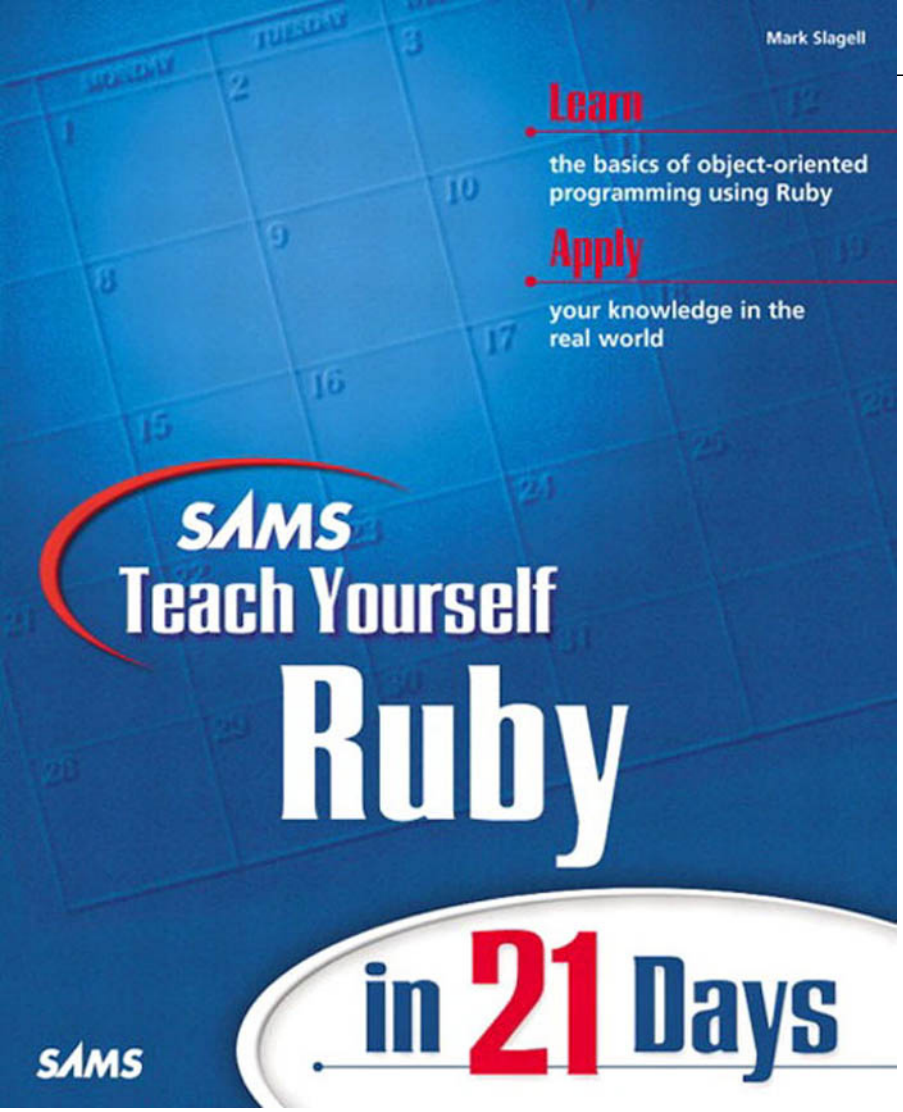
Mark Slagell
Learn
the basics of object-oriented programming using Ruby
Apply
your knowledge in the real world
SAMS
Teach Yourself
Ruby
in 21 Days
SAMS

Mark Slagell

Ruby in 21 Days

SAMS
800 East 96th St., Indianapolis, Indiana, 46240

Sams Teach Yourself Ruby in 21 Days

International Standard Book Number: 0-672-32252-8

Library of Congress Catalog Number: 2001094220

Printed in the United States of America

First Printing: March 2002

04 03 02 01 4 3 2 1

Associate Publisher
Mark Taber

Acquisitions Editor
Katie Purdum

Development Editor
Maryann Steinhart

Managing Editor
Charlotte Clapp

Project Editor
Matt Purcell

Copy Editor
Jerome Colburn
(Publication Services, Inc.)

Indexer
Jason Mortenson
(Publication Services, Inc.)

Production Editor
Theodore Young, Jr.
(Publication Services, Inc.)

Proofreader
Phil Hamer
(Publication Services, Inc.)

Technical Editors
Gene Winston
Dave Thomas
Daniel Solin

Team Coordinator
Amy Patton

Interior Designer
Gary Adair

Cover Designer
Aren Howell

Page Layout
Nina Betterly
Jennifer Faaborg
Jim Torbit
Michael Tarleton
Jessica Vonasch
(Publication Services, Inc.)

Contents at a Glance

Contents

About the Lead Author

Mark Slagell. . . .

. . . works as a system administrator and developer, using Ruby as a common language tool to solve problems in a heterogeneous Unix/Windows environment;

. . . has taught C++ programming and developed original course materials at the university level;

. . . holds a Master's degree in computer science and has participated in information security research involving the development of mobile agent tools in Java for distributed intrusion detection;

. . . got his start hand-assembling machine code for the 6502 and Z80 processors back in the dark, misty days of microcomputing (and predictably, has been fascinated ever since with the whole idea of computer *languages*);

. . . enjoys small-town life in Iowa and feels that in a world characterized by high-volume entertainment and fast-moving technology, nothing is more enjoyable than—or should ever take precedence over—a good game of softball.

About the Contributing Authors

David A. Black, Ph.D.. . .

. . . is an Associate Professor of Communication at Seton Hall University;

. . . has written for *Linux Journal*, as well as for numerous critical journals in film and media studies;

. . . is the author of the book *Law in Film*;

. . . was one of the organizers of Ruby Conference 2001;

. . . is active on the `ruby-talk` mailing list and `#ruby-lang` IRC channel.

Hal Fulton. . .

. . . has two degrees in computer science;

. . . has taught at the postsecondary level and has more than a decade of industry experience as a programmer;

. . . is a member of the ACM and the IEEE Computer Society;

. . . is also the author of *The Ruby Way* (Sams Publishing, 2001).

Dedication

To Brian Kernighan and Dennis Ritchie, whose little 1978 book
The C Programming Language *stands today as a model of computer language instruction.*

Acknowledgments

Where can I begin?

Where else but with Yukihiro “Matz” Matsumoto, creator of the breakthrough language you’re about to get hooked on.

Warm thanks go to David Alan Black and Hal Fulton for their substantial contributions to this volume, and to Dave Thomas for his careful reading, insights, and just-tactful-enough suggestions. I’d also like to thank Neil Conway for some ideas that helped shape the Ruby/Gtk chapter.

I have Akinori Musha to thank for bringing my English retranslation of Matz’s original Ruby tutorial into the mainstream, which seems to be what got me involved in the Ruby community in the first place, and Curt Clifton and Gary Leavens for their early encouragement and interest, which helped me decide to forge ahead with this project.

My friends Susan Yager and David Heddendorf helped me think about the book in a literary sense and suggested one of the better quotations that was used to open a chapter.

To my wife Amy for the patience I have not yet managed to exhaust, to older son Carter for taking on some extra tasks to help free up my time, and to younger son Kenny for asking, as only a five-year-old could, how my “story” was coming: Thank you all.

Tell Us What You Think!

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As an associate publisher for Sams Publishing, I welcome your comments. You can fax, e-mail, or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author name as well as your name and phone or fax number. I will carefully review your comments and share them with the author and editors who worked on the book.

Fax: 317-581-4770

E-mail: consumer@samspublishing.com

Mail: Mark Taber

Sams Publishing
201 West 103rd Street
Indianapolis, IN 46290 USA

Introduction

The book you hold in your hands is unusual.

You know that it is a book about computer programming, and it is about a language that you may not have heard a lot about; but that much is not, perhaps, unusual. There are many computer languages in existence, you are probably unfamiliar with most of them (isn't everybody?), and you've seen bookstore shelves overflowing with programming titles.

What is unusual is that Ruby is a very advanced programming language, yet this book claims to teach it to you while assuming almost nothing about your expertise or experience. If those two statements sound unrealistic and incompatible, don't immediately put the book down and give up. Instead, consider what it means for a programming language to *be* advanced. You might be inclined to say that an advanced language would be powerful, cryptic, and hard to learn; and would be the province of the highly experienced professional who has been doing it for years, already knows half a dozen languages inside and out, and has found them all wanting in some way.

And you'd be partly right. An advanced language would be powerful. It would also attract highly experienced professionals who found that other languages didn't do what they wanted in quite the way they wanted. But think again about the rest: A truly advanced language might not necessarily be cryptic, because being cryptic cannot be considered a virtue. Nor, by the same token, would it necessarily be hard to learn. And so its usefulness might not be limited to the most experienced programmers.

The world of programming languages has been undergoing sweeping changes in recent years, driven indirectly by advances in hardware. In the good old days, memory was scarce and expensive, and a computer didn't compute all that fast, although it always *felt* fast at the time, as I recall. The lack of cheap speed and cheap storage meant that to get a computer to do anything useful, a programmer had to work at a fairly low level of abstraction. We made our living close to the operating system and, in some cases, the hardware. We had to optimize and tweak on things to get them to run at an acceptable speed (though good compilers and other tools helped with this). Whenever we had collections of information, we thought carefully about how the items were organized in memory. Storage had to be explicitly allocated and freed, and it had to be done just right or things would go to pieces in confusing and seemingly random ways. It was tricky, and it was hard work.

Of course there's nothing wrong with hard work. But there's a good reason to get away from that particular sort of hard work as often as possible. When you write a computer

program, you are doing it to solve some particular problem, and that problem usually has nothing to do with the computer, the hardware in front of you, the operating system you're using, the language and compiler you use, and so on. The problem that you want to work on is, in and of itself, interesting. That problem is something you can probably clearly state in words (if you can't, you're in trouble from the start). And therein lies the need for advanced computer languages: When you are thinking about the *problem you wanted to solve in the first place*, you are working mentally at a different and higher level of abstraction than when you've got your hands all over the operating system. It's unnaturally hard to work at more than one level of abstraction at the same time. Not many of us are good at it. So we need high-level languages (what some people are calling "very high-level languages" or *VHLL*s) to help us concentrate on the problem at hand.

What Is Assumed About You

The idea of VHLLs is not new; the need has been present for a long time, and the term has been applied to a number of languages in the recent past. But Ruby is the first language, in my view, to achieve this combination of power, consistency, and readability. Interested people who may not have a programming background can reasonably hope to grasp it. Ruby has made a decisive crossing of the Geek Event Horizon and is the first serious language (as opposed to BASIC and similar toys) to have done so.

Accordingly, this book does not assume that you have any particular experience in programming or computer science. It assumes only that you have intelligence, curiosity, and some time; given those (and perhaps a healthy sense of humor), you can expect to pick up the new skills you are looking for.

On the other hand, neither is it assumed that you are a beginner. If you are a seasoned C, C++, Java, or Perl programmer, Ruby represents a paradigm shift, an opportunity to take a step back and reexamine how you have approached problems. You may find that there are some things you need to unlearn, but your past experience will also certainly prove useful at many points.

What's This About Levels of Abstraction?

Consider a task described as follows: Read a file named `raw`, extract all the lines containing the word `Brighton`, sort those lines in increasing numerical order by their third whitespace-separated field, and write the results to a new file named `processed`. This example will help us understand the motivation for VHLLs in more concrete terms.

Someone trying to perform that task in C would typically think at this level (at least, if trying to solve the problem in haste):

Declare two file handles. Reserve a small number of bytes as a character buffer. Is 80 enough? Yeah, that's probably reasonable. I can always increase it later if I find out differently. And I'll probably want a really big buffer to hold the whole file. But I don't know ahead of time how big the file will be . . . How big is big enough? It's safest to do that after opening the file. Take a deep breath and do it right. I'll ask the operating system for a buffer. So let's see, open the `"raw"` *file in read mode. Do a seek to move the file pointer to the end of the file and a* `tell` *to find out what the absolute position is, meaning that's the total length of the file. Remember that number. Rewind the file pointer back to the beginning. Ask the operating system for enough memory to hold the file. Check the return value in case there's a problem. Now I've got a pointer at the beginning of the big buffer, and I'll make a copy of that pointer. The original will stay put and the copy will move. While not yet at the input end-of-file, read a line into the smaller buffer and use a library function to search for a* `"Brighton"` *in the buffer. If the function returns something nonzero (or was it zero? Doesn't zero mean success? Wait, better look that up, I can never remember), it means the string was found in the buffer, so write the buffer to where the roving pointer points, then move the pointer forward the size of the input line. When at to the end of the input file, close it. Now I've got a buffer containing just the* `"Brighton"` *lines. So far, so good. But how to sort it? It will be hard to sort in place because the lines aren't all the same length. Maybe I should have made an array of pointers at the beginning of each line, then sorted the pointers . . . wait, this will work, I'll keep going through the buffer, picking out the smallest line I find each time, writing that line to the output file and then kind of blanking it out. Ready to proceed, so open the output file with name* `"processed"`*. Make a flag variable, set it to false, I mean 0. Set up a* `do-while` *loop using the flag. Copy the stationary buffer pointer to the roving one again. In the loop, move the roving pointer forward through the line until it sees whitespace twice. I should make an inner loop for that. Now use a library function to convert the text where the pointer is pointing into an integer. Wait, no, a float. Compare that to the lowest number we've seen so far. Hmm, I don't know what the lowest number seen so far is. Better declare a* `lowest` *variable, then initialize it to something impossibly high. Do that before the outer loop starts. If it's lower than the* `lowest`*, copy that value into* `lowest`*, remember where the start of the line was (Wait! I moved the pointer and don't know where the line started anymore. No problem, make a third pointer, copy the second pointer to it every time before you go looking for whitespace), and set the flag to true, I mean 1. At the end of the buffer (how will I know I'm at the end? Better back up and set up a fourth pointer after reading the input file, so we can compare position to that pointer), if the flag is 1, write the line that had lowest value to the output file, then mark the line so it won't be considered again. If the flag is 0, then we must have written all the lines, so we close the output file and we're done.*

Phew. Programming is hard, eh? And that's leaving out quite a few details and resorting to an inefficient and simple-minded sorting method. We also haven't accounted for the debugging phase, which can be counted on to be necessary because something is almost sure to go wrong somewhere. But eventually we'll get it to work—at least until we come across an input file that violates our assumptions, or until someone changes the requirements a little and we find that what we wrote won't easily adapt to the altered task and so we have to go back, Jack, and do it again. We spend our time putzing around with low-level programming details, struggling to come up with some rickety contraption to do the required job, instead of crafting a sensible, adaptable design.

I don't bring this up to disparage the C language. C programs tend to perform with good speed; there are tasks for which it is very well suited; it's possible to write in C and think much more clearly than the above example indicates. There are people who are very good at it. Either they have become so adept at the mental dance involved that they can do it efficiently—even unconsciously—or they have learned to avoid as much low-level thinking as possible and to make prudent use of their own and other people's code libraries, which is a first step toward true high-level programming.

Now let's head for more hospitable latitudes. Suppose you want to perform the same task in Ruby. Here's the likely thought process involved:

Make an array of the lines from the input file `"raw"`. *Get rid of the lines that don't contain* `"Brighton"`. *Sort the array in increasing numerical order by the third whitespace-separated field. Open an output file,* `"processed"`. *Print each line from the array to the file. Close the file.*

Well, that sounds suspiciously like the original problem description, doesn't it? It also suggests a short program. Observe:

```
the_lines = File.readlines("raw")
the_lines.delete_if { |l| l !~ "Brighton" }
the_lines.sort! { |x,y| (x.split)[2].to_f <=> (y.split)[2].to_f }
out_file = File.new("processed","w")
the_lines.each { |l| out_file.print l }
out_file.close
```

What just happened? We concentrated on what the job was about, and the high-level task description translated cleanly into a working program. No pointers, no memory allocation, no loops.[1] Thinking about the problem at hand, we shall see, is the Ruby Way.

Does it all sound too easy? Don't worry. There will be plenty of challenges ahead.

[1] *Not that we couldn't write "loopy" code in Ruby too. We can. But it isn't usually necessary.*

But when learning Ruby you'll find life to be pretty pleasant. While it doesn't solve all the big problems for you, it gives you a nice logical framework and a set of tools to handle the most tedious stuff so that you can keep working at a high level of abstraction, completing your jobs quickly, reliably, and maybe even happily.

"The Ruby Way"

We'll keep coming back to discussion about the Ruby Way, which is a phrase often thrown around in the online Ruby developer and user community. As presumptuous as it may be to suggest that there is only one Ruby Way—after all, my view of it, or way of expressing it, may be much different from another's—there is at least some shared intuition about what it is.[2]

> The Ruby Way is to avoid sweating the details when it's unnecessary.
>
> The Ruby Way is to solve most problems exactly once.
>
> The Ruby Way is to use your task specifications, whatever they are, to guide the writing of your programs.
>
> The Ruby Way is to test as you go.

Above all:

> The Ruby Way is to *use the language as a language* in the conventional, or literary, sense; something that exists to help you express yourself, not an obscure technical maze that keeps getting in your way, begging to be worked around and outsmarted.

Enjoy yourself. It's what we are all here for.

Organization of the Book

This book is organized into 21 lessons, and in keeping with the title, they are numbered by days. While it is undeniably true that everyone learns in different ways and at different speeds, it is suggested that something close to the chapter-per-day pace be observed to ensure that each lesson has a chance to sink in before the next is devoured.

At the end of most lessons there are a few exercises. They are intended to give you opportunities to test and hone your new skills, and also to goad you into contemplating some questions that will motivate you to the lessons yet to come.

---

[2] The Ruby Way *is also the title of a pretty good Ruby book you may want to pick up when you're done with this one.*