
 clone method, 62–63
 close method, 188
 Cluster::Array class, 42
 Collection classes
 Array method for, 63–66
 default hash values for, 74–79
 delegation in, 79–84
 duplication of, 59–63
 element inclusion checking in, 66–70
 folding with reduce, 70–74
 freeze method for, 9–11
 inheritance in, 79–84
 introduction to, 59–63
 mutation of, 59–63
 nil and, 63–66

 passed as arguments, 59–63
 scalar objects and, 63–66
 Set, 66–70
 Color class, 44–45
 Comma-separated value (CSV) files. *See*
 CSV (comma-separated value)
 files
 Comparable module, 52–53
 Comparisons, 49–53
 Compile-time warnings, 12–15
 Constants
 finding, 40–42
 freezing vs. mutation of, 9–11
 Struct::new and, 35–38
 Containers, 59. *See also* Collection
 classes
 Control flows, 86, 104–106
 Copying objects, 62–63
 count, 182
 counter= method, 33–34
 CSV (comma-separated value) files
 constants and, 35–36
 CSV#shift for, 190–191
 Set and, 69–70
 Cucumber RubyGem, 152
 current_user, 198–200
 Custom exceptions, 85–90
 Customer class, 19–24, 132

D

-d doc, ri command-line option, 166
 Darkfish, 166
 Decorator patterns, 118–121
 Deep copies, 62–63
 def_delegators, 81–82, 108–110
 Default hash values, 74–79
 default_proc, 83
 define_method, 115–122
 define_singleton_method, 120–121
 Delays, 101–103
 Delegation, 80–84
 describe method, 151
 Determinism, 152
 Digest::SHA256 class, 139–140
 DownloaderBase, 114–115
 Duck typing, 3, 63, 117
 Dumping objects, 63
 Duplication of collections, 59–63

E

effectiveruby.com, xv, 201
 Element inclusion checking, 66–70
 Encapsulation, 53–55

Encrypt class, 41

ensure

- exiting, 97–100
- managing resources with, 94–97
- releasing resources in, 185–186, 188
- for specific exceptions, 92–93

Enumerable module

- introduction to, 59
- reduce in, 71–74

eq!? method

- hash method and, 53
- for objects, 44–48
- Version class and, 49–53

equal? method, 43–48

equal sign (“=”) operator, 31–34

Equality

- “=” operator in, 31–34
- of case, 46–47
- eq!? for. *See* eq!? method
- equal? method for, 43–48
- overview of, 43–48

Errors

- descriptions of, 86
- “error” suffix and, 86–90
- exceptions vs. *See* Exceptions

Eval variants, 122–127

Exceptions

- audit trails and, 100–103
- blocks for, 94–97
- custom, 85–90
- ensure for, 94–100
- introduction to, 85
- jumping out of scope of, 104–106
- path testing and, 158–160
- raising. *See* Raising exceptions
- raising strings vs., 85–90
- rescuing most specific, 90–93
- resource management and, 94–97
- retry attempts and, 100–103
- throw vs. raise for, 104–106

Executed vs. correct code, 160–161.

See also Testing code

Expectations, 151–154

Explicit receivers, 33–34, 54–55

extended hook, 108–113

F

-f ri, rdoc command-line option, 166

false, 1–3

File class

- file_size method and, 139
- managing resources in, 94–96

- Marshal method and, 63
- run-time warnings in, 13

FileList class, 149

Finalizers, 185–188

Fixnum

- Comparable module and, 52–53
- default hash values and, 76
- passing as value, 60–63
- Proc#arity method and, 139

Flat profiles, 190

Folding collections, 70–74

for keywords, 196

Forwardable module

- hook methods and, 108–110
- inheritance and, 81
- monkey patching and, 130
- proxies and, 116

Freezing

- constants, 9–11
- freeze method for, 79–84, 108–113
- methods for, 108–113
- string literals, 196–197

frequency method, 75

full method, 34

Fuzz testing, 158–159

FuzzBert, 158–159

G

Garbage collectors

- GC module for, 193
- GC::stat method for, 182–183
- memory and, 179–185
- performance and, 192

gem utilities, 163, 170–172

Gem dependencies

- management of, 170–175
- setting upper bounds for, 175–178

Gems

- dependencies in, 170–178
- Gemfile for, 171–176
- Gemfile.lock files for, 172–176
- gemspec method for, 174
- Kernel#gem method for, 170
- requirements for, 175
- in Ruby. *See* RubyGems
- utilities for, 163, 170–172

Generational garbage collectors, 180–181

Global constants, 40

Global namespaces, 39

Global variables, 6–9, 14–15

group method, 173

H

Happy path testing, 158–160
 has_key? 78
 “has-a” relationships, 79–80
 Hash class
 “==” operator and, 53
 Array and, 65–66, 68–70
 in collections, 59, 74–79
 delegation and, 81–84
 equality of objects and, 44–48
 hash methods and. *See* hash methods
 HashProxy class and, 116–121
 inheritance in, 80
 RaisingHash class vs., 108–110
 Set and, 68–70, 80
 struct vs., 35–38
 hash methods
 Hash#fetch, 78
 Hash::[], 65
 objects as keys and, 45–46
 overriding, 53
 Heaps, 181–185
 Hook methods, 107–113
 host_addresses method, 10–11
 HTML files, 166
 HTTP requests, 152–155
 http://effectiveruby.com, xv, 201

I

ID3 data, 171–174
 Immediate sweeping phases, 181
 Inclusion
 include? method for, 63
 include method for, 21, 142
 included hooks for, 108–113
 included_modules method for, 24
 index.html files, 166
 Inheritance
 in collections, 79–84
 delegation vs., 79–84
 hierarchies of, 17–24
 inherited class hook in, 114–115
 inherited hook/method in, 110–111
 Parent::inherited method in, 111
 PreventInheritance module in, 111
 super and, 24–28
 initialize methods
 BasicObject#initialize, 29
 Child#initialize, 29
 in collections, 67–69
 for exceptions, 89–90
 finalizers and, 187
 hash tables and, 36–37

 initialize_copy, 31, 62–63, 82–84
 Parent#initialize, 29
 RDoc and, 165
 for subclasses, 28–31
 super and, 28–31
 inject method, 59
 Instance methods
 adding to Object class, 167–168
 definition of, 18
 instance_eval, 123–127
 instance_exec, 125–127
 InstanceMethodWatcher, 112
 Instance variables
 accessor methods for, 54
 class, 55–58
 definition of, 18
 Instances of classes, defined, 18
 Interactive Ruby Shell (IRB). *See* IRB
 (Interactive Ruby Shell)
 Invoking modified methods,
 133–136
 IRB (Interactive Ruby Shell)
 advanced features in, 166–169
 ancestors in, 142
 AuditDecorator in, 120
 comparison operators in, 51
 default hash values in, 75, 78
 equality of objects and, 43–48
 eval variants in, 123
 garbage collection in, 182
 HashProxy in, 117–118
 inheritance in, 19–21
 initialize method in, 29
 IRB::ExtendCommandBundle for,
 167–169
 libraries in, 169
 log_method in, 135
 overview of, 163–166
 Proc objects in, 136–140
 sessions in, 168–169
 is_a? method, 48
 “is-a” relationships, 79–80

J

JSON format, 171
 Jumping out of scope, 104–106

K

Kernel methods
 Kernel#gem, 170
 Kernel#print, 6
 Kernel#readline, 8
 Kernel module, 64, 122

Keys

- has_key? 78
- KEY constant, 41
- key values, 199
- KeyError, 82–84
- overview of, 74–79

klass, 109

L

lambda method, 137–138, 188

Lazy sweeping phases, 181

Lexical scopes, 40–41, 131–133

Libraries

- in Bundler, 171–175
- introduction to, 163
- in IRB, 169
- MiniTest and, 146
- profile, 189–195
- Ruby documentation in, 163–166

location object, 157

Lock::acquire, 96

log_method, 134–136

LogMethod module, 134–135

lookup method, 199

loop method, 104–106

Loops, 195–197

M

Macintosh System 7, 133

Major marking phases, 180–183

major_gc_count, 182

malloc_increase/malloc_limit, 182

Mark and sweep process, 180–181

Markdown format, 165

Marshal method, 62–63

Marshal::dump method, 190

MatchData object, 7

mean method, 36–38

Memoizing computations, 197–200

Memory

- finalizers and, 185–188
- garbage collectors and, 179–185
- introduction to, 179
- memoizing expensive computations in, 197–200
- memory_profiler gem for, 193–195
- object literals in loops and, 195–197
- profiling tools for, 189–195
- resource safety nets and, 185–188

Metaprogramming

- alias chaining in, 133–136
- class hooks in, generally, 107–113
- class hooks, super in, 114–115

- define_method in, 115–122
- eval variants in, 122–127
- hook methods in, 107–113
- introduction to, 107
- method_missing in, 115–122
- modified methods in, 133–136
- module hooks in, 107–113
- monkey patching alternatives in, 127–133

Proc#arity in, 136–141

- method_missing method in metaprogramming, 115–122
- overview of, 20–24
- super and, 27–28

Methods. *See also specific methods*

- arguments in, 59–63
- finding, 17–24
- method_hooks, 111–112
- missing. *See* method_missing method
- modified, 133–136
- naming rules in, 31
- singleton, 17–24

MiniTest

- MiniTest::Mock#verify for, 155–156
- spec testing with, 149–152
- unit testing with, 145–149

Minor marking phases, 180–181

minor_gc_count, 182

Mirroring namespace/directory structures, 40–42

Mocha, 156

Mock objects, 152–156

mod, 109

Modified methods, 133–136

Modules. *See also specific modules*

- case equality operators and, 48
- class instance variables vs. class variables in, 55–58
- classes vs., 18–19
- Comparable, 49–53
- definition of, 17–19
- equality in, 43–48
- freezing, 9–11
- hooks in, 107–113
- including silently, 17–24
- in inheritance hierarchies, 17–24
- module_eval for, 124–127
- module_exec for, 125–127
- namespace creation with, 38–42
- nesting code in, 38–42
- parsing in, 31–34
- prepend, 141–143
- protected methods and, 53–55

Modules (*continued*)

- setter methods in, 31–34
- struct vs. hash in, 35–38
- super and, generally, 24–28
- super when initializing subclasses in, 28–31

Monitor#alive? 153–155

Monitor#get method, 153–154

Monkey patching, 127–133, 134

MP3 files, 171–174

MrProper, 160

Mutation

- of collections, 59–63
- of constants, 9–11

N

NameError exceptions, 40–41

Namespaces, 38–42

National Oceanic and Atmospheric Administration (NOAA), 35

Nested arrays, 65–66

Nesting definitions in modules, 38–42

NETWORKS array, 10–11

next, 100

nil

- <=> operator returning, 49–53
- caching variables set to, 199–200
- in collections, 63–66
- disabling run-time warnings and, 12–15
- false vs., 1–3
- invalid keys returning, 75–79
- treating objects as if they are, 3–5

nil? method, 2–5

NOAA (National Oceanic and Atmospheric Administration), 35

Numbers, truth of, 1

O

Object literals, 195–197

Object-oriented programming (OOP), 17

Objects. *See also specific objects*

- class instance variables vs. class variables in, 55–58
- comparisons in, 49–53
- defined, 18
- definition of, 17–18
- equality in, 43–48
- equality of, 43–48
- in inheritance hierarchies, 17–24
- namespaces, creating, 38–42

old, 180–183

ordering, 49–53

parsing in, 31–34

protected methods and, 53–55

setter methods in, 31–34

struct vs. hash in, 35–38

super and, generally, 24–28

super when initializing subclasses and, 28–31

young, 180–183

ObjectSpace API, 193–194

ObjectSpace::define_finalizer method, 187

Old objects, 180–183

oldmalloc_increase/oldmalloc_limit, 182

only_space? method, 129–133

OnlySpace module, 129–130

OOP (object-oriented programming), 17

Operators

- %w, 52, 195
- "&," 72
- "||=", 74–75, 197–198
- "~," 140
- "<<," 76–77
- "<=>," 49–53
- "=" (equal sign), 31–34
- "=~," 6–7
- "==," 3, 43–48, 53
- "===," 46–48
- case equality, 46–47
- equal sign ("="), 31–34
- ordering, 52–53
- pessimistic version, 177
- Regex#~, 8–9
- spaceship, 50

Ordering operators, 52–53

Overriding methods, 24–28

P

Pages in heaps, 181

Parent#initialize, 29

Parentheses (), 25–28

Parsing, 31–34

Performance problems, 189–192

Perl, 6

Perlisms, 6–9

Person class, 131–132

Pessimistic version operators, 177

prepended hooks, 110–113

Prepending modules, 141–143

Private states, 53–55

Proc objects, 136–141, 186–188

Proc#arity method, 139–140

Profiling tools, 189–195
 property-based testing, 159–160
 Protected methods, 53–55
 Proxies, 116
 Pry, 169
 purge_unreachable method, 9–10

R

-r, Ruby command-line option, 166
 Raising exceptions. *See also* Exceptions
 custom exceptions in, 85–90
 delegation and, 84
 raising strings vs., 85–90
 RaisingHash for, 82–84, 108–110, 130
 throw vs. raise for, 104–106
 Rake, 14, 149
 Range class, 59
 RDoc (Ruby Documentation), 163–166
 read-eval-print loop (REPL) utilities,
 166–167, 169
 Reading class, 69–70
 Receivers
 definition of, 19, 23
 setter methods and, 31–34
 reduce method, 59, 70–74
 References, 60–63
 Refinements, 131–133
 Refutations, 149
 Regexp class, 47–48
 Regexp#~ operator, 8–9
 reject, 61
 replace! method, 83
 require, 151
 Rescue
 in custom exceptions, 86
 ensure vs. rescue for, 97–100
 in most specific exceptions, 90–93
 reset_var, 127
 Resource safety nets, 185–188
 Resource#close method, 188
 respond_to_missing? 117–122
 response.verify, 154–155
 retry, 86, 100–103
 return, 97–100, 104–106
 reverse, 79–80
 RI (Ruby Information), 163–166
 Role class, 67–69
 RSpec, 152
 Ruby
 1.9.3 version of, 155
 2.0 version of, 131–132, 142
 2.1 version of, 131–132, 182–184,
 195–197

collections in. *See* Collection classes
 comparisons in, 49–53
 conclusions about, 201
 Documentation in, 163–166
 equality in, 43–48
 exceptions in. *See* Exceptions
 false vs. nil in, 1–3
 gems in. *See* RubyGems
 Information in, 163–166
 inheritance hierarchies in, 17–24
 introduction to, 1
 libraries in. *See* Libraries
 memory management in. *See* Memory
 metaprogramming in. *See*
 Metaprogramming
 mutable constants in, 9–11
 namespaces in, 38–42
 nil objects in, 3–5
 parsing in, 31–34
 perlisms in, 6–9
 on Rails. *See* Ruby on Rails
 ruby -rprofile script.rb, 189
 RUBY_GC_HEAP_ settings for, 184
 RUBY_GC_MALLOC_ settings for, 184
 RUBY_GC_OLDMALLOC_ settings for,
 184–185
 RUBYOPT environment for, 14–15
 run-time warnings in, 12–15
 setter methods in, 31–34
 sharing private state via protected
 methods in, 53–55
 struct vs. hash in, 35–38
 super in, generally, 24–28
 super when initializing subclasses in,
 28–31
 testing in. *See* Testing code
 tools in. *See* Tools
 true values in, 1–3
 Ruby Documentation (RDoc), 163–166
 Ruby Information (RI), 163–166
 Ruby on Rails
 class hooks in, 114–115
 monkey patching alternatives in,
 127–128
 sessions in, 168–169
 testing and, 149
 RubyGems
 Bundler, 170–175
 Cucumber, 152
 FuzzBert, 158–159
 memory_profiler, 193–195
 MiniTest. *See* MiniTest
 monkey patching alternatives in, 128

RubyGems (*continued*)

- MrProper, 160
- number of, 170
- Pry, 169
- ruby-prof gem, 190–195
- SimpleCov, 160–161
- stackprof, 190–191

Run-time warnings, 12–15, 86–90

S

- Scalar objects, 63–66
- Scope, 104–106
- Seed message, 156–157
- select method, 73
- self, 33–34, 107–113
- Sessions, 168
- Set class
 - in collections, 68–70
 - inheritance and, 80
 - “set” files and, 70
- Setter methods, 31–34
- setup method, 148, 151
- Shallow copies, 62–63
- shipped? method, 198–200
- SimpleCov, 160–161
- Simulating determinism, 152
- Singleton classes, 19–24
- Singleton methods
 - hook methods as, 107–113
 - inheritance in, 19, 22–24
 - instance_eval/instance_exec as, 124–127
 - singleton_method_hooks in, 112–113
- Singleton patterns, 56–58
- Slots, 181–183
- Smalltalk, 17
- Spaceship operators, 50
- Spec tests, 149–152
- Special global variables, 6
- split method, 12–15
- stackprof gem, 190–191
- StandardError, 87–90, 93
- Statistics, 189–195
- StopIteration, 104–106
- Stream, 138–140
- Strings
 - String class for, 128–133
 - string literals, 43–44
 - StringExtra class for, 130–133
 - String#match vs. String#=~, 6, 8–9
- Strong Proc objects, 137–138
- Structured data
 - introduction to, 35

- Struct class for, 36–38

- Struct::new for, 37–38

sum method, 72–74

super

- behaviors of, 24–28
- delegation and, 84
- prepending and, 143
- when initializing subclasses, 28–31

Superclasses

- class variables in, 57–58
- introduction to, 18–24
- prepending and, 141–143

SuperForwardable module, 108–110

Sweeping phases, 180–181

T

taint method, 79–84, 108–113

TDD (test-driven development), 157

teardown method, 148, 151

“test_” prefix, 147–148

test-driven development (TDD), 157

Testing code

- effectiveness of, 156–161
- introduction to, 145
- MiniTest spec testing in, 149–152
- MiniTest unit testing in, 145–149
- mock objects in, 152–156
- simulating determinism in, 152

throw statements, 100, 104–106

to_i method, 4–5

to_s method, 4–5, 159–160

Tools

- Bundler, 170–175

- for gem dependencies, managing, 170–175

- for gem dependencies, setting upper bounds, 175–178

- introduction to, 163

- in IRB, 166–169

- for Ruby documentation, 163–166

total_allocated_object/total_freed_object, 182

true, 1–3, 43–48

Tuner objects, 60–62

U

Unary complement operator (“~”), 140

undefine_finalizer, 188

Underscore (“_”) variable feature, 168

Unit testing, 145–149

unlog_method, 135–136

untaint method, 79–84, 108

update method, 139

Upper bounds on version requirements,
175–178

URLs, 114

User::`find` method, 197

V

Values

 false, 1–3

 nil. *See* nil

 true, 1–3, 43–48

Variable shadowing, 13

Version class

`eq?` method and, 49–53

 MiniTest for, generally, 146–149

 MiniTest spec testing for, 150

 property-based testing of, 159–160

Version number specifications, 175–178

Versions of Ruby. *See* Ruby

W

-w, Ruby command-line option,
14–15

Weak Proc objects, 137–139

White-box testing, 153

Whitelisting, 91

`who_am_i?` method, 141–142

Widget class, 156–157

`Widget#overlapping?` method,
54–55

Y

yield statements, 95–96

Young objects, 180–183

Z

ZenTest, 161

Zero, 1