Addison-Wesley Professional Ruby Series

"Sticking to its tried and tested formula of cutting right to the techniques the modern day Rubyist needs to know, the latest edition of The Ruby Way *keeps its strong reputation going for the latest generation of the Ruby language."*

—**PETER COOPER,** Editor *of Ruby Weekly*

THE RUBY WAY

THIRD EDITION

HAL FULTON

with **ANDRÉ ARKO**

Praise for *The Ruby Way,* Third Edition

"Sticking to its tried and tested formula of cutting right to the techniques the modern day Rubyist needs to know, the latest edition of *The Ruby Way* keeps its strong reputation going for the latest generation of the Ruby language."

Peter Cooper
Editor of *Ruby Weekly*

"The authors' excellent work and meticulous attention to detail continues in this latest update; this book remains an outstanding reference for the beginning Ruby programmer—as well as the seasoned developer who needs a quick refresh on Ruby. Highly recommended for anyone interested in Ruby programming."

Kelvin Meeks
Enterprise Architect

Praise for Previous Editions of *The Ruby Way*

"Among other things, this book excels at explaining metaprogramming, one of the most interesting aspects of Ruby. Many of the early ideas for Rails were inspired by the first edition, especially what is now Chapter 11. It puts you on a rollercoaster ride between 'How could I use this?' and 'This is so cool!' Once you get on that rollercoaster, there's no turning back."

David Heinemeier Hansson
Creator of Ruby on Rails,
Founder at Basecamp

"The appearance of the second edition of this classic book is an exciting event for Rubyists—and for lovers of superb technical writing in general. Hal Fulton brings a lively erudition and an engaging, lucid style to bear on a thorough and meticulously exact exposition of Ruby. You palpably feel the presence of a teacher who knows a tremendous amount and really wants to help you know it too."

David Alan Black
Author of *The Well-Grounded Rubyist*

"This is an excellent resource for gaining insight into how and why Ruby works. As someone who has worked with Ruby for several years, I still found it full of new tricks and techniques. It's accessible both as a straight read and as a reference that one can dip into and learn something new."

Chet Hendrickson
Agile software pioneer

"Ruby's a wonderful language—but sometimes you just want to get something done. Hal's book gives you the solution and teaches a good bit about why that solution is good Ruby."

Martin Fowler
Chief Scientist, ThoughtWorks
Author of *Patterns of Enterprise Application Architecture*

The Ruby Way

Third Edition

Hal Fulton

with André Arko

Addison-Wesley

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

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2014945504

ISBN-13: 978-0-321-71463-3
ISBN-10: 0-321-71463-6

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana
First printing: March 2015

Editor-in-Chief
Mark Taub

Executive Editor
Debra Williams-Cauley

Development Editor
Songlin Qiu

Managing Editor
Kristy Hart

Project Editor
Andy Beaster

Copy Editor
Bart Reed

Indexer
Ken Johnson

Proofreader
Sarah Kearns

Cover Designer
Chuti Prasertsith

Senior Compositor
Gloria Schurick

To my parents, without whom I would not be possible
—Hal

This page intentionally left blank

Contents

Foreword

Foreword to the Third Edition

Yesterday I was reading an article about geek fashion in `Wired.com`. According to it, wearing a Rubyconf 2012 t-shirt these days signals to people: "I work for Oracle."

Wow. How far we've come in the last 10 years!

For quite some time, using Ruby set you apart from the mainstream. Now it seems we are the mainstream. And what a long, strange journey it has been to get there.

Ruby adoption took a long time by today's standards. I read this book in 2005, and at that point, the first edition was over four years old. Ruby had just begun its second wave of adoption thanks to DHH and the start of Rails mania. It seemed like there might be a couple hundred people in the entire (English-speaking) world that used Ruby. Amazingly, at that point, the first edition of this book was already four years old. That's how ahead of its time it was.

This new edition keeps the writing style that has made the book such a hit with experienced programmers over the years. The long first chapter covers fundamental basics of object-orientation and the Ruby language. It's a must read for anyone new to the language. But it does so in concise, fast-moving narrative that assumes you already know how to create software.

From there, the chapters follow a distinctive pattern. A bit of backstory narrative, followed by rapid-fire bits of knowledge about the Ruby language. Snippets of example code are abundant and help to illuminate the concept under discussion. You can lift code samples verbatim into your programs. Especially once you get into the more practical applications chapters later in the book.

A brief bit of personal backstory seems appropriate. I owe a huge debt of gratitude to Hal for this book and the way that he wrote it. In 2005, I started work on a manuscript for Addison Wesley about the use of Ruby on Rails in the enterprise. It was my first attempt at authoring a book, and after penning about two chapters, I got stuck. Few people were using Ruby or Rails in the enterprise at that time and I had to remind myself that I was attempting to write non-fiction.

After discussing options with my editor, we determined that the best course of action might be to ditch the idea and start on a new one. *The Rails Way* was to cover the nascent Ruby on Rails framework in the style of this book. I employed terse narrative accompanying plentiful code examples. Instead of long listings, I interspersed commentary between sprinkles of code that provided just enough samples of the framework to make sense.

Like *The Ruby Way*, I aimed for breadth of coverage rather than depth. I wanted *The Rails Way* to claim permanent real estate on the desk of the serious Rails programmer. Like *The Ruby Way*, I wanted my book to be a default go-to reference. In contrast to other Rails books, I skipped tutorial material and ignored complete beginners.

And it was a huge success! Safe to say that without Hal's book, my own book would not exist and my career would have taken a less successful trajectory.

But enough congratulatory retrospective! Let's get back to the present day and the newest edition of *The Ruby Way* that you're currently reading. The immensely talented André Arko joins Hal this time around. What a great team! They deliver a painstaking revision that brings the book up to date with the latest edition of our beloved Ruby language.

My personal highlights of this edition include the following:

- A whole chapter of in-depth coverage of the new Onigmo regular expression engine. I love its beautiful and concise explanations of concepts such as positive and negative lookahead and lookbehind.
- The Internationalization chapter tackles thorny issues around String encoding and Unicode normalization. Bloggers have covered the subject in spotty fashion over the years, but having it all presented in one place is invaluable.
- The Ruby and Web Applications chapter manages to squeeze a crash-course in Rack, Sinatra, and Rails into less than 30 pages.

---

* Want proof of André's ingenuity? See how he cuts the load time for a real Rails app down to 500ms or less at http://andre.arko.net/2014/06/27/rails-in-05-seconds/.

I predict that this edition of *The Ruby Way* will be as successful as its predecessors. It gives me great joy to make it the latest addition to our Professional Ruby Series.

Obie Fernandez
September 15, 2014

Foreword to the Second Edition

In ancient China, people, especially philosophers, thought that something was hidden behind the world and every existence. It can never be told, nor explained, nor described in concrete words. They called it *Tao* in Chinese and *Do* in Japanese. If you translate it into English, it is the word for *Way.* It is the *Do* in Judo, Kendo, Karatedo, and Aikido. They are not only martial arts, but they also include a philosophy and a way of life.

Likewise, Ruby the programming language has its philosophy and way of thinking. It enlightens people to think differently. It helps programmers have more fun in their work. It is not because Ruby is from Japan but because programming is an important part of the human being (well, at least *some* human beings), and Ruby is designed to help people have a better life.

As always, "Tao" is difficult to describe. I feel it but have never tried to explain it in words. It's just too difficult for me, even in Japanese, my native tongue. But a guy named Hal Fulton tried, and his first try (the first edition of this book) was pretty good. This second version of his trial to describe the Tao of Ruby becomes even better with help from many people in the Ruby community. As Ruby becomes more popular (partly due to Ruby on Rails), it becomes more important to understand the secret of programmers' productivity. I hope this book helps you to become an efficient programmer.

Happy Hacking.

Yukihiro "Matz" Matsumoto
August 2006, Japan
まつもと ゆきひろ

Foreword to the First Edition

Shortly after I first met with computers in the early 80s, I became interested in programming languages. Since then I have been a "language geek." I think the reason for this interest is that programming languages are ways to express human thought. They are fundamentally human-oriented.

Despite this fact, programming languages have tended to be machine-oriented. Many languages were designed for the convenience of the computer.

But as computers became more powerful and less expensive, this situation gradually changed. For example, look at structured programming. Machines do not care whether programs are structured well; they just execute them bit by bit. Structured programming is not for machines, but for humans. This is true of object-oriented programming as well.

The time for language design focusing on humans has been coming.

In 1993, I was talking with a colleague about scripting languages, about their power and future. I felt scripting to be the way future programming should be—human-oriented.

But I was not satisfied with existing languages such as Perl and Python. I wanted a language that was more powerful than Perl and more object-oriented than Python. I couldn't find the ideal language, so I decided to make my own.

Ruby is not the simplest language, but the human soul is not simple in its natural state. It loves simplicity and complexity at the same time. It can't handle too many complex things, nor too many simple things. It's a matter of balance.

So to design a human-oriented language, Ruby, I followed the Principle of Least Surprise. I consider that everything that surprises me less is good. As a result, I feel a natural feeling, even a kind of joy, when programming in Ruby. And since the first release of Ruby in 1995, many programmers worldwide have agreed with me about the joy of Ruby programming.

As always I'd like to express my greatest appreciation to the people in the Ruby community. They are the heart of Ruby's success.

I am also thankful to the author of this book, Hal E. Fulton, for declaring the Ruby Way to help people.

This book explains the philosophy behind Ruby, distilled from my brain and the Ruby community. I wonder how it can be possible for Hal to read my mind to know and reveal this secret of the Ruby Way. I have never met him face to face; I hope to meet him soon.

I hope this book and Ruby both serve to make your programming fun and happy.

Yukihiro "Matz" Matsumoto
September 2001, Japan
まつもと ゆきひろ

Acknowledgments

Acknowledgments for the Third Edition

As can be expected by now, the process of updating this book for the third edition turned out to be somewhat monumental. Ruby has changed dramatically since the days of 1.8, and being a Ruby programmer is far more popular now than it has ever been before.

Verifying, updating, and rewriting this book took quite some time longer than expected. Ruby has progressed from 1.9 through 2.0 and 2.1, and this book has progressed through at least as many edits and rewrites along the way.

Many people contributed to making this book possible. At Addison-Wesley, Debra Williams Cauley, Songlin Qiu, Andy Beaster, and Bart Reed provided the encouragement, coordination, and editing needed to complete this edition. The contributions of Russ Olsen and André Arko were *absolutely invaluable*.

This edition was technically edited by Russ Olsen and Steve Klabnik, providing feedback and suggestions that made the book more accurate and understandable. Russ also provided the Ruby libraries and scripts that compiled the latest version of the book itself. As always, any errors are mine, not theirs.

Suggestions, code samples, or simply helpful explanations were provided by Dave Thomas, David Alan Black, Eric Hodel, Chad Fowler, Brad Ediger, Sven Fuchs, Jesse Storimer, Luke Francl, and others over the years.

Special thanks go to Paul Harrison and the rest of my colleagues at Simpli.fi for their encouragement and support.

I also wish to honor the memory of Guy Decoux and more recently Jim Weirich. Jim in particular made significant contributions to this book and to our community.

Final thanks are owed, as always, to Matz himself for creating Ruby, and to you, the reader of this book. I hope it is able to teach, inform, and maybe even amuse you.

Acknowledgments for the Second Edition

Common sense says that a second edition will only require half as much work as the first edition required. Common sense is wrong.

Even though a large part of this book came directly from the first edition, even that part had to be tweaked and tuned. Every single sentence in this book had to pass through (at the very least) a filter that asked: Is what was true in 2001 still true in 2006? And that, of course, was only the beginning.

In short, I put in many hundreds of hours of work on this second edition—nearly as much time as on the first. And yet I am "only the author."

A book is possible only through the teamwork of many people. On the publisher's side, I owe thanks to Debra Williams-Cauley, Songlin Qiu, and Mandie Frank for their hard work and infinite patience. Thanks go to Geneil Breeze for her tireless copy editing and picking bits of lint from my English. There are also others I can't name because their work was completely behind the scenes, and I never talked with them.

Technical editing was done primarily by Shashank Date and Francis Hwang. They did a great job, and I appreciate it. Errors that slipped through are my responsibility, of course.

Thanks go to the people who supplied explanations, wrote sample code, and answered numerous questions for me. These include Matz himself (Yukihiro Matsumoto), Dave Thomas, Christian Neukirchen, Chad Fowler, Curt Hibbs, Daniel Berger, Armin Roehrl, Stefan Schmiedl, Jim Weirich, Ryan Davis, Jenny W., Jim Freeze, Lyle Johnson, Martin DeMello, Matt Lawrence, the infamous *why the lucky stiff*, Ron Jeffries, Tim Hunter, Chet Hendrickson, Nathaniel Talbott, and Bil Kleb.

Special thanks goes to the heavier contributors. Andrew Johnson greatly enhanced my regular expression knowledge. Paul Battley made great contributions to the internationalization chapter. Masao Mutoh added to that same chapter and also contributed material on GTK. Austin Ziegler taught me the secrets of writing PDF files. Caleb Tennis added to the Qt material. Eric Hodel added to the Rinda and Ring material, and James Britt contributed heavily to the web development chapter.

Thanks and appreciation again must go to Matz, not only for his assistance but for creating Ruby in the first place. *Domo arigato gozaimasu*!

Again I have to thank my parents. They have encouraged me without ceasing and are looking forward to seeing this book. I will make programmers of them both yet.

And once again, I have to thank all of the Ruby community for their tireless energy, productivity, and community spirit. I particularly thank the readers of this book (in both editions). I hope you find it informative, useful, and perhaps even entertaining.

Acknowledgments for the First Edition

Writing a book is truly a team effort; this is a fact I could not fully appreciate until I wrote one myself. I recommend the experience, although it is a humbling one. It is a simple truth that without the assistance of many other people, this book would not have existed.

Thanks and appreciation must first go to Matz (Yukihiro Matsumoto), who created the Ruby language in the first place. Domo arigato gozaimasu!

Thanks goes to Conrad Schneiker for conceiving the overall idea for the book and helping to create its overall structure. He also did me the service of introducing me to the Ruby language in 1999.

Several individuals have contributed material to the body of the book. The foremost of these was Guy Hurst, who wrote substantial parts of the earlier chapters as well as two of the appendices. His assistance was absolutely invaluable.

Thanks also goes to the other contributors, whom I'll name in no particular order. Kevin Smith did a great job on the GTK section of Chapter 6, saving me from a potentially steep learning curve on a tight schedule. Patrick Logan, in the same chapter, shed light on the mysteries of the FOX GUI. Chad Fowler, in Chapter 9, plumbed the depths of XML and also contributed to the CGI section.

Thanks to those who assisted in proofreading or reviewing or in other miscellaneous ways: Don Muchow, Mike Stok, Miho Ogishima, and others already mentioned. Thanks to David Eppstein, the mathematics professor, for answering questions about graph theory.

One of the great things about Ruby is the support of the community. There were many on the mailing list and the newsgroup who answered questions and gave me ideas and assistance. Again in no particular order, these are Dave Thomas, Andy Hunt, Hee-Sob Park, Mike Wilson, Avi Bryant, Yasushi Shoji ("Yashi"), Shugo Maeda, Jim Weirich, "arton," and Masaki Suketa. I'm sorry to say I have probably overlooked someone.

To state the obvious, a book would never be published without a publisher. Many people behind the scenes worked hard to produce this book; primarily I have to thank William Brown, who worked closely with me and was a constant source of encouragement; and Scott Meyer, who delved deeply into the details of putting the material together. Others I cannot even name because I have never heard of them. You know who you are.

I have to thank my parents, who watched this project from a distance, encouraged me along the way, and even bothered to learn a little bit of computer science for my sake.

A writer friend of mine once told me, "If you write a book and nobody reads it, you haven't really written a book." So, finally, I want to thank the reader. This book is for you. I hope it is of some value.

About the Authors

Hal Fulton first began using Ruby in 1999. In 2001, he started work on *The Ruby Way*, which was the second Ruby book published in English. Fulton was an attendee at the very first Ruby conference in 2001 and has presented at numerous other Ruby conferences on three continents, including the first European Ruby Conference in 2003. He holds two degrees in computer science from the University of Mississippi and taught computer science for four years. He has worked for more than 25 years with various forms of UNIX and Linux. He is now at Simpli.fi in Fort Worth, Texas, where he works primarily in Ruby.

André Arko first encountered Ruby as a student in 2004, and reading the first edition of this book helped him decide to pursue a career as a Ruby programmer. He is team lead of Bundler, the Ruby dependency manager, and has created or contributes to dozens of other open source projects. He works at Cloud City Development as a consultant providing team training and expertise on Ruby and Rails as well as developing web applications.

André enjoys sharing hard-won knowledge and experience with other developers, and has spoken at over a dozen Ruby conferences on four continents. He is a regular volunteer at RailsBridge and RailsGirls programming outreach events, and works to increase diversity and inclusiveness in both the Ruby community and technology as a field. He lives in San Francisco, California.

Introduction

The way that can be named is not the true Way.

—*Lao Tse,* Tao Te Ching

The title of this book is *The Ruby Way.* This is a title that begs for a disclaimer.

It has been my aim to align this book with the philosophy of Ruby as well as I could. That has also been the aim of the other contributors. Credit for success must be shared with these others, but the blame for any mistakes must rest solely with me.

Of course, I can't presume to tell you with exactness what the spirit of Ruby is all about. That is primarily for Matz to say, and I think even he would have difficulty communicating all of it in words.

In short, *The Ruby Way* is only a book, but the Ruby Way is the province of the language creator and the community as a whole. This is something difficult to capture in a book.

Still, I have tried in this introduction to pin down a little of the ineffable spirit of Ruby. The wise student of Ruby will not take it as totally authoritative.

About the Third Edition

Everything changes, and Ruby is no exception. There are many changes and much new material in this edition. In a larger sense, every single chapter in this book is "new." I have revised and updated every one of them, making hundreds of minor changes and dozens of major changes. I deleted items that were obsolete or of lesser importance; I changed material to fit changes in Ruby itself; I added examples and commentary to every chapter.

As the second Ruby book in the English language (after *Programming Ruby*, by Dave Thomas and Andy Hunt), *The Ruby Way was* designed to be complementary to that book rather than overlap with it; that is still true today.

There have been numerous changes between Ruby 1.8, covered in the second edition, and Ruby 2.1, covered here. It's important to realize, however, that these were made with great care, over several years. Ruby is still Ruby. Much of the beauty of Ruby is derived from the fact that it changes slowly and deliberately, crafted by the wisdom of Matz and the other developers.

Today we have a proliferation of books on Ruby and more articles published than we can bother to notice. Web-based tutorials and documentation resources abound.

New tools and libraries have appeared. The most common of these seem to be tools by developers for other developers: web frameworks, blogging tools, markup tools, and interfaces to exotic data stores. But there are many others, of course—GUIs, number-crunching, web services, image manipulation, source control, and more.

Ruby editor support is widespread and sophisticated. IDEs are available that are useful and mature (and which share some overlap with the GUI builders).

It's also undeniable that the community has grown and changed. Ruby is by no means a niche language today; it is used in government departments such as NASA and NOAA, enterprise companies such as IBM and Motorola, and well-known websites such as Wikipedia, GitHub, and Twitter. It is used for graphics work, database work, numerical analysis, web development, and more. In short—and I mean this in the positive sense—Ruby has gone mainstream.

Updating this book has been a labor of love. I hope it is useful to you.

How This Book Works

You probably won't learn Ruby from this book. There is relatively little in the way of introductory or tutorial information. If you are totally new to Ruby, you might want start with another book.

Having said that, programmers are a tenacious bunch, and I grant that it might be possible to learn Ruby from this book. Chapter 1, "Ruby in Review," does contain a brief introduction and some tutorial information.

Chapter 1 also contains a comprehensive "gotcha" list (which has been difficult to keep up to date). The usefulness of this list in Chapter 1 will vary widely from one reader to another because we cannot all agree on what is intuitive.

This book is largely intended to answer questions of the form "How do I…?." As such, you can expect to do a lot of skipping around. I'd be honored if everyone read every page from front to back, but I don't expect that. It's more my expectation that

you will browse the table of contents in search of techniques you need or things you find interesting.

As it turns out, I have talked to many people since the first edition, and they *did* in fact read it cover to cover. What's more, I have had more than one person report to me that they did learn Ruby here. So anything is possible.

Some things this book covers may seem elementary. That is because people vary in background and experience; what is obvious to one person may not be to another. I have tried to err on the side of completeness. On the other hand, I have tried to keep the book at a reasonable size (obviously a competing goal).

This book can be viewed as a sort of "inverted reference." Rather than looking up the name of a method or a class, you will look things up by function or purpose. For example, the `String` class has several methods for manipulating case: `capitalize`, `upcase`, `casecmp`, `downcase`, and `swapcase`. In a reference work, these would quite properly be listed alphabetically, but in this book they are all listed together.

Of course, in striving for completeness, I have sometimes wandered onto the turf of the reference books. In many cases, I have tried to compensate for this by offering more unusual or diverse examples than you might find in a reference.

I have tried for a high code-to-commentary ratio. Overlooking the initial chapter, I think I've achieved this. Writers may grow chatty, but programmers always want to see the code. (If not, they *should* want to.)

The examples here are sometimes contrived, for which I must apologize. To illustrate a technique or principle *in isolation from a real-world problem* can be difficult. However, the more complex or high level the task was, the more I attempted a real-world solution. Thus, if the topic is concatenating strings, you may find an unimaginative code fragment involving `"foo"` and `"bar"`, but when the topic is something like parsing XML, you will usually find a much more meaningful and realistic piece of code.

This book has two or three small quirks to which I'll confess up front. One is the avoidance of the "ugly" Perl-like global variables such as `$_` and the others. These are present in Ruby, and they work fine; they are used daily by most or all Ruby programmers. But in nearly all cases, their use can be avoided, and I have taken the liberty of omitting them in most of the examples.

Another quirk is that I avoid using standalone expressions when they don't have side effects. Ruby is expression oriented, and that is a good thing; I have tried to take advantage of that feature. But in a code fragment, I prefer to not write expressions that merely return a value that is not usable. For example, the expression `"abc" + "def"` can illustrate string concatenation, but I would write something like `str = "abc" + "def"` instead. This may seem wordy to some, but it may seem more natural to you

if you are a C programmer who really notices when functions are void or nonvoid (or an old-time Pascal programmer who thinks in procedures and functions).

My third quirk is that I don't like the "pound" notation to denote instance methods. Many Rubyists will think I am being verbose in saying "instance method `crypt` of class `String`" rather than saying `String#crypt`, but I think no one will be confused. (Actually, I am slowly being converted to this usage, as it is obvious the pound notation is not going away.)

I have tried to include "pointers" to outside resources whenever appropriate. Time and space did not allow putting everything into this book that I wanted, but I hope I have partially made up for that by telling you where to find related materials. The ruby-doc.org and rdoc.info websites are probably the foremost of these sources; you will see them referenced many times in this book.

Here, at the front of the book, there is usually a gratuitous reference to the typefaces used for code, and how to tell code fragments from ordinary text. But I won't insult your intelligence; you've read computer books before.

I want to point out that roughly 10 percent of this book was written by other people. That does not even include tech editing and copy editing. You should read the acknowledgments in this (and every) book. Most readers skip them. Go read them now. They're good for you, like vegetables.

About the Book's Source Code

Every significant code fragment has been collected into an archive for the reader to download. Look for this archive on the informit.com site or at the book's own site, therubyway.io.

It is offered both as a `.tgz` file and as a `.zip` file. Code fragments that are very short or can't be run "out of context" will usually not appear in the archive.

What Is the "Ruby Way"?

Let us prepare to grapple with the ineffable itself, and see if we may not eff it after all.

—*Douglas Adams,* Dirk Gently's Holistic Detective Agency

What do we mean by the Ruby Way? My belief is that there are two related aspects: One is the philosophy of the design of Ruby; the other is the philosophy of its usage. It is natural that design and use should be interrelated, whether in software or

hardware; why else should there be such a field as ergonomics? If I build a device and put a handle on it, it is because I expect someone to grab that handle.

Ruby has a nameless quality that makes it what it is. We see that quality present in the design of the syntax and semantics of the language, but it is also present in the programs written for that interpreter. Yet as soon as we make this distinction, we blur it.

Clearly Ruby is not just a tool for creating software, but it is a piece of software in its own right. Why should the workings of Ruby *programs* follow laws different from those that guide the workings of the *interpreter*? After all, Ruby is highly dynamic and extensible. There might be reasons that the two levels should differ here and there, probably for accommodating to the inconvenience of the real world. But in general, the thought processes can and should be the same. Ruby could be implemented in Ruby, in true Hofstadter-like fashion, though it is not at the time of this writing.

We don't often think of the etymology of the word *way*, but there are two important senses in which it is used. On the one hand, it means *a method or technique*, but it can also mean *a road or path*. Obviously these two meanings are interrelated, and I think when I say "the Ruby Way," I mean both of them.

So what we are talking about is a thought process, but it is also a path that we follow. Even the greatest software guru cannot claim to have reached perfection but only to follow the path. And there may be more than one path, but here I can only talk about one.

The conventional wisdom says that *form follows function*. And the conventional wisdom is, of course, conventionally correct. But Frank Lloyd Wright (speaking in his own field) once said, "Form follows function—that has been misunderstood. Form and function should be one, joined in a spiritual union."

What did Wright mean? I would say that this truth is not something you learn from a book, but from experience.

However, I would argue that Wright expressed this truth elsewhere in pieces easier to digest. He was a great proponent of simplicity, saying once, "An architect's most useful tools are an eraser at the drafting board and a wrecking bar at the site."

So, one of Ruby's virtues is simplicity. Shall I quote other thinkers on the subject? According to Antoine de St. Exupéry, "Perfection is achieved, not when there is nothing left to add, but when there is nothing left to take away."

But Ruby is a complex language. How can I say that it is simple?

If we understood the universe better, we might find a "law of conservation of complexity"—a fact of reality that disturbs our lives like entropy so that we cannot avoid it but can only redistribute it.

And that is the key. We can't avoid complexity, but we can push it around. We can bury it out of sight. This is the old "black box" principle at work; a black box performs a complex task, but it possesses simplicity *on the outside*.

If you haven't already lost patience with my quotations, a word from Albert Einstein is appropriate here: "Everything should be as simple as possible, but no simpler."

So in Ruby, we see simplicity embodied from the programmer's view (if not from the view of those maintaining the interpreter). Yet we also see the capacity for compromise. In the real world, we must bend a little. For example, every entity in a Ruby program should be a true object, but certain values such as integers are stored as immediate values. In a trade-off familiar to computer science students for decades, we have traded elegance of design for practicality of implementation. In effect, we have traded one kind of simplicity for another.

What Larry Wall said about Perl holds true: "When you say something in a small language, it comes out big. When you say something in a big language, it comes out small." The same is true for English. The reason that biologist Ernst Haeckel could say "Ontogeny recapitulates phylogeny" in only three words was that he had these powerful words with highly specific meanings at his disposal. We allow inner complexity of the language because it enables us to shift the complexity away from the individual utterance.

I would state this guideline this way: Don't write 200 lines of code when ten will do.

I'm taking it for granted that brevity is generally a good thing. A short program fragment will take up less space in the programmer's brain; it will be easier to grasp as a single entity. As a happy side effect, fewer bugs will be injected while the code is being written.

Of course, we must remember Einstein's warning about simplicity. If we put brevity too high on our list of priorities, we will end up with code that is hopelessly obfuscated. Information theory teaches us that compressed data is statistically similar to random noise; if you have looked at C or APL or regular expression notation—especially badly written—you have experienced this truth firsthand. "Simple, but not too simple"; that is the key. Embrace brevity, but do not sacrifice readability.

It is a truism that both brevity and readability are good. But there is an underlying reason for this—one so fundamental that we sometimes forget it. The reason is that computers exist for humans, not humans for computers.

In the old days, it was almost the opposite. Computers cost millions of dollars and ate electricity at the rate of many kilowatts. People acted as though the computer was

a minor deity and the programmers were humble supplicants. An hour of the computer's time was more expensive than an hour of a person's time.

When computers became smaller and cheaper, high-level languages also became more popular. These were inefficient from the computer's point of view but efficient from the human perspective. Ruby is simply a later development in this line of thought. Some, in fact, have called it a *VHLL* (Very High-Level Language); though this term is not well-defined, I think its use is justified here.

The computer is supposed to be the servant, not the master, and, as Matz has said, a smart servant should do a complex task with a few short commands. This has been true through all the history of computer science. We started with machine languages and progressed to assembly language and then to high-level languages.

What we are talking about here is a shift from a *machine-centered* paradigm to a *human-centered* one. In my opinion, Ruby is an excellent example of human-centric programming.

I'll shift gears a little. There was a wonderful little book from the 1980s called *The Tao of Programming* (by Geoffrey James). Nearly every line is quotable, but I'll repeat only this: "A program should follow the 'Law of Least Astonishment.' What is this law? It is simply that the program should always respond to the user in the way that astonishes him least." (Of course, in the case of a language interpreter, the *user* is the programmer.)

I don't know whether James coined this term, but his book was my first introduction to the phrase. This is a principle that is well known and often cited in the Ruby community, though it is usually called the *Principle of Least Surprise*, or *POLS*. (I myself stubbornly prefer the acronym *LOLA*.)

Whatever you call it, this rule is a valid one, and it has been a guideline throughout the ongoing development of the Ruby language. It is also a useful guideline for those who develop libraries or user interfaces.

The only problem, of course, is that different people are surprised by different things; there is no universal agreement on how an object or method "ought" to behave. We can strive for consistency and strive to justify our design decisions, and each person can train his own intuition.

For the record, Matz has said that "least surprise" should refer to *him* as the designer. The more you think like him, the less Ruby will surprise you. And I assure you, imitating Matz is not a bad idea for most of us.

No matter how logically constructed a system may be, your intuition needs to be trained. Each programming language is a world unto itself, with its own set of assumptions, and human languages are the same. When I took German, I learned that all

nouns were capitalized, but the word *deutsch* was not. I complained to my professor; after all, this was the *name* of the language, wasn't it? He smiled and said, "Don't fight it."

What he taught me was to *let German be German*. By extension, that is good advice for anyone coming to Ruby from some other language. Let Ruby be Ruby. Don't expect it to be Perl, because it isn't; don't expect it to be LISP or Smalltalk, either. On the other hand, Ruby has common elements with all three of these. Start by following your expectations, but when they are violated, don't fight it (unless Matz agrees it's a needed change).

Every programmer today knows the orthogonality principle (which would better be termed the *orthogonal completeness principle*). Suppose we have an imaginary pair of axes with a set of comparable language entities on one and a set of attributes or capabilities on the other. When we talk of "orthogonality," we usually mean that the space defined by these axes is as "full" as we can logically make it.

Part of the Ruby Way is to strive for this orthogonality. An array is in some ways similar to a hash, so the operations on each of them should be similar. The limit is reached when we enter the areas where they are different.

Matz has said that "naturalness" is to be valued over orthogonality. But to fully understand what is natural and what is not may take some thinking and some coding.

Ruby strives to be friendly to the programmer. For example, there are aliases or synonyms for many method names; `size` and `length` will both return the number of entries in an array. Some consider this sort of thing to be an annoyance or anti-feature, but I consider it a good design.

Ruby strives for consistency and regularity. There is nothing mysterious about this; in every aspect of life, we yearn for things to be regular and parallel. What makes it a little more tricky is learning when to violate this principle.

For instance, Ruby has the habit of appending a question mark (`?`) to the name of a predicate-like method. This is well and good; it clarifies the code and makes the namespace a little more manageable. But what is more controversial is the similar use of the exclamation point in marking methods that are "destructive" or "dangerous" in the sense that they modify their receivers. The controversy arises because *not all* of the destructive methods are marked in this way. Shouldn't we be consistent?

No, in fact we should not. Some of the methods by their very nature change their receiver (such as the `Array` methods `replace` and `concat`). Some of them are "writer" methods allowing assignment to a class attribute; we should *not* append an exclamation point to the attribute name or the equal sign. Some methods arguably change the state of the receiver, such as `read`; this occurs too frequently to be marked

in this way. If every destructive method name ended in a `!`, our programs soon would look like sales brochures for a multilevel marketing firm.

Do you notice a kind of tension between opposing forces, a tendency for all rules to be violated? Let me state this as Fulton's Second Law: *Every rule has an exception, except Fulton's Second Law.* (Yes, there is a joke there, a very small one.)

What we see in Ruby is not a "foolish consistency" nor a rigid adherence to a set of simple rules. In fact, perhaps part of the Ruby Way is that it is *not* a rigid and inflexible approach. In language design, as Matz once said, you should "follow your heart."

Yet another aspect of the Ruby philosophy is, do *not fear change at runtime; do not fear what is dynamic.* The world is dynamic; why should a programming language be static? Ruby is one of the most dynamic languages in existence.

I would also argue that another aspect is, do not be a slave to performance issues. When performance is unacceptable, the issue must be addressed, but it should normally not be the first thing you think about. Prefer elegance over efficiency where efficiency is less than critical. Then again, if you are writing a library that may be used in unforeseen ways, performance may be critical from the start.

When I look at Ruby, I perceive a balance between different design goals, a complex interaction reminiscent of the *n*-body problem in physics. I can imagine it might be modeled as an Alexander Calder mobile. It is perhaps this interaction itself, the harmony, that embodies Ruby's philosophy rather than just the individual parts. Programmers know that their craft is not just science and technology but art. I hesitate to say that there is a spiritual aspect to computer science, but just between you and me, there certainly is. (If you have not read Robert Pirsig's *Zen and the Art of Motorcycle Maintenance*, I recommend that you do so.)

Ruby arose from the human urge to create things that are useful and beautiful. Programs written in Ruby should spring from the same source. That, to me, is the essence of the Ruby Way.

This page intentionally left blank

CHAPTER 1
Ruby in Review

Language shapes the way we think and determines what we can think about.

—Benjamin Lee Whorf

It is worth remembering that a new programming language is sometimes viewed as a panacea, especially by its adherents. But no one language will supplant all the others; no one tool is unarguably the best for every possible task. There are many different problem domains in the world and many possible constraints on problems within those domains.

Above all, there are different ways of *thinking* about these problems, stemming from the diverse backgrounds and personalities of the programmers themselves. For these reasons, there is no foreseeable end to the proliferation of languages. And as long as there is a multiplicity of languages, there will be a multiplicity of personalities defending and attacking them. In short, there will always be "language wars"; in this book, however, we do not intend to participate in them.

Yet in the constant quest for newer and better program notations, we have stumbled across ideas that endure, that transcend the context in which they were created. Just as Pascal borrowed from Algol, just as Java borrowed from C, so will every language borrow from its predecessors.

A language is both a toolbox and a playground; it has a practical side, but it also serves as a test bed for new ideas that may or may not be widely accepted by the computing community.

One of the most far-reaching of these ideas is the concept of object-oriented programming (OOP). Although many would argue that the overall significance of OOP is evolutionary rather than revolutionary, no one can say that it has not had an impact on the industry. Twenty-five years ago, object orientation was for the most part an academic curiosity; today it is a universally accepted paradigm.

In fact, the ubiquitous nature of OOP has led to a significant amount of "hype" in the industry. In a classic paper of the late 1980s, Roger King observed, "If you want to sell a cat to a computer scientist, you have to tell him it's object-oriented." Additionally, there are differences of opinion about what OOP really is, and even among those who are essentially in agreement, there are differences in terminology.

It is not our purpose here to contribute to the hype. We do find OOP to be a useful tool and a meaningful way of thinking about problems; we do not claim that it cures cancer.

As for the exact nature of OOP, we have our pet definitions and favorite terminology; but we make these known only to communicate effectively, not to quibble over semantics.

We mention all this because it is necessary to have a basic understanding of OOP to proceed to the bulk of this book and understand the examples and techniques. Whatever else might be said about Ruby, it is definitely an object-oriented language.

1.1 An Introduction to Object Orientation

Before talking about Ruby specifically, it is a good idea to talk about object-oriented programming in the abstract. These first few pages review those concepts with only cursory references to Ruby before we proceed to a review of the Ruby language itself.

1.1.1 What Is an Object?

In object-oriented programming, the fundamental unit is the *object*. An object is an entity that serves as a container for data and also controls access to the data. Associated with an object is a set of *attributes,* which are essentially no more than variables belonging to the object. (In this book, we will loosely use the ordinary term *variable* for an attribute.) Also associated with an object is a set of functions that provide an interface to the functionality of the object, called *methods.*

It is essential that any OOP language provide *encapsulation.* As the term is commonly used, it means first that the attributes and methods of an object are associated specifically with that object, or bundled with it; second, it means that the scope of

those attributes and methods is by default the object itself (an application of the principle of *data hiding*).

An object is considered to be an instance or manifestation of an *object class* (usually simply called a *class*). The class may be thought of as the blueprint or pattern; the object itself is the thing created from that blueprint or pattern. A class is often thought of as an *abstract type*—a more complex type than, for example, an integer or character string.

When an object (an instance of a class) is created, it is said to be *instantiated.* Some languages have the notion of an explicit *constructor* and *destructor* for an object—functions that perform whatever tasks are needed to initialize an object and (respectively) to "destroy" it. We may as well mention prematurely that Ruby has what might be considered a constructor but certainly does not have any concept of a destructor (because of its well-behaved garbage collection mechanism).

Occasionally a situation arises in which a piece of data is more "global" in scope than a single object, and it is inappropriate to put a copy of the attribute into each instance of the class. For example, consider a class called `MyDogs`, from which three objects are created: `fido`, `rover`, and `spot`. For each dog, there might be such attributes as age and date of vaccination. But suppose that we want to store the owner's name (the owner of *all* the dogs). We could certainly put it in each object, but that is wasteful of memory and at the very least a misleading design. Clearly the *owner_name* attribute belongs not to any individual object but to the class itself. When it is defined that way (and the syntax varies from one language to another), it is called a class attribute (or *class variable*).

Of course, there are many situations in which a class variable might be needed. For example, suppose that we wanted to keep a count of how many objects of a certain class had been created. We could use a class variable that was initialized to zero and incremented with every instantiation; the class variable would be associated with the class and not with any particular object. In scope, this variable would be just like any other attribute, but there would only be one copy of it for the entire class and the entire set of objects created from that class.

To distinguish between class attributes and ordinary attributes, the latter are sometimes explicitly called *object attributes* (or *instance attributes*). We use the convention that any attribute is assumed to be an instance attribute unless we explicitly call it a class attribute.

Just as an object's methods are used to control access to its attributes and provide a clean interface to them, so is it sometimes appropriate or necessary to define a method associated with a class. A *class method*, not surprisingly, controls access to the

class variables and also performs any tasks that might have classwide effects rather than merely objectwide. As with data attributes, methods are assumed to belong to the object rather than the class unless stated otherwise.

It is worth mentioning that there is a sense in which all methods are class methods. We should not suppose that when 100 objects are created, we actually copy the code for the methods 100 times! But the rules of scope assure us that each object method operates only on the object whose method is being called, providing us with the necessary illusion that object methods are associated strictly with their objects.

1.1.2 Inheritance

We come now to one of the real strengths of OOP, which is *inheritance.* Inheritance is a mechanism that allows us to extend a previously existing entity by adding features to create a new entity. In short, inheritance is a way of reusing code. (Easy, effective code reuse has long been the Holy Grail of computer science, resulting in the invention decades ago of parameterized subroutines and code libraries. OOP is only one of the later efforts in realizing this goal.)

Typically we think of inheritance at the class level. If we have a specific class in mind, and there is a more general case already in existence, we can define our new class to inherit the features of the old one. For example, suppose that we have a class named `Polygon` that describes convex polygons. If we then find ourselves dealing with a `Rectangle` class, we can inherit from `Polygon` so that `Rectangle` has all the attributes and methods that `Polygon` has. For example, there might be a method that calculates perimeter by iterating over all the sides and adding their lengths. Assuming that everything was implemented properly, this method would automatically work for the new class; the code would not have to be rewritten.

When a class B inherits from a class A, we say that B is a *subclass* of A, or conversely A is the *superclass* of B. In slightly different terminology, we may say that A is a *base class* or *parent class,* and B is a *derived class* or *child class.*

A derived class, as we have seen, may treat a method inherited from its base class as if it were its own. On the other hand, it may redefine that method entirely if it is necessary to provide a different implementation; this is referred to as *overriding* a method. In addition, most languages provide a way for an overridden method to call its namesake in the parent class; that is, the method `foo` in B knows how to call method `foo` in A if it wants to. (Any language that does not provide this feature is under suspicion of not being truly object oriented.) Essentially the same is true for data attributes.

The relationship between a class and its superclass is interesting and important; it is usually described as the *is-a* relationship, because a `Square` "is a" `Rectangle`, and a `Rectangle` "is a" `Polygon`, and so on. Thus, if we create an inheritance hierarchy (which tends to exist in one form or another in any OOP language), we see that the more specific entity "is a" subclass of the more general entity at any given point in the hierarchy. Note that this relationship is transitive—in the previous example, we easily see that a `Square` "is a" `Polygon`. Note also that the relationship is not commutative—we know that every `Rectangle` is a `Polygon`, but not every `Polygon` is a `Rectangle`.

This brings us to the topic of *multiple inheritance* (MI). It is conceivable that a new class could inherit from more than one class. For example, the classes `Dog` and `Cat` can both inherit from the class `Mammal`, and `Sparrow` and `Raven` can inherit from `WingedCreature`. But what if we want to define a `Bat`? It can reasonably inherit from both the classes `Mammal` and `WingedCreature`. This corresponds well with our real-life experience in which things are not members of just one category but of many non-nested categories.

MI is probably the most controversial area in OOP. One camp will point out the potential for ambiguity that must be resolved. For example, if `Mammal` and `WingedCreature` both have an attribute called `size` (or a method called `eat`), which one will be referenced when we refer to it from a `Bat` object? Another related difficulty is the *diamond inheritance problem*—so called because of the shape of its inheritance diagram, with both superclasses inheriting from a single common superclass. For example, imagine that `Mammal` and `WingedCreature` both inherit from `Organism`; the hierarchy from `Organism` to `Bat` forms a diamond. But what about the attributes that the two intermediate classes both inherit from their parent? Does `Bat` get two copies of each of them? Or are they merged back into single attributes because they come from a common ancestor in the first place?

These are both issues for the language designer rather than the programmer. Different OOP languages deal with the issues differently. Some provide rules allowing one definition of an attribute to "win out," or a way to distinguish between attributes of the same name, or even a way of aliasing or renaming the identifiers. This in itself is considered by many to be an argument against MI—the mechanisms for dealing with name clashes and the like are not universally agreed upon but are language dependent. C++ offers a minimal set of features for dealing with ambiguities; those of Eiffel are probably better, and those of Perl are different from both.

The alternative, of course, is to disallow MI altogether. This is the approach taken by such languages as Java and Ruby. This sounds like a drastic compromise; however,

as we shall see later, it is not as bad as it sounds. We will look at a viable alternative to traditional MI, but we must first discuss polymorphism, yet another OOP buzzword.

1.1.3 Polymorphism

Polymorphism is the term that perhaps inspires the most semantic disagreement in the field. Everyone seems to know what it is, but everyone has a different definition. (In recent years, "What is polymorphism?" has become a popular interview question. If it is asked of you, I recommend quoting an expert such as Bertrand Meyer or Bjarne Stroustrup; that way, if the interviewer disagrees, his beef is with the expert and not with you.)

The literal meaning of polymorphism is "the ability to take on multiple forms or shapes." In its broadest sense, this refers to the ability of different objects to respond in different ways to the same message (or method invocation).

Damian Conway, in his book *Object-Oriented Perl*, distinguishes meaningfully between two kinds of polymorphism. The first, *inheritance polymorphism*, is what most programmers are referring to when they talk about polymorphism.

When a class inherits from its superclass, we know (by definition) that any method present in the superclass is also present in the subclass. Thus, a chain of inheritance represents a linear hierarchy of classes that can respond to the same set of methods. Of course, we must remember that any subclass can redefine a method; that is what gives inheritance its power. If I call a method on an object, typically it will be either the one it inherited from its superclass or a more appropriate (more specialized) method tailored for the subclass.

In statically typed languages such as C++, inheritance polymorphism establishes type compatibility down the chain of inheritance (but not in the reverse direction). For example, if B inherits from A, a pointer to an A object can also point to a B object; but the reverse is not true. This type compatibility is an essential OOP feature in such languages—indeed it almost sums up polymorphism—but polymorphism certainly exists in the absence of static typing (as in Ruby).

The second kind of polymorphism Conway identifies is *interface polymorphism.* This does not require any inheritance relationship between classes; it only requires that the interfaces of the objects have methods of a certain name. The treatment of such objects as being the same "kind" of thing is thus a kind of polymorphism (though in most writings, it is not explicitly referred to as such).

Readers familiar with Java will recognize that it implements both kinds of polymorphism. A Java class can extend another class, inheriting from it via the `extends` keyword; or it may implement an interface, acquiring a known set of methods (which

must then be overridden) via the `implements` keyword. Because of syntax requirements, the Java interpreter can determine at compile time whether a method can be invoked on a particular object.

Ruby supports interface polymorphism but in a different way, providing *modules* whose methods may be *mixed in* to existing classes (interfacing to user-defined methods that are expected to exist). This, however, is not the way modules are usually used. A module consists of methods and constants that may be used as though they were actual parts of that class or object; when a module is mixed in via the `include` statement, this is considered to be a restricted form of multiple inheritance. According to the language designer, Yukihiro Matsumoto (often called Matz), it can be viewed as *single inheritance with implementation sharing*. This is a way of preserving the benefits of MI without suffering all the consequences.

1.1.4 A Few More Terms

In languages such as C++, there is the concept of *abstract* classes—classes that must be inherited from and cannot be instantiated on their own. This concept does not exist in the more dynamic Ruby language, although if the programmer really wants, it is possible to fake this kind of behavior by forcing the methods to be overridden. Whether this is useful is left as an exercise for the reader.

The creator of C++, Bjarne Stroustrup, also identifies the concept of a *concrete type*. This is a class that exists only for convenience; it is not designed to be inherited from, nor is it expected that there will ever be another class derived from it. In other words, the benefits of OOP are basically limited to encapsulation. Ruby does not specifically support this concept through any special syntax (nor does C++), but it is naturally well suited for the creation of such classes.

Some languages are considered to be more "pure" OO than others. (We also use the term *radically object oriented*.) This refers to the concept that *every* entity in the language is an object; every primitive type is represented as a full-fledged class, and variables and constants alike are recognized as object instances. This is in contrast to such languages as Java, C++, and Eiffel. In these, the more primitive data types (especially constants) are not first-class objects, though they may sometimes be treated that way with "wrapper" classes. Arguably there are languages that are *more* radically object oriented than Ruby, but they are relatively few.

Most OO languages are static; the methods and attributes belonging to a class, the global variables, and the inheritance hierarchy are all defined at compile time. Perhaps the largest conceptual leap for a Ruby programmer is that these are all handled *dynamically* in Ruby. Definitions and even inheritance can happen at runtime—

in fact, we can truly say that every declaration or definition is actually *executed* during the running of the program. Among many other benefits, this obviates the need for conditional compilation and can produce more efficient code in many circumstances.

This sums up the whirlwind tour of OOP. Throughout the rest of the book, we have tried to make consistent use of the terms introduced here. Let's proceed now to a brief review of the Ruby language itself.

1.2 Basic Ruby Syntax and Semantics

In the previous pages, we have already seen that Ruby is a *pure, dynamic* OOP language. Let's look briefly at some other attributes before summarizing the syntax and semantics.

Ruby is an *agile* language. It is "malleable" and encourages frequent, easy (manual) refactoring.

Ruby is an *interpreted* language. Of course, there may be later implementations of a Ruby compiler for performance reasons, but we maintain that an interpreter yields great benefits not only in rapid prototyping but also in the shortening of the development cycle overall.

Ruby is an *expression-oriented* language. Why use a statement when an expression will do? This means, for instance, that code becomes more compact as the common parts are factored out and repetition is removed.

Ruby is a *very high-level language* (VHLL). One principle behind the language design is that the computer should work for the programmer rather than vice versa. The "density" of Ruby means that sophisticated and complex operations can be carried out with relative ease as compared to lower-level languages.

Let's start by examining the overall look and feel of the language and some of its terminology. We'll briefly examine the nature of a Ruby program before looking at examples.

To begin with, Ruby is essentially a line-oriented language—more so than languages such as C but not so much as antique languages such as FORTRAN. Tokens can be crowded onto a single line as long as they are separated by whitespace as needed. Statements may share a single line if they are separated by semicolons; this is the only time the terminating semicolon is really needed. A line may be continued to the next line by ending it with a backslash or by letting the parser know that the statement is not complete—for example, by ending a line with a comma.

There is no main program as such; execution proceeds in general from top to bottom. In more complex programs, there may be numerous definitions at the top, followed by the (conceptual) main program at the bottom; but even in that case, execution proceeds from the top down because definitions in Ruby are executed.

1.2.1 Keywords and Identifiers

The keywords (or reserved words) in Ruby typically cannot be used for other purposes. These are as follows:

```
BEGIN     END       alias     and       begin
break     case      class     def       defined?
do        else      elsif     end       ensure
false     for       if        in        module
next      nil       not       or        redo
rescue    retry     return    self      super
then      true      undef     unless    until
when      while     yield
```

Variables and other identifiers normally start with an alphabetic letter or a special modifier. The basic rules are as follows:

- Local variables (and pseudovariables such as `self` and `nil`) begin with a lowercase letter or an underscore.
- Global variables begin with `$` (a dollar sign).
- Instance variables (within an object) begin with `@` (an at sign).
- Class variables (within a class) begin with `@@` (two at signs).
- Constants begin with capital letters.
- For purposes of forming identifiers, the underscore (_) may be used as a lowercase letter.
- Special variables starting with a dollar sign (such as `$1` and `$/`) are set by the Ruby interpreter itself.

Here are some examples of each of these:

- **Local variables** `alpha _ident some_var`
- **Pseudovariables** `self nil __FILE__`
- **Constants** `K9chip Length LENGTH`
- **Instance variables** `@foobar @thx1138 @NOT_CONST`
- **Class variable** `@@phydeaux @@my_var @@NOT_CONST`
- **Global variables** `$beta $B12vitamin $NOT_CONST`

1.2.2 Comments and Embedded Documentation

Comments in Ruby begin with a pound sign (#) outside a string or character constant and proceed to the end of the line:

```
x = y + 5 # This is a comment.
# This is another comment.
puts "# But this isn't."
```

Comments immediately before definitions typically document the thing that is about to be defined. This embedded documentation can often be retrieved from the program text by external tools. Typical documentation comments can run to several comment lines in a row.

```
# The purpose of this class
# is to cure cancer
# and instigate world peace
class ImpressiveClass
```

Given two lines starting with `=begin` and `=end`, everything between those lines (inclusive) is treated as a comment. (These can't be preceded by whitespace.)

```
=begin
Everything on lines
inside here will be a
comment as well.
=end
```

1.2.3 Constants, Variables, and Types

In Ruby, variables do not have types, but the objects they refer to do have types. The simplest data types are character, numeric, and string.

Numeric constants are mostly intuitive, as are strings. Generally, a double-quoted string is subject to additional interpretation, and a single-quoted string is more "as is," allowing only an escaped backslash.

In double-quoted strings, we can do "interpolation" of variables and expressions, as shown here:

```
a = 3
b = 79
puts "#{a} times #{b} = #{a*b}"   #  3 times 79 = 237
```

For more information on literals (numbers, strings, regular expressions, and so on), refer to later chapters.

There is a special kind of string worth mentioning, primarily useful in small scripts used to glue together larger programs. The command output string is sent to the operating system as a command to be executed, whereupon the output of the command is substituted back into the string. The simple form of this string uses the *grave accent* (sometimes called a *back-tick* or *back-quote*) as a beginning and ending delimiter; the more complex form uses the `%x` notation:

```
'whoami'
'ls -l'
%x[grep -i meta *.html | wc -l]
```

Regular expressions in Ruby look similar to character strings, but they are used differently. The usual delimiter is a slash character.

For those familiar with Perl, regular expression handling is similar in Ruby. Incidentally, we'll use the abbreviation *regex* throughout the remainder of the book; many people abbreviate it as *regexp,* but that is not as pronounceable. For details on regular expressions, see Chapter 3, "Working with Regular Expressions."

Arrays in Ruby are a powerful construct; they may contain data of any type or may even mix types. As we shall see in Chapter 8, "Arrays, Hashes, and Other Enumerables," all arrays are instances of the class `Array` and thus have a rich set of methods that can operate on them. An array constant is delimited by brackets; the following are all valid array expressions:

```
[1, 2, 3]
[1, 2, "buckle my shoe"]
[1, 2, [3,4], 5]
["alpha", "beta", "gamma", "delta"]
```

The second example shows an array containing both integers and strings; the third example in the preceding code shows a nested array, and the fourth example shows an array of strings. As in most languages, arrays are zero based; for instance, in the last array in the preceding code, `"gamma"` is element number 2. Arrays are dynamic and do not need to have a size specified when they are created.

Because the array of strings is so common (and so inconvenient to type), a special syntax has been set aside for it, similar to what we have seen already:

```
%w[alpha beta gamma delta]
%w(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec)
%w/am is are was were be being been/
```

Such a shorthand is frequently called "syntax sugar" because it offers a more convenient alternative to another syntactic form. In this case, the quotes and commas are not needed; only whitespace separates the individual elements. In the case of an element that contains whitespace, of course, this would not work.

An array variable can use brackets to index into the array. The resulting expression can be both examined and assigned to:

```
val = myarray[0]
print stats[j]
x[i] = x[i+1]
```

Another powerful construct in Ruby is the *hash,* also known in other circles as an *associative array* or *dictionary.* A hash is a set of associations between paired pieces of data; it is typically used as a lookup table or a kind of generalized array in which the index need not be an integer. Each hash is an instance of the class `Hash`.

A hash constant is typically represented between delimiting braces, with the symbol `=>` separating the individual keys and values. The key can be thought of as an index where the corresponding value is stored. There is no restriction on types of the keys or the corresponding values. Here are some hashes:

```
{1 => 1, 2 => 4, 3 => 9, 4 => 16, 5 => 25, 6 => 36}
{"cat" => "cats", "ox" => "oxen", "bacterium" => "bacteria"}
{"odds" => [1,3,5,7], "evens" => [2,4,6,8]}
{"foo" => 123, [4,5,6] => "my array", "867-5309" => "Jenny"}
```

Hashes also have an additional syntax that creates keys that are instances of the `Symbol` class (which is explained further in later material):

```
{hydrogen: 1, helium: 2, carbon: 12}
```

A hash variable can have its contents accessed by essentially the same bracket notation that arrays use:

```
print phone_numbers["Jenny"]
plurals["octopus"] = "octopi"
atomic_numbers[:helium] #=> 2
```

It should be stressed, however, that both arrays and hashes have many methods associated with them; these methods give them their real usefulness. The section "OOP in Ruby," later in the chapter, will expand on this a little more.

1.2.4 Operators and Precedence

Now that we have established our most common data types, let's look at Ruby's operators. They are arranged here in order from highest to lowest precedence:

`::`	Scope
`[]`	Indexing
`**`	Exponentiation
`+ - ! ~`	Unary positive/negative, not, …
`* / %`	Multiplication, division, …
`+ -`	Addition/subtraction
`<< >>`	Logical shifts, …
`&`	Bitwise AND
`| ^`	Bitwise OR, XOR
`> >= < <=`	Comparison
`== === <=> != =~ !~`	Equality, inequality, …
`&&`	Boolean AND
`||`	Boolean OR

`.. ...`	Range operators
= (also +=, -=, …)	Assignment
`?:`	Ternary decision
`not`	Boolean negation
`and or`	Boolean AND, OR

Some of the preceding symbols serve more than one purpose; for example, the operator << is a bitwise left shift but is also an append operator (for arrays, strings, and so on) and a marker for a here-document. Likewise, the + is for numeric addition as well as for string concatenation. As we shall see later, many of these operators are just shortcuts for method names.

Now we have defined most of our data types and many of the possible operations on them. Before going any further, let's look at a sample program.

1.2.5 A Sample Program

In a tutorial, the first program is always `Hello, world!` But in a whirlwind tour like this one, let's start with something slightly more advanced. Here is a small interactive console-based program to convert between Fahrenheit and Celsius temperatures:

```
print "Please enter a temperature and scale (C or F): "
STDOUT.flush
str = gets
exit if str.nil? || str.empty?
str.chomp!
temp, scale = str.split(" ")

abort "#{temp} is not a valid number." if temp !~ /-?\d+/

temp = temp.to_f
case scale
  when "C", "c"
    f = 1.8*temp + 32
  when "F", "f"
    c = (5.0/9.0)*(temp-32)
else
  abort "Must specify C or F."
end

if f.nil?
  puts "#{c} degrees C"
```

```
else
  puts "#{f} degrees F"
end
```

Here are some examples of running this program. These show that the program can convert from Fahrenheit to Celsius, convert from Celsius to Fahrenheit, and handle an invalid scale or an invalid number:

```
Please enter a temperature and scale (C or F): 98.6 F
37.0 degrees C

Please enter a temperature and scale (C or F): 100 C
212.0 degrees F

Please enter a temperature and scale (C or F): 92 G
Must specify C or F.

Please enter a temperature and scale (C or F): junk F
junk is not a valid number.
```

Now, as for the mechanics of the program: We begin with a `print` statement, which is actually a call to the `Kernel` method `print`, to write to standard output. This is an easy way of leaving the cursor "hanging" at the end of the line.

Following this, we call `gets` (get string from standard input), assigning the value to `str`. We then do a `chomp!` to remove the trailing newline.

Note that any apparently "free-standing" function calls such as `print` and `gets` are actually methods of `Object` (probably originating in `Kernel`). In the same way, `chomp` is a method called with `str` as a receiver. Method calls in Ruby usually can omit the parentheses; for example, `print "foo"` is the same as `print("foo")`.

The variable `str` refers to (or informally, it "holds") a character string, but there is no reason it could not hold some other type instead. In Ruby, data have types, but variables do not. A variable springs into existence as soon as the interpreter sees an assignment to that variable; there are no "variable declarations" as such.

The `exit` is a call to a method that terminates the program. On this same line there is a control structure called an *if-modifier*. This is like the `if` statement that exists in most languages, but backwards; it comes after the action, does not permit an `else`, and does not require closing. As for the condition, we are checking two things: Does `str` have a value (is it non-`nil`) and is it a non-null string? In the case of an immediate end-of-file, our first condition will hold; in the case of a newline with no preceding data, the second condition will hold.

The || operator has the same effect as or, but is preferred because it has higher precedence and produces less-confusing results. The same statement could be written this way:

```
exit if not str or not str[0]
```

The reason these tests work is that a variable can have a nil value, and nil evaluates to false in Ruby. In fact, nil and false evaluate as false, and everything else evaluates as true. Specifically, the null string "" and the number 0 do *not* evaluate as false.

The next statement performs a chomp! operation on the string (to remove the trailing newline). The exclamation point as a prefix serves as a warning that the operation actually changes the value of its receiver rather than just returning a value. The exclamation point is used in many such instances to remind the programmer that a method has a side effect or is more "dangerous" than its unmarked counterpart. The method chomp, for example, returns the same result but does not modify its receiver.

The next statement is an example of multiple assignment. The split method splits the string into an array of values, using the space as a delimiter. The two assignable entities on the left-hand side will be assigned the respective values resulting on the right-hand side.

The if statement that follows uses a simple regex to determine whether the number is valid; if the string fails to match a pattern consisting of an optional minus sign followed by one or more digits, it is an invalid number (for our purposes), and the program exits. Note that the if statement is terminated by the keyword end; though it was not needed here, we could have had an else clause before the end. The keyword then is optional; we tend not to use it in this book.

The to_f method is used to convert the string to a floating point number. We are actually assigning this floating point value back to temp, which originally held a string.

The case statement chooses between three alternatives—the cases in which the user specified a C, specified an F, or used an invalid scale. In the first two instances, a calculation is done; in the third, we print an error and exit. When printing, the puts method will automatically add a newline after the string that is given.

Ruby's case statement, by the way, is far more general than the example shown here. There is no limitation on the data types, and the expressions used are all arbitrary and may even be ranges or regular expressions.

There is nothing mysterious about the computation. But consider the fact that the variables `c` and `f` are referenced first inside the branches of the case. There are no declarations as such in Ruby; because a variable only comes into existence when it is assigned, this means that when we fall through the `case` statement, only one of these variables actually has a valid value.

We use this fact to determine after the fact which branch was followed, so that we can do a slightly different output in each instance. Testing `f` for a `nil` is effectively a test of whether the variable has a meaningful value. We do this here only to show that it can be done; obviously, two different `print` statements could be used inside the `case` statement if we wanted.

The perceptive reader will notice that we used only "local" variables here. This might be confusing because their scope certainly appears to cover the entire program. What is happening here is that the variables are all local to the *top level* of the program (written *toplevel* by some). The variables appear global because there are no lower-level contexts in a program this simple; but if we declared classes and methods, these top-level variables would not be accessible within those.

1.2.6 Looping and Branching

Let's spend some time looking at control structures. We have already seen the simple `if` statement and the `if`-modifier; there are also corresponding structures based on the keyword `unless` (which also has an optional `else`), as well as expression-oriented forms of `if` and `unless`. To summarize these forms, these two statements are equivalent:

```
if x < 5
  statement1
end


unless x >= 5
  statement1
end
```

And so are these:

```
if x < 5
  statement1
else
  statement2
end
```

```
unless x < 5
  statement2
else
  statement1
end
```

And these:

```
statement1 if y == 3

statement1 unless y != 37.0
```

And these are also equivalent:

```
x = if a > 0 then b else c end

x = unless a <= 0 then c else b end
```

Note that the keyword `then` may always be omitted except in the final (expression-oriented) cases. Note also that the modifier form cannot have an `else` clause.

The `case` statement in Ruby is more powerful than in most languages. This multiway branch can even test for conditions other than equality—for example, a matched pattern. The test used by the `case` statement is called the *case equality operator* (`===`), and its behavior varies from one object to another. Let's look at this example:

```
case "This is a character string."
when "some value"
  puts "Branch 1"
when "some other value"
  puts "Branch 2"
when /char/
  puts "Branch 3"
else
  puts "Branch 4"
end
```

The preceding code prints `Branch 3`. Why? It first tries to check for equality between the tested expression and one of the strings `"some value"` or `"some other value"`; this fails, so it proceeds. The third test is for a pattern within the string; `when /char/` is equivalent to `if /char/ === "This is a character string."`. The

test succeeds, and the third `print` statement is performed. The `else` clause always handles the default case in which none of the preceding tests succeeds.

If the tested expression is an integer, the compared value can be an integer range (for example, `3..8`). In this case, the expression is tested for membership in that range. In all instances, the first successful branch will be taken.

Although the `case` statement usually behaves predictably, there are a few subtleties you should appreciate. We will look at these later.

As for looping mechanisms, Ruby has a rich set. The `while` and `until` control structures are both pretest loops, and both work as expected: One specifies a continuation condition for the loop, and the other specifies a termination condition. They also occur in "modifier" form, such as `if` and `unless`. There is also the `loop` method of the `Kernel` module (by default an infinite loop), and there are iterators associated with various classes.

The examples here assume an array called `list`, defined something like this:

```
list = %w[alpha bravo charlie delta echo]
```

They all step through the array and write out each element.

```
i = 0                       # Loop 1 (while)
while i < list.size do
  print "#{list[i]} "
  i += 1
end


i = 0                       # Loop 2 (until)
until i == list.size do
  print "#{list[i]} "
  i += 1
end

i = 0                       # Loop 3 (post-test while)
begin
  print "#{list[i]} "
  i += 1
end while i < list.size

i = 0                       # Loop 4 (post-test until)
begin
  print "#{list[i]} "
```

```
  i += 1
end until i == list.size

for x in list do          # Loop 5 (for)
  print "#{x} "
end

list.each do |x|          # Loop 6 ('each' iterator)
  print "#{x} "
end

i = 0                     # Loop 7 ('loop' method)
n=list.size-1
loop do
  print "#{list[i]} "
  i += 1
  break if i > n
end

i = 0                     # Loop 8 ('loop' method)
n=list.size-1
loop do
  print "#{list[i]} "
  i += 1
  break unless i <= n
end

n=list.size               # Loop 9 ('times' iterator)
n.times do |i|
  print "#{list[i]} "
end

n = list.size-1           # Loop 10 ('upto' iterator)
0.upto(n) do |i|
  print "#{list[i]} "
end

n = list.size-1           # Loop 11 (for)
for i in 0..n do
  print "#{list[i]} "
end

list.each_index do |x|    # Loop 12 ('each_index' iterator)
```

```
  print "#{list[x]} "
end
```

Let's examine these in detail. Loops 1 and 2 are the "standard" forms of the `while` and `until` loops; they behave essentially the same, but their conditions are negations of each other. Loops 3 and 4 are the same thing in "post-test" versions; the test is performed at the end of the loop rather than at the beginning. Note that the use of `begin` and `end` in this context is strictly a kludge or hack; what is really happening is that a `begin`/`end` block (used for exception handling) is followed by a `while` or `until` modifier. In other words, this is only an illustration. Don't code this way.

Loop 6 is arguably the "proper" way to write this loop. Note the simplicity of 5 and 6 compared with the others; there is no explicit initialization and no explicit test or increment. This is because an array "knows" its own size, and the standard iterator `each` (loop 6) handles such details automatically. Indeed, loop 3 is merely an indirect reference to this same iterator because the `for` loop works for any object having the iterator each defined. The `for` loop is only another way to call `each`.

Loops 7 and 8 both use the loop construct; as stated previously, `loop` looks like a keyword introducing a control structure, but it is really a method of the module `Kernel`, not a control structure at all.

Loops 9 and 10 take advantage of the fact that the array has a numeric index; the `times` iterator executes a specified number of times, and the `upto` iterator carries its parameter up to the specified value. Neither of these is truly suitable for this instance.

Loop 11 is a `for` loop that operates specifically on the index values, using a range, and loop 12 likewise uses the `each_index` iterator to run through the list of array indices.

In the preceding examples, we have not laid enough emphasis on the "modifier" form of the `while` and `until` loops. These are frequently useful, and they have the virtue of being concise. These two additional fragments both mean the same:

```
perform_task() until finished

perform_task() while not finished
```

Another fact is largely ignored in these examples: Loops do not always run smoothly from beginning to end, in a predictable number of iterations, or ending in a single predictable way. We need ways to control these loops further.

The first way is the `break` keyword, shown in loops 7 and 8. This is used to "break out" of a loop; in the case of nested loops, only the innermost one is halted. This will be intuitive for C programmers.

The `redo` keyword is jumps to the start of the loop body in `while` and `until` loops.

The `next` keyword effectively jumps to the end of the innermost loop and resumes execution from that point. It works for any loop or iterator.

The iterator is an important concept in Ruby, as we have already seen. What we have not seen is that the language allows user-defined iterators in addition to the pre-defined ones.

The default iterator for any object is called `each`. This is significant partly because it allows the `for` loop to be used. But iterators may be given different names and used for varying purposes.

It is also possible to pass parameters via `yield`, which will be substituted into the block's parameter list (between vertical bars). As a somewhat contrived example, the following iterator does nothing but generate integers from 1 to 10, and the call of the iterator generates the first ten cubes:

```
def my_sequence
  (1..10).each do |i|
    yield i
  end
end

my_sequence {|x| puts x**3 }
```

Note that `do` and `end` may be substituted for the braces that delimit a block. There are differences, but they are fairly subtle.

1.2.7 Exceptions

Ruby supports *exceptions*, which are standard means of handling unexpected errors in modern programming languages.

By using exceptions, special return codes can be avoided, as well as the nested `if else` "spaghetti logic" that results from checking them. Even better, the code that detects the error can be distinguished from the code that knows how to handle the error (because these are often separate anyway).

The `raise` statement raises an exception. Note that `raise` is not a reserved word but a method of the module `Kernel`. (There is an alias named `fail`.)

```
raise                                    # Example 1
raise "Some error message"               # Example 2
raise ArgumentError                      # Example 3
raise ArgumentError, "Bad data"          # Example 4
raise ArgumentError.new("Bad data")      # Example 5
raise ArgumentError, "Bad data", caller[0] # Example 6
```

In the first example in the preceding code, the last exception encountered is re-raised. In example 2, a `RuntimeError` (the default error) is created using the string `Some error message`.

In example 3, an `ArgumentError` is raised; in example 4, this same error is raised with the message "Bad data" associated with it. Example 5 behaves exactly the same as example 4. Finally, example 6 adds traceback information of the form `"filename:line"` or `"filename:line:in 'method'"` (as stored in the `caller` array).

Now, how do we handle exceptions in Ruby? The `begin-end` block is used for this purpose. The simplest form is a `begin-end` block with nothing but our code inside:

```
begin
  # Just runs our code.
  # ...
end
```

This is of no value in catching errors. The block, however, may have one or more `rescue` clauses in it. If an error occurs at any point in the code, between `begin` and `rescue`, control will be passed immediately to the appropriate `rescue` clause:

```
begin
  x = Math.sqrt(y/z)
  # ...
rescue ArgumentError
  puts "Error taking square root."
rescue ZeroDivisionError
  puts "Attempted division by zero."
end
```

Essentially the same thing can be accomplished by this fragment:

```
begin
  x = Math.sqrt(y/z)
  # ...
rescue => err
  puts err
end
```

Here, the variable `err` is used to store the value of the exception; printing it causes it to be translated to some meaningful character string. Note that because the error type is not specified, the `rescue` clause will catch any descendant of `StandardError`. The notation `rescue => variable` can be used with or without an error type before the `=>` symbol.

In the event that error types are specified, it may be that an exception does not match any of these types. For that situation, we are allowed to use an `else` clause after all the `rescue` clauses:

```
begin
  # Error-prone code...
rescue Type1
  # ...
rescue Type2
  # ...
else
  # Other exceptions...
end
```

In many cases, we want to do some kind of recovery. In that event, the keyword `retry` (within the body of a `rescue` clause) restarts the `begin` block and tries those operations again:

```
begin
  # Error-prone code...
rescue
  # Attempt recovery...
  retry  # Now try again
end
```

Finally, it is sometimes necessary to write code that "cleans up" after a `begin-end` block. In the event this is necessary, an `ensure` clause can be specified:

```
begin
  # Error-prone code...
rescue
  # Handle exceptions
ensure
  # This code is always executed
end
```

The code in an `ensure` clause is always executed before the `begin-end` block exits. This happens regardless of whether an exception occurred.

Exceptions may be caught in two other ways. First, there is a modifier form of the rescue clause:

```
x = a/b rescue puts("Division by zero!")
```

In addition, the body of a method definition is an implicit `begin-end` block; the `begin` is omitted, and the entire body of the method is subject to exception handling, ending with the `end` of the method:

```
def some_method
  # Code...
rescue
  # Recovery...
end
```

This sums up the basics of exception handling as well as the discussion of fundamental syntax and semantics.

There are numerous aspects of Ruby we have not discussed here. The rest of this chapter is devoted to the more advanced features of the language, including a collection of Ruby lore that will help the intermediate programmer learn to "think in Ruby."

1.3 OOP in Ruby

Ruby has all the elements more generally associated with OOP languages, such as objects with encapsulation and data hiding, methods with polymorphism and overriding, and classes with hierarchy and inheritance. It goes further and adds limited metaclass features, singleton methods, modules, and mixins.

Similar concepts are known by other names in other OOP languages, but concepts of the same name may have subtle differences from one language to another. This section elaborates on the Ruby understanding and usage of these elements of OOP.

1.3.1 Objects

In Ruby, all numbers, strings, arrays, regular expressions, and many other entities are actually objects. Work is done by executing the methods belonging to the object:

```
3.succ                  # 4
"abc".upcase            # "ABC"
[2,1,5,3,4].sort        # [1,2,3,4,5]
some_object.some_method  # some result
```

In Ruby, every object is an instance of some class; the class contains the implementation of the methods:

```
"abc".class        # String
"abc".class.class  # Class
```

In addition to encapsulating its own attributes and operations, an object in Ruby has an identity:

```
"abc".object_id   # 53744407
```

This object ID is usually of limited usefulness to the programmer.

1.3.2 Built-in Classes

More than 30 built-in classes are predefined in the Ruby class hierarchy. Like many other OOP languages, Ruby does not allow multiple inheritance, but that does not necessarily make it any less powerful. Modern OO languages frequently follow the single inheritance model. Ruby does support modules and mixins, which are discussed in the next section. It also implements object IDs, as we just saw, which support the implementation of persistent, distributed, and relocatable objects.

To create an object from an existing class, the `new` method is typically used:

```
myFile = File.new("textfile.txt","w")
myString = String.new("This is a string object")
```

This is not always explicitly required, however. When using *object literals*, you do not need to bother with calling `new`, as we did in the previous example:

```
your_string = "This is also a string object"
number = 5  # new not needed here, either
```

Variables are used to hold references to objects. As previously mentioned, variables themselves have no type, nor are they objects themselves; they are simply references to objects:

```
x = "abc"
```

An exception to this is that small immutable objects of some built-in classes, such as `Fixnum`, are copied directly into the variables that refer to them. (These objects are no bigger than pointers, and it is more efficient to deal with them in this way.) In this case, assignment makes a copy of the object, and the heap is not used.

Variable assignment causes object references to be shared:

```
y = "abc"
x = y
x         # "abc"
```

After `x = y` is executed, variables `x` and `y` both refer to the same object:

```
x.object_id     # 53732208
y.object_id     # 53732208
```

If the object is mutable, a modification done to one variable will be reflected in the other:

```
x.gsub!(/a/,"x")
y                 # "xbc"
```

Reassigning one of these variables has no effect on the other, however:

```
# Continuing previous example...
x = "abc"
y                 # still has value "xbc"
```

A mutable object can be made immutable using the `freeze` method:

```
x.freeze
x.gsub!(/b/,"y")  # Error!
```

A symbol is a little unusual; it's like an atom in Lisp. It acts like a kind of immutable string, and multiple uses of a symbol all reference the same value. A symbol can be converted to a string with the `to_s` method:

```
suits = [:hearts, :clubs, :diamonds, :spades]
lead = suits[1].to_s   # "clubs"
```

Similar to arrays of strings, arrays of symbols can be created using the syntax shortcut `%i`:

```
suits = %i[hearts clubs diamonds spades] # an array of symbols
```

1.3.3 Modules and Mixins

Many built-in methods are available from class ancestors. Of special note are the `Kernel` methods mixed-in to the `Object` class; because `Object` is the universal parent class, the methods added to it from `Kernel` are also universally available. These methods form an important part of Ruby.

The terms *module* and *mixin* are nearly synonymous. A module is a collection of methods and constants that is external to the Ruby program. It can be used simply for namespace management, but the most common use of a module is to have its features "mixed" into a class (by using `include`). In this case, it is used as a mixin.

This term was apparently borrowed most directly from Python. (It is sometimes written as *mix-in,* but we write it as a single word.) It is worth noting that some Lisp variants have had this feature for more than two decades.

Do not confuse this usage of the term *module* with another usage common in computing. A Ruby module is not an external source or binary file (though it may be stored in one of these). A Ruby module instead is an OOP abstraction similar to a class.

An example of using a module for namespace management is the frequent use of the `Math` module. To use the definition of pi, for example, it is not necessary to `include` the `Math` module; you can simply use `Math::PI` as the constant.

A mixin is a way of getting some of the benefits of multiple inheritance without dealing with all the difficulties. It can be considered a restricted form of multiple inheritance, but the language creator Matz has called it "single inheritance with implementation sharing."

Note that `include` adds features of a module to the current space; the `extend` method adds features of a module to an object. With `include`, the module's methods become available as instance methods; with `extend`, they become available as class methods.

We should mention that `load` and `require` do not relate to modules but rather to Ruby source and binary files (statically or dynamically loadable). A `load` operation reads a file and runs it in the current context so that its definitions become available at that point. A `require` operation is similar to a load, but it will not load a file if it has already been loaded.

The Ruby novice, especially from a C background, may be tripped up by `require` and `include`, which are basically unrelated to each other. You may easily find yourself doing a `require` followed by an `include` to use some externally stored module.

1.3.4 Creating Classes

Ruby has numerous built-in classes, and additional classes may be defined in a Ruby program. To define a new class, the following construct is used:

```
class ClassName
  # ...
end
```

The name of the class is itself a global constant and therefore must begin with an uppercase letter. The class definition can contain class constants, class variables, class methods, instance variables, and instance methods. Class-level information is available to all objects of the class, whereas instance-level information is available only to the one object.

By the way, classes in Ruby do not, strictly speaking, have names. The "name" of a class is just a constant that is a reference to an object of type `Class` (because, in Ruby, `Class` is a class). There can certainly be more than one constant referring to a class, and these can be assigned to variables just as we can with any other object (because, in Ruby, `Class` is an object). If all this confuses you, don't worry about it. For the sake of convenience, the novice can think of a Ruby class name as being like a C++ class name.

Here we define a simple class:

```
class Friend
  @@myname = "Fred" # a class variable

  def initialize(name, gender, phone)
    @name, @sex, @phone = name, gender, phone
    # These are instance variables
  end
```

```
  def hello   # an instance method
    puts "Hi, I'm #{@name}."
  end

  def Friend.our_common_friend   # a class method
    puts "We are all friends of #{@@myname}."
  end

end

f1 = Friend.new("Susan", "female", "555-0123")
f2 = Friend.new("Tom", "male", "555-4567")

f1.hello                  # Hi, I'm Susan.
f2.hello                  # Hi, I'm Tom.
Friend.our_common_friend  # We are all friends of Fred.
```

Because class-level data is accessible throughout the class, it can be initialized at the time the class is defined. If an instance method named `initialize` is defined, it is guaranteed to be executed right after an instance is allocated. The `initialize` method is similar to the traditional concept of a constructor, but it does not have to handle memory allocation. Allocation is handled internally by `new`, and deallocation is handled transparently by the garbage collector.

Now consider this fragment, and pay attention to the `getmyvar`, `setmyvar`, and `myvar=` methods:

```
class MyClass

  NAME = "Class Name" # class constant
  @@count = 0         # initialize a class variable

  def initialize      # called when object is allocated
    @@count += 1
    @myvar = 10
  end

  def self.getcount   # class method
    @@count           # class variable
  end

  def getcount        # instance returns class variable!
    @@count           # class variable
  end
```

```
  def getmyvar        # instance method
    @myvar            # instance variable
  end

  def setmyvar(val)   # instance method sets @myvar
    @myvar = val
  end

  def myvar=(val)     # Another way to set @myvar
    @myvar = val
  end
end

foo = MyClass.new  # @myvar is 10
foo.setmyvar 20    # @myvar is 20
foo.myvar = 30     # @myvar is 30
```

Instance variables are different for each object that is an instance of the class. Class variables are shared between the class itself and every instance of the class. To create a variable that belongs only to the class, use an instance variable inside a class method. This *class instance variable* will not be shared with instances and is therefore often preferred over class variables.

In the preceding code, we see that `getmyvar` returns the value of `@myvar` and that `setmyvar` sets it. (In the terminology of many programmers, these would be referred to as a *getter* and a *setter,* respectively.) These work fine, but they do not exemplify the "Ruby way" of doing things. The method `myvar=` looks like assignment overloading (though strictly speaking, it isn't); it is a better replacement for `setmyvar`, but there is a better way yet.

The class `Module` contains methods called `attr`, `attr_accessor`, `attr_reader`, and `attr_writer`. These can be used (with symbols as parameters) to automatically handle controlled access to the instance data. For example, the three methods `getmyvar`, `setmyvar`, and `myvar=` can be replaced by a single line in the class definition:

```
attr_accessor :myvar
```

This creates a method `myvar` that returns the value of `@myvar` and a method `myvar=` that enables the setting of the same variable. The methods `attr_reader` and `attr_writer` create read-only and write-only versions of an attribute, respectively.

Within the instance methods of a class, the pseudovariable `self` can be used as needed. This is only a reference to the current receiver, the object on which the instance method is invoked.

The modifying methods `private`, `protected`, and `public` can be used to control the visibility of methods in a class. (Instance variables are always private and inaccessible from outside the class, except by means of accessors.) Each of these modifiers takes a symbol like `:foo` as a parameter; if this is omitted, the modifier applies to all subsequent definitions in the class. Here is an example:

```
class MyClass

  def method1
  # ...
  end

  def method2
  # ...
  end

  def method3
  # ...
  end

  private :method1
  public :method2
  protected :method3

  private

  def my_method
  # ...
  end

  def another_method
  # ...
  end

end
```

In the preceding class, `method1` will be private, `method2` will be public, and `method3` will be protected. Because of the `private` method with no parameters, both `my_method` and `another_method` will be private.

The `public` access level is self-explanatory; there are no restrictions on access or visibility. The `private` level means that the method is accessible only within the class or its subclasses, and it is callable only in "function form"—with `self`, implicit or explicit, as a receiver. The `protected` level means that a method can be called by other objects of the class or its subclasses, unlike a private method (which can only be called on `self`).

The default visibility for the methods defined in a class is `public`. The exception is the instance-initializing method `initialize`. Methods defined at the top level are also public by default; if they are private, they can be called only in function form (as, for example, the methods defined in `Object`).

Ruby classes are themselves objects, being instances of the parent class `Class`. Ruby classes are always concrete; there are no abstract classes. However, it is theoretically possible to implement abstract classes in Ruby if you really want to do so.

The class `Object` is at the root of the hierarchy. It provides all the methods defined in the built-in Kernel module. (Technically, BasicObject is the parent of Object . It acts as a kind of "blank slate" object that does not have all the baggage of a normal object.)

To create a class that inherits from another class, define it in this way:

```
class MyClass < OtherClass
  # ...
end
```

In addition to using built-in methods, it is only natural to define your own and also to redefine and override existing ones. When you define a method with the same name as an existing one, the previous method is overridden. If a method needs to call the "parent" method that it overrides (a frequent occurrence), the keyword `super` can be used for this purpose.

Operator overloading is not strictly an OOP feature, but it is familiar to C++ programmers and certain others. Because most operators in Ruby are simply methods anyway, it should come as no surprise that these operators can be overridden or defined for user-defined classes. Overriding the meaning of an operator for an existing class may be rare, but it is common to want to define operators for new classes.

It is possible to create aliases or synonyms for methods. The syntax (used inside a class definition) is as follows:

```
alias_method :newname, :oldname
```

The number of parameters will be the same as for the old name, and it will be called in the same way. An alias creates a copy of the method, so later changes to the original method will not be reflected in aliases created beforehand.

There is also a Ruby keyword called `alias`, which is similar; unlike the method, it can alias global variables as well as methods, and its arguments are not separated by a comma.

1.3.5 Methods and Attributes

As we've seen, methods are typically used with simple class instances and variables by separating the receiver from the method with a period (`receiver.method`). In the case of method names that are punctuation, the period is omitted. Methods can take arguments:

```
Time.mktime(2014, "Aug", 24, 16, 0)
```

Because every expression returns a value, method calls may typically be chained or stacked:

```
3.succ.to_s
/(x.z).*?(x.z).*?/.match("x1z_1a3_x2z_1b3_").to_a[1..3]
3+2.succ
```

Note that there can be problems if the cumulative expression is of a type that does not support that particular method. Specifically, some methods return `nil` under certain conditions, and this usually causes any methods tacked onto that result to fail. (Of course, `nil` is an object in its own right, but it will not have all the same methods that, for example, an array would have.)

Certain methods may have blocks passed to them. This is true of all iterators, whether built in or user defined. A block is usually passed as a `do-end` block or a brace-delimited block; it is not treated like the other parameters preceding it, if any. See especially the `File.open` example:

```
my_array.each do |x|
  x.some_action
end

File.open(filename) { |f| f.some_action }
```

Methods may take a variable number of arguments:

```
receiver.method(arg1, *more_args)
```

In this case, the method called treats `more_args` as an array that it deals with as it would any other array. In fact, an asterisk in the list of formal parameters (on the last or only parameter) can likewise "collapse" a sequence of actual parameters into an array:

```
def mymethod(a, b, *c)
  print a, b
  c.each do |x| print x end
end

mymethod(1,2,3,4,5,6,7)

# a=1, b=2, c=[3,4,5,6,7]
```

Ruby also supports *named parameters,* which are called *keyword arguments* in the Python realm; the concept dates back at least as far as the Ada language developed in the 1960s and 70s. Named parameters simultaneously set default values and allow arguments to be given in any order because they are explicitly labeled:

```
def mymethod(name: "default", options: {})
  options.merge!(name: name)
  some_action_with(options)
end
```

When a named parameter has its default omitted in the method definition, it is a required named parameter:

```
def other_method(name:, age:)
  puts "Person #{name} is aged #{age}."
  # It's an error to call this method without specifying
  # values for name and age.
end
```

Ruby has the capability to define methods on a per-object basis (rather than per class). Such methods are called *singletons,* and they belong solely to that object and have no effect on its class or superclasses. As an example, this might be useful in pro-

gramming a GUI; you can define a button action for a widget by defining a singleton method for the button object.

Here is an example of defining a singleton method on a string object:

```
str = "Hello, world!"
str2 = "Goodbye!"

def str.spell
  self.split(/./).join("-")
end

str.spell      # "H-e-l-l-o-,- -w-o-r-l-d-!"
str2.spell     # error!
```

Be aware that the method is defined for the object itself, and not for the variable.

It is theoretically possible to create a prototype-based object system using singleton methods. This is a less traditional form of OOP without classes. The basic structuring mechanism is to construct a new object using an existing object as a delegate; the new object is exactly like the old object except for things that are overridden. This enables you to build prototype/delegation-based systems rather than inheritance based, and, although we do not have experience in this area, we do feel that this demonstrates the power of Ruby.

1.4 Dynamic Aspects of Ruby

Ruby is a dynamic language in the sense that objects and classes may be altered at runtime. Ruby has the capability to construct and evaluate pieces of code in the course of executing the existing statically coded program. It has a sophisticated reflection API that makes it more "self-aware"; this enables the easy creation of debuggers, profilers, and similar tools and also makes certain advanced coding techniques possible.

This is perhaps the most difficult area a programmer will encounter in learning Ruby. In this section, we briefly examine some of the implications of Ruby's dynamic nature.

1.4.1 Coding at Runtime

We have already discussed `load` and `require`, but it is important to realize that these are not built-in statements or control structures or anything of that nature; they are actual methods. Therefore, it is possible to call them with variables or expressions as

parameters or to call them conditionally. Contrast with this the `#include` directive in C or C++, which is evaluated and acted on at compile time.

Code can be constructed piecemeal and evaluated. As another contrived example, consider this `calculate` method and the code calling it:

```
def calculate(op1, operator, op2)
  string = op1.to_s + operator + op2.to_s
    # operator is assumed to be a string; make one big
    # string of it and the two operands
  eval(string)   # Evaluate and return a value
end

@alpha = 25
@beta = 12

puts calculate(2, "+", 2)        # Prints 4
puts calculate(5, "*", "@alpha") # Prints 125
puts calculate("@beta", "**", 3) # Prints 1728
```

As an even more extreme example, the following code prompts the user for a method name and a single line of code; then it actually defines the method and calls it:

```
puts "Method name: "
meth_name = gets
puts "Line of code: "
code = gets

string = %[def #{meth_name}\n #{code}\n end]   # Build a string
eval(string)                                   # Define the method
eval(meth_name)                                # Call the method
```

Frequently, programmers want to code for different platforms or circumstances and still maintain only a single code base. In such a case, a C programmer would use `#ifdef` directives, but in Ruby, definitions are executed. There is no "compile time," and everything is dynamic rather than static. So if we want to make some kind of decision like this, we can simply evaluate a flag at runtime:

```
if platform == Windows
  action1
elsif platform == Linux
```

```
  action2
else
  default_action
end
```

Of course, there is a small runtime penalty for coding in this way because the flag may be tested many times in the course of execution. But this example does essentially the same thing, enclosing the platform-dependent code in a method whose name is the same across all platforms:

```
if platform == Windows
  def my_action
    action1
  end
elsif platform == Linux
  def my_action
    action2
  end
else
  def my_action
    default_action
  end
end
```

In this way, the same result is achieved, but the flag is only evaluated once; when the user's code calls `my_action`, it will already have been defined appropriately.

1.4.2 Reflection

Languages such as Smalltalk, LISP, and Java implement (to varying degrees) the notion of a *reflective* programming language—one in which the active environment can query the objects that define it and extend or modify them at runtime.

Ruby allows reflection quite extensively but does not go as far as Smalltalk, which even represents control structures as objects. Ruby control structures and blocks are *not* objects. (A `Proc` object can be used to "objectify" a block, but control structures are never objects.)

The keyword `defined?` (with the question mark) may be used to determine whether an identifier name is in use:

```
if defined? some_var
  puts "some_var = #{some_var}"
```

```
else
  puts "The variable some_var is not known."
end
```

Similarly, the method `respond_to?` determines whether an object can respond to the specified method call (that is, whether that method is defined for that object). The `respond_to?` method is defined in class `Object`.

Ruby supports runtime-type information in a radical way. The type (or class) of an object can be determined at runtime using the method `class` (defined in `Object`). Similarly, `is_a?` tells whether an object is of a certain class (including the superclasses); `kind_of?` is an alias. Here is an example:

```
puts "abc".class   # Prints String
puts 345.class     # Prints Fixnum
rover = Dog.new

print rover.class  # Prints Dog

if rover.is_a? Dog
  puts "Of course he is."
end

if rover.kind_of? Dog
  puts "Yes, still a dog."
end

if rover.is_a? Animal
  puts "Yes, he's an animal, too."
end
```

It is possible to retrieve an exhaustive list of all the methods that can be invoked for a given object; this is done by using the `methods` method, defined in `Object`. There are also variations such as `instance_methods`, `private_instance_methods`, and so on.

Similarly, you can determine the class variables and instance variables associated with an object. By the nature of OOP, the lists of methods and variables include the entities defined not only in the object's class but also in its superclasses. The `Module` class has a method called `constants` that is used to list the constants defined within a module.