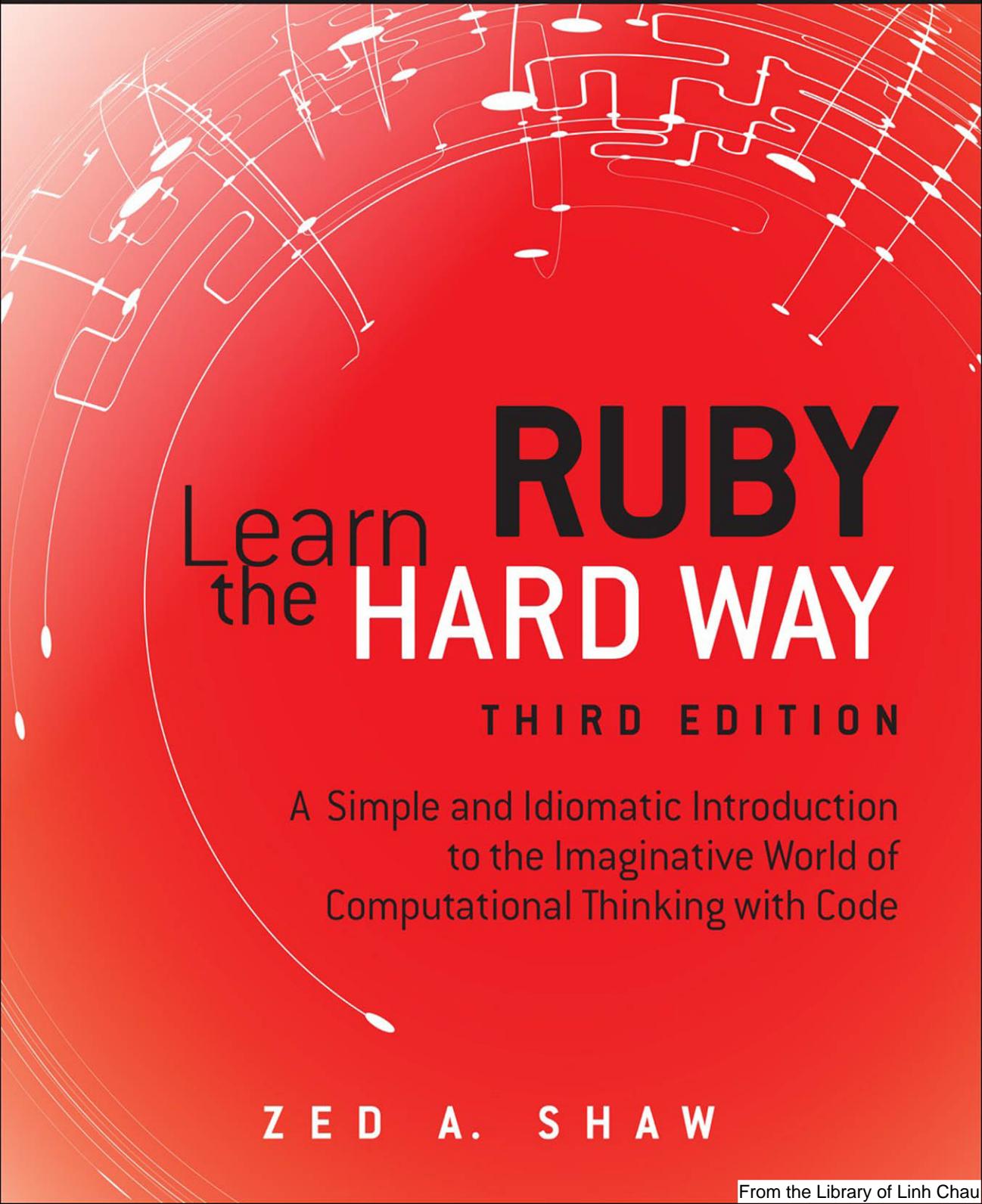


ZED SHAW'S HARD WAY SERIES

The background features a stylized globe graphic composed of white lines and dots on a red-to-orange gradient background.

Learn **RUBY** the **HARD WAY**

THIRD EDITION

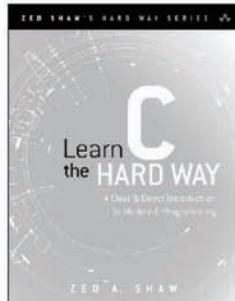
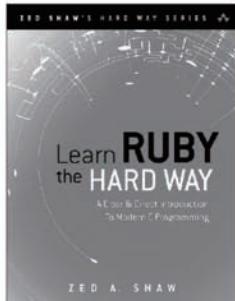
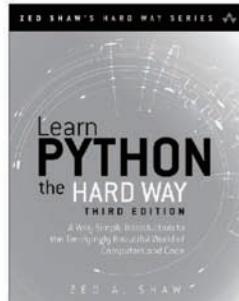
A Simple and Idiomatic Introduction
to the Imaginative World of
Computational Thinking with Code

ZED A. SHAW

LEARN RUBY THE HARD WAY

Third Edition

Zed Shaw's Hard Way Series



▲ Addison-Wesley

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

Zed Shaw's Hard Way Series emphasizes instruction and *making* things as the best way to get started in many computer science topics. Each book in the series is designed around short, understandable exercises that take you through a course of instruction that creates working software. All exercises are thoroughly tested to verify they work with real students, thus increasing your chance of success. The accompanying video walks you through the code in each exercise. Zed adds a bit of humor and inside jokes to make you laugh while you're learning.



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

informIT.com
the trusted technology learning source

▲ Addison-Wesley

Safari[®]
Books Online

LEARN RUBY THE HARD WAY

A Simple and Idiomatic Introduction
to the Imaginative World of
Computational Thinking with Code

Third Edition

Zed A. Shaw

◆ Addison-Wesley

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

From the Library of Linh Chau

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

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

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

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

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

Visit us on the Web: informat.com/hardway

Library of Congress Cataloging-in-Publication Data

Shaw, Zed, author.

Learn Ruby the hard way : a simple and idiomatic introduction to the imaginative world of computational thinking with code / Zed A. Shaw.—Third edition.

pages cm

Includes index.

ISBN 978-0-321-88499-2 (pbk. : alk. paper)

1. Ruby (Computer program language) I. Title.

QA76.73.R83S536 2014

005.1'17—dc23

2014033534

Copyright © 2015 Zed A. Shaw

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-321-88499-2

ISBN-10: 0-321-88499-X

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, December 2014

Contents

Preface	1
Acknowledgments	1
The Hard Way Is Easier	2
Reading and Writing	2
Attention to Detail	2
Spotting Differences	3
Do Not Copy-Paste	3
Using the Included Videos	3
A Note on Practice and Persistence	3
A Warning for the Smarties	4
Exercise 0 The Setup	6
Mac OS X	6
OS X: What You Should See	7
Windows	7
Windows: What You Should See	8
Linux	8
Linux: What You Should See	9
Finding Things on the Internet	10
Warnings for Beginners	10
Exercise 1 A Good First Program	12
What You Should See	14
Study Drills	16
Common Student Questions	17
Exercise 2 Comments and Pound Characters	18
What You Should See	18
Study Drills	18
Common Student Questions	19
Exercise 3 Numbers and Math	20
What You Should See	21
Study Drills	21
Common Student Questions	22

Exercise 4 Variables and Names	24
What You Should See	25
Study Drills	25
Common Student Questions	25
Exercise 5 More Variables and Printing	28
What You Should See	28
Study Drills	29
Common Student Questions	29
Exercise 6 Strings and Text	30
What You Should See	31
Study Drills	31
Common Student Question	31
Exercise 7 More Printing	32
What You Should See	32
Study Drills	33
Common Student Questions	33
Exercise 8 Printing, Printing	34
What You Should See	34
Study Drills	35
Common Student Questions	35
Exercise 9 Printing, Printing, Printing	36
What You Should See	36
Study Drills	37
Common Student Questions	37
Exercise 10 What Was That?	38
What You Should See	39
Escape Sequences	39
Study Drills	40
Common Student Questions	40
Exercise 11 Asking Questions	42
What You Should See	42
Study Drills	43
Common Student Question	43
Exercise 12 Prompting People for Numbers	44
What You Should See	44
Study Drills	44

Exercise 13	Parameters, Unpacking, Variables	46
What You Should See	46	
Study Drills	47	
Common Student Questions	47	
Exercise 14	Prompting and Passing	50
What You Should See	50	
Study Drills	51	
Common Student Questions	51	
Exercise 15	Reading Files	52
What You Should See	53	
Study Drills	53	
Common Student Questions	54	
Exercise 16	Reading and Writing Files	56
What You Should See	57	
Study Drills	57	
Common Student Questions	58	
Exercise 17	More Files	60
What You Should See	60	
Study Drills	61	
Common Student Questions	61	
Exercise 18	Names, Variables, Code, Functions	62
What You Should See	63	
Study Drills	64	
Common Student Questions	65	
Exercise 19	Functions and Variables	66
What You Should See	67	
Study Drills	67	
Common Student Questions	67	
Exercise 20	Functions and Files	70
What You Should See	71	
Study Drills	71	
Common Student Questions	71	
Exercise 21	Functions Can Return Something	74
What You Should See	75	
Study Drills	75	
Common Student Questions	76	

Exercise 22	What Do You Know So Far?	78
	What You Are Learning	78
Exercise 23	Read Some Code	80
Exercise 24	More Practice	82
	What You Should See	83
	Study Drills	83
	Common Student Questions	83
Exercise 25	Even More Practice	84
	What You Should See	85
	Study Drills	87
	Common Student Questions	87
Exercise 26	Congratulations, Take a Test!	88
	Common Student Questions	88
Exercise 27	Memorizing Logic	90
	The Truth Terms	90
	The Truth Tables	91
	Common Student Question	92
Exercise 28	Boolean Practice	94
	What You Should See	96
	Study Drills	96
	Common Student Questions	96
Exercise 29	What If	98
	What You Should See	99
	Study Drills	99
	Common Student Question	99
Exercise 30	Else and If	100
	What You Should See	101
	Study Drills	101
	Common Student Question	101
Exercise 31	Making Decisions	102
	What You Should See	103
	Study Drills	103
	Common Student Questions	103
Exercise 32	Loops and Arrays	106
	What You Should See	108

Study Drills	108
Common Student Questions	108
Exercise 33 While Loops	110
What You Should See	111
Study Drills	112
Common Student Questions	112
Exercise 34 Accessing Elements of Arrays	114
Study Drills	115
Exercise 35 Branches and Functions	116
What You Should See	118
Study Drills	118
Common Student Questions	118
Exercise 36 Designing and Debugging	120
Rules for If-Statements	120
Rules for Loops	120
Tips for Debugging	121
Homework	121
Exercise 37 Symbol Review	122
Keywords	122
Data Types	124
String Escape Sequences	124
Operators	125
Reading Code	126
Study Drills	126
Common Student Question	127
Exercise 38 Doing Things to Arrays	128
What You Should See	129
What Arrays Can Do	129
When to Use Arrays	130
Study Drills	131
Common Student Questions	131
Exercise 39 Hashes, Oh Lovely Hashes	132
A Hash Example	133
What You Should See	135
What Hashes Can Do	136

Making Your Own Hash Module	136
The Code Description	140
Three Levels of Arrays	142
What You Should See (Again)	142
When to Use Hashes or Arrays	143
Study Drills	143
Common Student Questions	144
Exercise 40 Modules, Classes, and Objects	146
Modules Are Like Hashes	146
Classes Are Like Modules	148
Objects Are Like Require	148
Getting Things from Things	150
A First Class Example	150
What You Should See	151
Study Drills	151
Common Student Question	151
Exercise 41 Learning to Speak Object Oriented	152
Word Drills	152
Phrase Drills	152
Combined Drills	153
A Reading Test	153
Practice English to Code	156
Reading More Code	156
Common Student Questions	156
Exercise 42 Is-A, Has-A, Objects, and Classes	158
How This Looks in Code	159
Study Drills	161
Common Student Questions	161
Exercise 43 Basic Object-Oriented Analysis and Design	164
The Analysis of a Simple Game Engine	165
Write or Draw about the Problem	165
Extract Key Concepts and Research Them	166
Create a Class Hierarchy and Object Map for the Concepts	167
Code the Classes and a Test to Run Them	168

Repeat and Refine	169
Top Down Versus Bottom Up	170
The Code for "Gothons from Planet Percal #25"	170
What You Should See	176
Study Drills	177
Common Student Question	177
Exercise 44 Inheritance Versus Composition	178
What Is Inheritance?	178
Implicit Inheritance	179
Override Explicitly	180
Alter Before or After	180
All Three Combined	182
Using <code>super()</code> with <code>initialize</code>	183
Composition	183
When to Use Inheritance or Composition	185
Study Drills	186
Common Student Questions	186
Exercise 45 You Make a Game	188
Evaluating Your Game	188
Function Style	189
Class Style	189
Code Style	190
Good Comments	190
Evaluate Your Game	191
Exercise 46 A Project Skeleton	192
Creating the Skeleton Project Directory	192
Final Directory Structure	193
Testing Your Setup	195
Using the Skeleton	195
Required Quiz	195
Common Student Questions	196
Exercise 47 Automated Testing	198
Writing a Test Case	198
Testing Guidelines	201

What You Should See	201
Study Drills	202
Common Student Questions	202
Exercise 48 Advanced User Input	204
Our Game Lexicon	204
Breaking Up a Sentence	205
Lexicon Tuples	205
Scanning Input	205
Exceptions and Numbers	206
A Test First Challenge	206
What You Should Test	207
Study Drills	209
Common Student Questions	210
Exercise 49 Making Sentences	212
Match and Peek	212
The Sentence Grammar	213
A Word on Exceptions	213
The Parser Code	213
Playing with the Parser	216
What You Should Test	217
Study Drills	217
Common Student Question	217
Exercise 50 Your First Website	218
Installing Sinatra	218
Make a Simple “Hello World” Project	219
What’s Happening Here?	220
Stopping and Reloading Sinatra	221
Fixing Errors	221
Create Basic Templates	222
Study Drills	223
Common Student Questions	223
Exercise 51 Getting Input from a Browser	224
How the Web Works	224
How Forms Work	226

Creating HTML Forms	227
Creating a Layout Template	228
Writing Automated Tests for Forms	229
Study Drills	230
Common Student Question	231
Exercise 52 The Start of Your Web Game	232
Refactoring the Exercise 43 Game	232
Sessions and Tracking Users	237
Creating an Engine	238
Your Final Exam	240
Next Steps	242
How to Learn Any Programming Language	243
Advice from an Old Programmer	246
Appendix Command Line Crash Course	249
Introduction: Shut Up and Shell	249
How to Use This Appendix	249
You Will Be Memorizing Things	250
The Setup	251
Do This	251
You Learned This	252
Do More	252
Paths, Folders, and Directories (pwd)	254
Do This	255
You Learned This	256
Do More	256
If You Get Lost	256
Do This	257
You Learned This	257
Make a Directory (mkdir)	257
Do This	257
You Learned This	259
Do More	259
Change Directory (cd)	260
Do This	260

You Learned This	263
Do More	264
List Directory (ls)	264
Do This	264
You Learned This	269
Do More	269
Remove Directory (rmdir)	269
Do This	270
You Learned This	272
Do More	272
Moving Around (pushd, popd)	273
Do This	273
You Learned This	275
Do More	275
Making Empty Files (Touch, New-Item)	276
Do This	276
You Learned This	277
Do More	277
Copy a File (cp)	277
Do This	277
You Learned This	280
Do More	281
Moving a File (mv)	281
Do This	281
You Learned This	283
Do More	283
View a File (less, MORE)	283
Do This	284
You Learned This	284
Do More	284
Stream a File (cat)	285
Do This	285
You Learned This	286
Do More	286

Removing a File (rm)	286
Do This	286
You Learned This	288
Do More	289
Exiting Your Terminal (exit)	289
Do This	289
You Learned This	289
Do More	289
Command Line Next Steps	290
UNIX Bash References	290
PowerShell References	290
Index	291

This page intentionally left blank

Preface

This simple book is meant to get you started in programming. The title says it's the hard way to learn to write code, but it's actually not. It's only the "hard" way because it uses a technique called *instruction*. Instruction is where I tell you to do a sequence of controlled exercises designed to build a skill through repetition. This technique works very well with beginners, who know nothing and need to acquire basic skills before they can understand more complex topics. It's used in everything from martial arts to music, to even basic math and reading skills.

This book instructs you in Ruby by slowly building and establishing skills through techniques like practice and memorization, then applying them to increasingly difficult problems. By the end of the book you will have the tools needed to begin learning more complex programming topics. I like to tell people that my book gives you your "programming black belt." What this means is that you know the basics well enough to now start learning programming.

If you work hard, take your time, and build these skills, you will learn to code.

Acknowledgments

I would like to thank Angela for helping me with the first two versions of this book. Without her, I probably wouldn't have bothered to finish it at all. She did the copyediting of the first draft, and supported me immensely while I wrote it.

I also want to thank Rob Sobers for suggesting I make a Ruby version of my Python book and doing the initial work helping me convert it to use Ruby.

I'd also like to thank Greg Newman for doing the original cover art, Brian Shumate for early website designs, and all of the people who read this book and took the time to send me feedback and corrections.

Thank you.

The Hard Way Is Easier

With the help of this book, you will do the incredibly simple things that all programmers do to learn a programming language:

1. Go through each exercise.
2. Type in each sample *exactly*.
3. Make it run.

That's it. This will be *very* difficult at first, but stick with it. If you go through this book, and do each exercise for one or two hours a night, you will have a good foundation for moving onto another book about Ruby to continue your studies. This book won't turn you into a programmer overnight, but it will get you started on the path to learning how to code.

This book's job is to teach you the three most essential skills that a beginning programmer needs to know: reading and writing, attention to detail, and spotting differences.

Reading and Writing

If you have a problem typing, you will have a problem learning to code, and especially if you have a problem typing the fairly odd characters in source code. Without this simple skill you will be unable to learn even the most basic things about how software works.

Typing the code samples and getting them to run will help you learn the names of the symbols, get familiar with typing them, and get you reading the language.

Attention to Detail

The one skill that separates bad programmers from good programmers is attention to detail. In fact, it's what separates the good from the bad in any profession. You must pay attention to the tiniest details of your work or you will miss important elements of what you create. In programming, this is how you end up with bugs and difficult-to-use systems.

By going through this book, and copying each example *exactly*, you will be training your brain to focus on the details of what you are doing, as you are doing it.

Spotting Differences

A very important skill (that most programmers develop over time) is the ability to visually notice differences between things. An experienced programmer can take two pieces of code that are slightly different and immediately start pointing out the differences. Programmers have invented tools to make this even easier, but we won't be using any of these. You first have to train your brain the hard way; then use the tools.

While you do these exercises, typing each one in, you will be making mistakes. It's inevitable; even seasoned programmers would make a few. Your job is to compare what you have written to what's required, and fix all the differences. By doing so, you will train yourself to notice mistakes, bugs, and other problems.

Do Not Copy-Paste

You must type each of these exercises in, manually. If you copy and paste, you might as well not even do them. The point of these exercises is to train your hands, your brain, and your mind in how to read, write, and see code. If you copy-paste, you are cheating yourself out of the effectiveness of the lessons.

Using the Included Videos

Learn Ruby the Hard Way has more than five hours of instructional videos to help you with the book. There is one video for each exercise where I either demonstrate the exercise, or give you tips for completing the exercise. The best way to use the videos is if you are stuck when attempting an exercise or for review after you have completed an exercise. This will slowly wean you off of using videos to learn programming and build your skills at understanding code directly. Stick with it, and slowly you won't need the videos, or any videos, to learn programming. You'll be able to just read for the information you need.

A Note on Practice and Persistence

While you are studying programming, I'm studying how to play guitar. I practice it every day for at least two hours. I play scales, chords, and arpeggios for an hour; and then I learn music theory, ear training, songs, and anything else I can. Some days I study guitar and music for eight hours because I feel like it and it's fun. To me, repetitive practice is natural and just how to learn something. I know that to get good at anything I have to practice every day, even if I suck that day (which is often) or it's difficult. Keep trying and eventually it'll be easier and fun.

Between the time that I wrote *Learn Python the Hard Way* and *Learn Ruby the Hard Way*, I discovered drawing and painting. I fell in love with making visual art at the age of 39; and I have been spending every day studying it in much the same way that I studied guitar, music, and programming. I collected books of instructional material, did what the books said, painted every day, and focused on enjoying the process of learning. I am by no means an “artist,” or even that good, but I can now say that I can draw and paint. The same method I’m teaching you in this book applied to my adventures in art. If you break the problem down into small exercises and lessons, and do them every day, you can learn to do almost anything. If you focus on slowly improving and enjoying the learning process, then you will benefit no matter how good you are at it.

As you study this book, and continue with programming, remember that anything worth doing is difficult at first. Maybe you are the kind of person who is afraid of failure, so you give up at the first sign of difficulty. Maybe you never learned self-discipline, so you can’t do anything that’s “boring.” Maybe you were told that you are “gifted,” so you never attempt anything that might make you seem stupid or not a prodigy. Maybe you are competitive and unfairly compare yourself to someone like me who’s been programming for more than 20 years.

Whatever your reason for wanting to quit, *keep at it*. Force yourself. If you run into a Study Drill you can’t do, or a lesson you just do not understand, then skip it and come back to it later. Just keep going, because with programming there’s this very odd thing that happens. At first, you will not understand anything. It’ll be weird, just like with learning any human language. You will struggle with words, and not know what symbols are what, and it’ll all be very confusing. Then one day *BANG*—your brain will snap and you will suddenly “get it.” If you keep doing the exercises and keep trying to understand them, you will get it. You might not be a master coder, but you will at least understand how programming works.

If you give up, you won’t ever reach this point. You will hit the first confusing thing (which is everything at first) and then stop. If you keep trying—keep typing it in, trying to understand it and reading about it—you will eventually get it. If you go through this whole book, and you still do not understand how to code, at least you gave it a shot. You can say you tried your best and a little more and it didn’t work out, but at least you tried. You can be proud of that.

A Warning for the Smarties

Sometimes people who already know a programming language will read this book and feel I’m insulting them. There is nothing in this book that is intended to be interpreted as condescending, insulting, or belittling. I simply know more about programming than my *intended* readers. If you think you are smarter than me, then you will feel talked down to and there’s nothing I can do about that because you are not my *intended* reader.

If you are reading this book and flipping out at every third sentence because you feel I'm insulting your intelligence, then I have three points of advice for you:

1. Stop reading my book. I didn't write it for you. I wrote it for people who don't already know everything.
2. Empty before you fill. You will have a hard time learning from someone with more knowledge if you already know everything.
3. Go learn Lisp. I hear people who know everything really like Lisp.

For everyone else who's here to learn, just read everything as if I'm smiling and I have a mischievous little twinkle in my eye.

The Setup

This exercise has no code. It is simply the exercise you complete to get your computer to run Ruby. You should follow these instructions as exactly as possible. For example, Mac OS X computers already have Ruby 2, so do not install Ruby 3 (or any Ruby).

WARNING! If you do not know how to use PowerShell on Windows, Terminal on OS X, or bash on Linux, then you need to go learn that first. You should do the exercises in Appendix A first before continuing with these exercises.

Mac OS X

Do the following tasks to complete this exercise:

1. Go to <http://www.barebones.com/products/textwrangler/> with your browser, get the TextWrangler text editor, and install it.
2. Put TextWrangler (your editor) in your dock so you can reach it easily.
3. Find your Terminal program. Search for it. You will find it.
4. Put your Terminal in your dock as well.
5. Run your Terminal program. It won't look like much.
6. In your Terminal program, run `ruby -v` to get your Ruby version.
7. If ruby is anything less than version "2.0," then your Ruby is too old. You have three choices at this point:
 - a. Upgrade your OS X to the latest version. It's free now so there's no excuse.
 - b. Go to <https://www.ruby-lang.org/en/downloads/> and try one of the installers there.
 - c. Ask a friend to help you install Ruby 2.x or greater.
8. You should be back at a prompt similar to what you had before you typed `ruby -v`. If not, find out why.
9. Learn how to make a directory in Terminal.
10. Learn how to change into a directory in Terminal.
11. Use your editor to create a file in this directory. You will make the file, "Save" or "Save As...," and pick this directory.

12. Go back to Terminal using just the keyboard to switch windows.
13. Back in Terminal, list the directory to see your newly created file.

OS X: What You Should See

Here's me doing this on my OS X computer in Terminal. Your computer might be different, but should be similar to this.

```
Last login: Sat Apr 24 00:56:54 on ttys001
~ $ ruby -v
ruby 2.1.2p95 (2014-05-08 revision 45877) [x86_64-darwin11.0]
~ $ mkdir mystuff
~ $ cd mystuff
mystuff $ ls
# ... Use TextWrangler here to edit test.txt....
mystuff $ ls
test.txt
mystuff $
```

Windows

1. Go to <http://notepad-plus-plus.org/> with your browser, get the Notepad++ text editor, and install it. You do not need to be the administrator to do this.
2. Make sure you can get to Notepad++ easily by putting it on your desktop and/or in Quick Launch. Both options are available during setup.
3. Run PowerShell from the Start menu. Search for it and you can just press Enter to run it.
4. Make a shortcut to it on your desktop and/or Quick Launch for your convenience.
5. Run your PowerShell program (which I will call Terminal later). It won't look like much.
6. In your PowerShell (Terminal) program, run ruby -v. You run things in Terminal by just typing the command name and pressing Enter.
 - a. If you run ruby and it's not there (ruby is not recognized..), install it from <http://rubyinstaller.org/> or <http://ruby.org/download>.
 - b. Make sure you install Ruby 2.0 or Ruby 2.1, but not Ruby 1.8 or 1.9.
 - c. Close PowerShell and then start it again to make sure Ruby now runs. If it doesn't, restart may be required.
 - d. You should now see ruby -v print out a version that is 2.1 or 2.0, not 1.9 or 1.8.
7. You should be back at a prompt similar to what you had before you typed ruby. If not, find out why.

8. Learn how to make a directory in PowerShell (Terminal).
9. Learn how to change into a directory in PowerShell (Terminal).
10. Use your editor to create a file in this directory. Make the file, "Save" or "Save As...," and pick this directory.
11. Go back to PowerShell (Terminal) using just the keyboard to switch windows. Look it up if you can't figure it out.
12. Back in PowerShell (Terminal), list the directory to see your newly created file.

From now on, when I say "Terminal" or "shell" I mean PowerShell and that's what you should use.

Windows: What You Should See

```
> ruby -v
ruby 2.1.2p95 (2014-05-08 revision 45877)
> mkdir mystuff
> cd mystuff

... Here you would use Notepad++ to make test.txt in mystuff ...

>
> dir
Volume in drive C is
Volume Serial Number is 085C-7E02

Directory of C:\Documents and Settings\you\mystuff

04.05.2010  23:32      <DIR>        .
04.05.2010  23:32      <DIR>        ..
04.05.2010  23:32                  6 test.txt
                           1 File(s)       6 bytes
                           2 Dir(s)  14 804 623 360 bytes free

>
```

It is still correct if you see different information than mine, but yours should be similar.

Linux

Linux is a varied operating system with a bunch of different ways to install software. I'm assuming if you are running Linux then you know how to install packages, so here are your instructions:

1. Use your Linux package manager and install the gedit text editor.
2. Make sure you can get to gedit easily by putting it in your window manager's menu.

- a. Run gedit so we can fix some stupid defaults it has.
 - b. Open Preferences and select the Editor tab.
 - c. Change Tab width: to 4.
 - d. Select (make sure a check mark is in it) Insert spaces instead of tabs.
 - e. Turn on "Automatic indentation" as well.
 - f. Open the View tab and turn on "Display line numbers."
3. Find your Terminal program. It could be called GNOME Terminal, Konsole, or xterm.
 4. Put your Terminal in your dock as well.
 5. Run your Terminal program. It won't look like much.
 6. In your Terminal program, run ruby -v. You run things in Terminal by just typing the name and pressing Enter.
 - a. If you run ruby -v and it's not there, install it. *Make sure you install Ruby 2.1 or 2.0, not Ruby 1.9 or 1.8*
 7. Type quit() and hit Enter to exit Ruby.
 8. You should be back at a prompt similar to what you had before you typed ruby -v. If not, find out why.
 9. Learn how to make a directory in Terminal.
 10. Learn how to change into a directory in Terminal.
 11. Use your editor to create a file in this directory. Typically you will make the file, Save or Save As . . . , and pick this directory.
 12. Go back to Terminal using just the keyboard to switch windows. Look it up if you can't figure it out.
 13. Back in Terminal, list the directory to see your newly created file.

Linux: What You Should See

```
$ ruby
ruby 2.1.2p95 (2014-05-08 revision 45877)
$ mkdir mystuff
$ cd mystuff
# ... Use gedit here to edit test.txt ...
$ ls
test.txt
$
```

It is still correct if you see different information than mine, but yours should be similar.

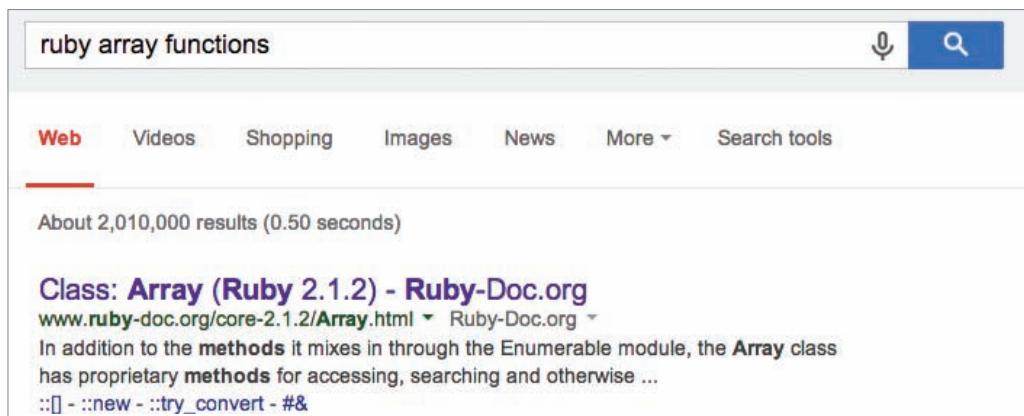
Finding Things on the Internet

A major part of this book is learning to research programming topics online. I'll tell you to "search for this on the Internet," and your job is to use a search engine to find the answer. The reason I have you search instead of just giving you the answer is because I want you to be an independent learner who does not need my book when you're done with it. If you can find the answers to your questions online then you are one step closer to not needing me, and that is my goal.

Thanks to search engines such as Google, you can easily find anything I tell you to find. If I say, "search online for the ruby array functions," then you simply do this:

1. Go to <http://google.com/>
2. Type: ruby array functions
3. Read the websites listed to find the best answer.

Here's a screenshot of me doing this search:



Warnings for Beginners

You are done with this exercise. This exercise might be hard for you depending on your familiarity with your computer. If it is difficult, take the time to read and study and get through it, because until you can do these very basic things, you will find it difficult to get much programming done.

If someone tells you to stop at a specific exercise in this book or to skip certain ones, you should ignore that person. Anyone trying to hide knowledge from you—or worse, make you get it from them instead of through your own efforts—is trying to make you depend on them for your skills. Don't listen to them and do the exercises anyway so that you learn how to educate yourself.

If a programmer tells you to use `vim` or `emacs`, just say "no." These editors are for when you are a better programmer. All you need right now is an editor that lets you put text into a file. We will use `gedit`, `TextWrangler`, or `Notepad++` (from now on called "the text editor" or "a text editor") because they are simple and the same on all computers. Professional programmers use these text editors so they are good enough for you starting out.

A programmer will eventually tell you to use Mac OS X or Linux. If the programmer likes fonts and typography, he'll tell you to get a Mac OS X computer. If he likes control and has a huge beard, he'll tell you to install Linux. Again, use whatever computer you have right now that works. All you need is an editor, a Terminal, and Ruby.

A programmer on OS X will try to make you use Homebrew or Macports, but you should *not* use these. All you need is Ruby, `TextWrangler`, and Terminal. Homebrew and Macports are for installing lots of software in a way that ruins your computer and requires more skills to fix than you or your programmer friend have. Avoid it and your computer will stay sane while you are working through this book.

A Ruby programmer will tell you to use "idiomatic" Ruby. The word "idiomatic" means something different to Ruby programmers than it does to everyone else in the world. To you and me, "idiomatic" means "a confusing phrase only a native speaker would know and one which should be avoided in well written language." To a Ruby programmer, "idiomatic" means "way.easy.most.natural.people.all.do.unless.real.programmer." The code in this book is idiomatic, but it is simple because you are a beginner. When given a choice between a Ruby idiom and clear code for a beginner, I chose clear code for a beginner. You should tell this Ruby programmer that it's a piece of cake to do idiomatic Ruby later when the whole ball of wax is front and center.

Finally, this setup helps you do three things very reliably while you work on the exercises:

1. *Write* exercises using your text editor: `gedit` on Linux, `TextWrangler` on OS X, or `Notepad++` on Windows.
2. *Run* the exercises you wrote.
3. *Fix* them when they are broken.
4. *Repeat*.

Anything else will only confuse you, so stick to the plan.

A Good First Program

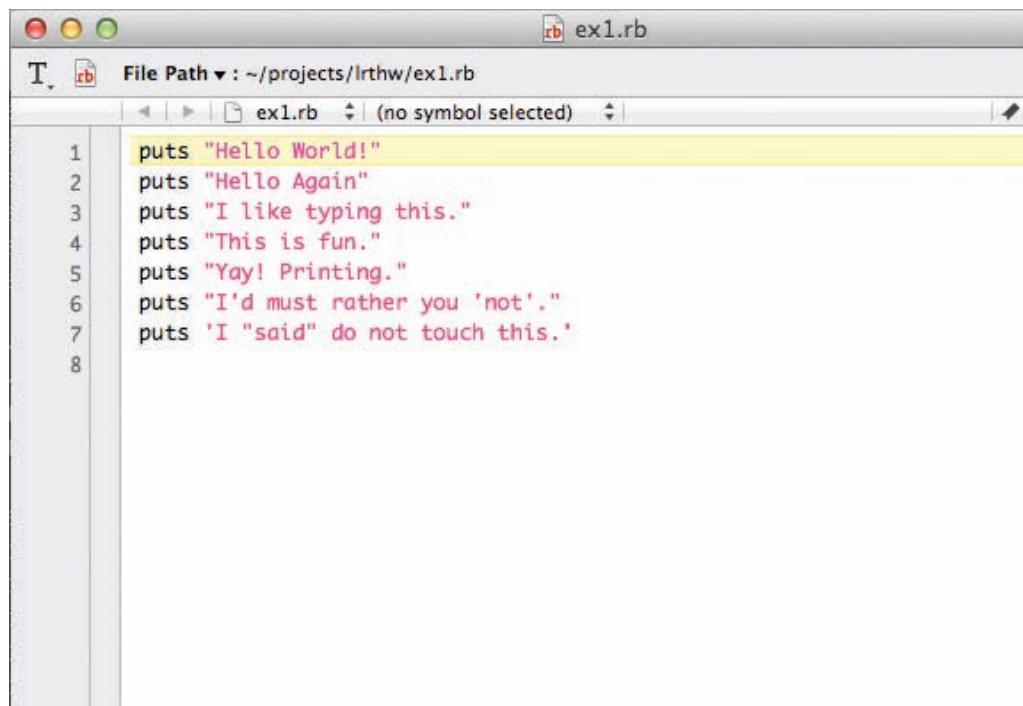
You should have spent a good amount of time in Exercise 0 learning how to install a text editor, run the text editor, run Terminal, and work with both of them. If you haven't done that, then do not go on. You will not have a good time. This is the only time I'll start an exercise with a warning that you should not skip or get ahead of yourself.

Type the following text into a single file named `ex1.rb`. Ruby works best with files ending in `.rb`.

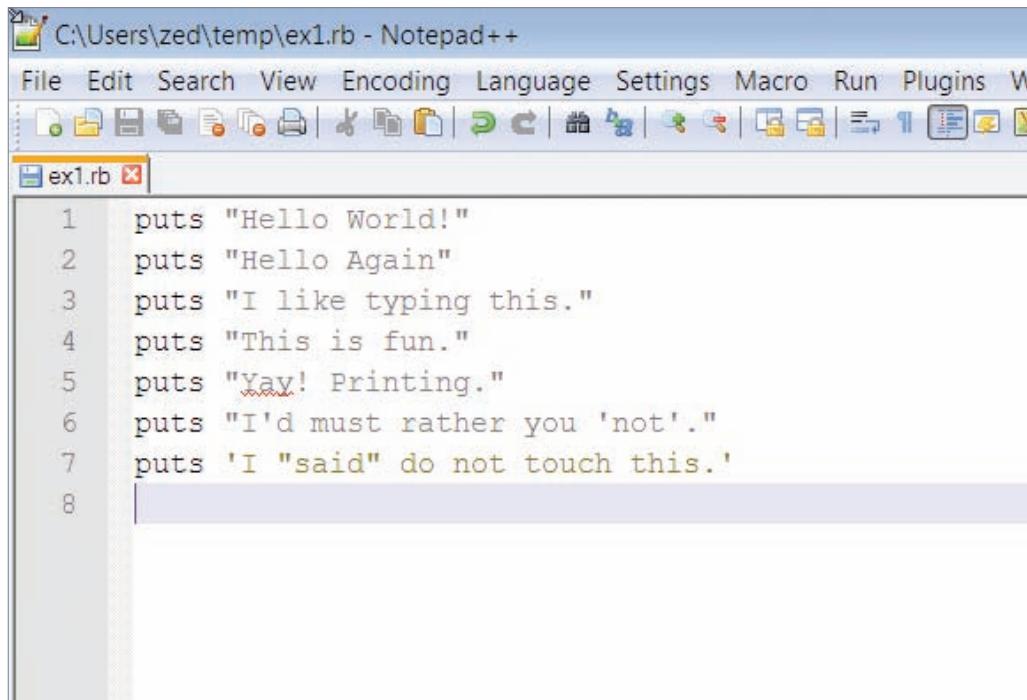
`ex1.rb`

```
1 puts "Hello World!"  
2 puts "Hello Again"  
3 puts "I like typing this."  
4 puts "This is fun."  
5 puts "Yay! Printing."  
6 puts "I'd much rather you 'not'."  
7 puts 'I "said" do not touch this.'
```

If you are on Mac OS X, then this is what your text editor might look like if you use TextWrangler:



If you are on Windows using Notepad++, then this is what it would look like:

A screenshot of the Notepad++ application window. The title bar says "C:\Users\zed\temp\ex1.rb - Notepad++". The menu bar includes File, Edit, Search, View, Encoding, Language, Settings, Macro, Run, Plugins, and Window. Below the menu is a toolbar with various icons. The main editor area shows the following Ruby code:

```
1 puts "Hello World!"
2 puts "Hello Again"
3 puts "I like typing this."
4 puts "This is fun."
5 puts "Yay! Printing."
6 puts "I'd much rather you 'not'."
7 puts 'I "said" do not touch this.'
```

The code uses color coding for syntax: "puts" is blue, strings are red, and the file path in the title bar is green.

Don't worry if your editor doesn't look exactly the same, it should be close though. When you create this file, keep in mind these points:

1. I did not type the line numbers on the left. Those are printed in the book so I can talk about specific lines by saying, "See line 5." You do not type line numbers into Ruby scripts.
2. I have the `puts` at the beginning of the line and it looks exactly the same as what I have in `ex1.rb`. Exactly means exactly, not kind of sort of the same. Every single character has to match for it to work. Color doesn't matter, only the characters you type.

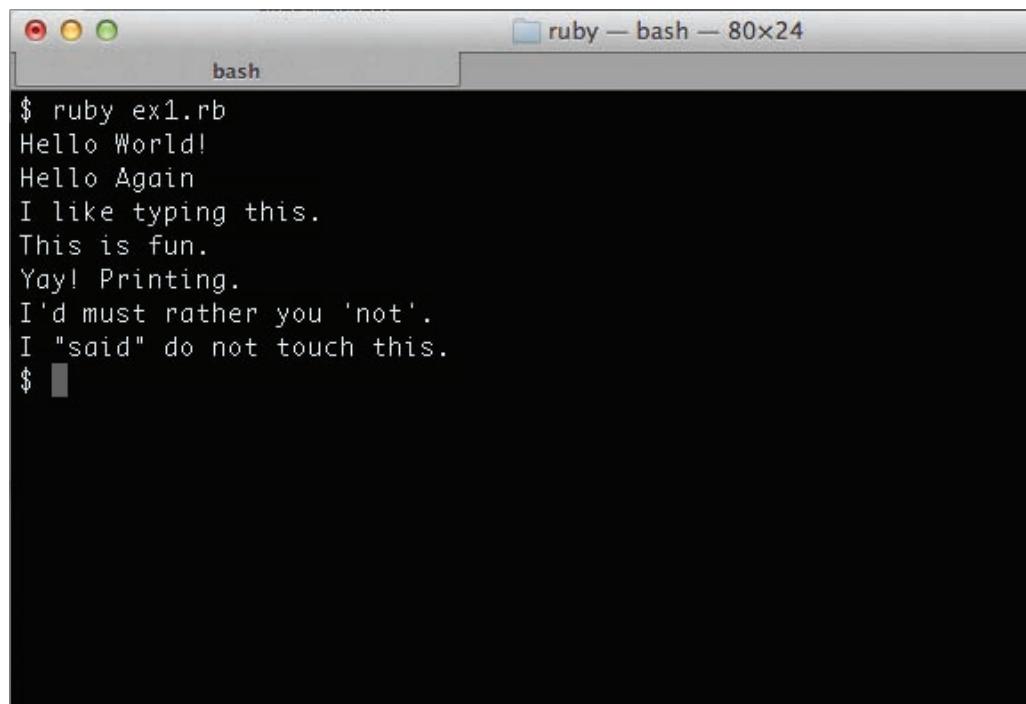
In Terminal *run* the file by typing:

```
ruby ex1.rb
```

If you did it right, then you should see the same output as in the *What You Should See* section of this exercise. If not, you have done something wrong. No, the computer is not wrong.

What You Should See

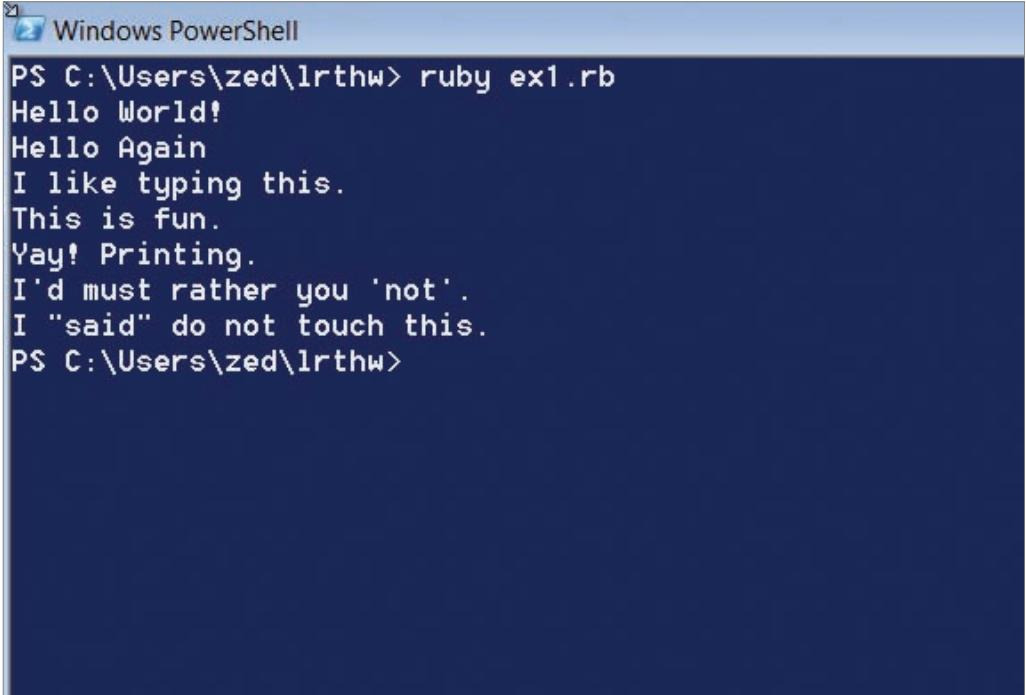
On Mac OS X in Terminal you should see this:



A screenshot of a Mac OS X Terminal window. The window title is "ruby — bash — 80x24". The terminal prompt is "\$ ". The output of the script is as follows:

```
$ ruby ex1.rb
Hello World!
Hello Again
I like typing this.
This is fun.
Yay! Printing.
I'd must rather you 'not'.
I "said" do not touch this.
$ 
```

On Windows in PowerShell you should see this:



A screenshot of a Windows PowerShell window titled "Windows PowerShell". The window shows the command "PS C:\Users\zed\lrthw> ruby ex1.rb" followed by the output of the script, which consists of several lines of text printed to the screen.

```
PS C:\Users\zed\lrthw> ruby ex1.rb
Hello World!
Hello Again
I like typing this.
This is fun.
Yay! Printing.
I'd much rather you 'not'.
I "said" do not touch this.
PS C:\Users\zed\lrthw>
```

You may see different names before the `ruby ex1.rb` command, but the important part is that you type the command and see the output is the same as mine.

If you have an error, it will look like this:

```
> ruby ex1.rb  
ex1.rb:3: syntax error, unexpected tCONSTANT, expecting $end  
puts "I like typing this."
```

It's important that you can read these error messages, because you will be making many of these mistakes. Even I make many of these mistakes. Let's look at this line by line.

1. We ran our command in Terminal to run the `ex1.rb` script.
 2. Ruby tells us that the file `ex1.rb` has an error on line 3. The type of error is "syntax error," and then some programmer jargon you can usually ignore.
 3. It prints the offending line of code for us to see.
-

WARNING! If you are from another country, and you get errors about ASCII encodings, then put this at the top of your Ruby scripts:

```
# -*- coding: utf-8 -*-
```

It will fix them so that you can use Unicode UTF-8 in your scripts without a problem.

Study Drills

The Study Drills contain things you should *try* to do. If you can't, skip it and come back later.

For this exercise, try these things:

1. Make your script print another line.
2. Make your script print only one of the lines.
3. Put a `#` (octothorpe) character at the beginning of a line. What did it do? Try to find out what this character does.

From now on, I won't explain how each exercise works unless an exercise is different.

NOTE: An "octothorpe" is also called a "pound," "hash," "mesh," or any number of names. Pick the one that makes you chill out.

Common Student Questions

These are *actual* questions that real students have asked when doing this exercise.

How do you get colors in your editor?

Save your file first as a .rb file, such as ex1.rb. Then you'll have color when you type.

I get ruby: No such file or directory – ex1.rb (LoadError).

You need to be in the same directory as the file you created. Make sure you use the cd command to go there first. For example, if you saved your file in lrthw/ex1.rb, then you would type cd lrthw/ before trying to run ruby ex1.rb. If you don't know what any of that means, then go through *Appendix A*.

Comments and Pound Characters

Comments are very important in your programs. They are used to tell you what something does in English, and they are used to disable parts of your program if you need to remove them temporarily. Here's how you use comments in Ruby:

ex2.rb

```

1 # A comment, this is so you can read your program later.
2 # Anything after the # is ignored by ruby.
3
4 puts "I could have code like this." # and the comment after is ignored
5
6 # You can also use a comment to "disable" or comment out a piece of code:
7 # puts "This won't run."
8
9 puts "This will run."

```

From now on, I'm going to write code like this. It is important for you to understand that everything does not have to be literal. Your screen and program may visually look different, but what's important is the text you type into the file you're writing in your text editor. In fact, I could work with any text editor and the results would be the same.

What You Should See

Exercise 2 Session

```

$ ruby ex2.rb
I could have code like this.
This will run.

```

Again, I'm not going to show you screenshots of all the Terminals possible. You should understand that the above is not a literal translation of what your output should look like visually, but the text between the first `$ ruby ...` and last `$` lines will be what you focus on.

Study Drills

1. Find out if you were right about what the `#` character does and make sure you know what it's called (octothorpe or pound character).
2. Take your `ex2.rb` file and review each line going backward. Start at the last line, and check each word in reverse against what you should have typed.

3. Did you find more mistakes? Fix them.
4. Read what you typed out loud, including saying each character by its name. Did you find more mistakes? Fix them.

Common Student Questions

Are you sure # is called the pound character?

I call it the octothorpe because that is the only name that no country uses, and that works in every country. Every country thinks its name for this one character is both the most important way to do it and the only way it's done. To me, this is simply arrogance and, really, y'all should just chill out and focus on more important things like learning to code.

If # is for comments, then how come # -*- coding: utf-8 -*- works?

Ruby still ignores that as code, but it's used as a kind of "hack" or workaround for problems with setting and detecting the format of a file. You will also find a similar kind of comment for editor settings.

Why does the # in puts "Hi # there." not get ignored?

The # in that code is inside a string, so it will be put into the string until the ending " character is hit. These pound characters are just considered characters and aren't considered comments.

How do I comment out multiple lines?

Put a # in front of each one.

I can't figure out how to type a # character on my country's keyboard.

Some countries use the Alt key and combinations of other keys to print characters foreign to their language. You'll have to look online in a search engine to see how to type it.

Why do I have to read code backward?

It's a trick to make your brain not attach meaning to each part of the code, and doing that makes you process each piece exactly. This catches errors and is a handy error-checking technique.

Numbers and Math

Every programming language has some kind of way of doing numbers and math. Do not worry: programmers lie frequently about being math geniuses when they really aren't. If they were math geniuses, they would be doing math, not writing buggy web frameworks so they can drive race cars.

This exercise has lots of math symbols. Let's name them right away so you know what they are called. As you type each one in, say the name. When saying them feels boring you can stop saying them. Here are the names:

- + plus
- - minus
- / slash
- * asterisk
- % percent
- < less-than
- > greater-than
- <= less-than-equal
- >= greater-than-equal

Notice how the operations are missing? After you type in the code for this exercise, go back and figure out what each of these symbols does and complete the table. For example, + does addition.

ex3.rb

```

1 puts "I will now count my chickens:"
2
3 puts "Hens #{25 + 30 / 6}"
4 puts "Roosters #{100 - 25 * 3 % 4}"
5
6 puts "Now I will count the eggs:"
7
8 puts 3 + 2 + 1 - 5 + 4 % 2 - 1 / 4 + 6
9
10 puts "Is it true that 3 + 2 < 5 - 7?"
11
12 puts 3 + 2 < 5 - 7
13

```

```
14 puts "What is 3 + 2? #{3 + 2}"
15 puts "What is 5 - 7? #{5 - 7}"
16
17 puts "Oh, that's why it's false."
18
19 puts "How about some more."
20
21 puts "Is it greater? #{5 > -2}"
22 puts "Is it greater or equal? #{5 >= -2}"
23 puts "Is it less or equal? #{5 <= -2}"
```

What You Should See

Exercise 3 Session

```
$ ruby ex3.rb
I will now count my chickens:
Hens 30
Roosters 97
Now I will count the eggs:
7
Is it true that 3 + 2 < 5 - 7?
false
What is 3 + 2? 5
What is 5 - 7? -2
Oh, that's why it's false.
How about some more.
Is it greater? true
Is it greater or equal? true
Is it less or equal? false
```

Study Drills

1. Above each line, use the # to write a comment to yourself explaining what the line does.
2. Remember in Exercise 0 when you started Ruby? Start Ruby this way again and using the math operators, use Ruby as a calculator.
3. Find something you need to calculate and write a new .rb file that does it.
4. Notice the math seems "wrong"? There are no fractions, only whole numbers. You need to use a "floating point" number, which is a number with a decimal point, as in 10.5, or 0.89, or even 3.0.
5. Rewrite ex3.rb to use floating point numbers so it's more accurate. 20.0 is a floating point number.

Common Student Questions

Why is the % character a "modulus" and not a "percent"?

Mostly that's just how the designers chose to use that symbol. In normal writing, you are correct to read it as a "percent." In programming, this calculation is typically done with simple division and the / operator. The % modulus is a different operation that just happens to use the % symbol.

How does % work?

Another way to say it is, "X divided by Y with J remaining." For example, "100 divided by 16 with 4 remaining." The result of % is the J part, or the remaining part.

What is the order of operations?

In the United States we use an acronym called PEMDAS, which stands for Parentheses Exponents Multiplication Division Addition Subtraction. That's the order Ruby follows as well.

This page intentionally left blank

Variables and Names

Now you can print things with puts and you can do math. The next step is to learn about variables. In programming a variable is nothing more than a name for something, similar to how my name "Zed" is a name for, "The human who wrote this book." Programmers use these variable names to make their code read more like English, and because they have lousy memories. If they didn't use good names for things in their software, they'd get lost when they tried to read their code again.

If you get stuck with this exercise, remember the tricks you have been taught so far of finding differences and focusing on details:

1. Write a comment above each line explaining to yourself what it does in English.
2. Read your .rb file backward.
3. Read your .rb file out loud, saying even the characters.

ex4.rb

```
1  cars = 100
2  space_in_a_car = 4.0
3  drivers = 30
4  passengers = 90
5  cars_not_driven = cars - drivers
6  cars_driven = drivers
7  carpool_capacity = cars_driven * space_in_a_car
8  average_passengers_per_car = passengers / cars_driven
9
10
11 puts "There are #{cars} cars available."
12 puts "There are only #{drivers} drivers available."
13 puts "There will be #{cars_not_driven} empty cars today."
14 puts "We can transport #{carpool_capacity} people today."
15 puts "We have #{passengers} to carpool today."
16 puts "We need to put about #{average_passengers_per_car} in each car."
```

NOTE: The _ in space_in_a_car is called an underscore character. Find out how to type it if you do not already know. We use this character a lot to put an imaginary space between words in variable names.

What You Should See

Exercise 4 Session

```
$ ruby ex4.rb
There are 100 cars available.
There are only 30 drivers available.
There will be 70 empty cars today.
We can transport 120.0 people today.
We have 90 to carpool today.
We need to put about 3 in each car.
```

Study Drills

When I wrote this program the first time I had a mistake, and Ruby told me about it like this:

```
There are 100 cars available.
There are only 30 drivers available.
There will be 70 empty cars today.
ex4.rb:14: undefined local variable or method 'carpool_capacity' for
main:Object (NameError)
```

Explain this error in your own words. Make sure you use line numbers and explain why.

Here are more study drills:

1. I used 4.0 for space_in_a_car, but is that necessary? What happens if it's just 4?
2. Remember that 4.0 is a floating point number. It's just a number with a decimal point, and you need 4.0 instead of just 4 so that it is a floating point.
3. Write comments above each of the variable assignments.
4. Make sure you know what = is called (equals), and that it's making names for things.
5. Remember that _ is an underscore character.
6. Try running ruby from Terminal as a calculator like you did before and use variable names to do your calculations. Popular variable names are also i, x, and j.

Common Student Questions

What is the difference between = (single-equal) and == (double-equal)?

The = (single-equal) assigns the value on the right to a variable on the left. The == (double-equal) tests if two things have the same value. You'll learn about this in Exercise 27.

Can we write x=100 instead of x = 100?

You can, but it's bad form. You should add space around operators like this so that it's easier to read.

What do you mean by "read the file backward"?

Very simple. Imagine you have a file with 16 lines of code in it. Start at line 16, and compare it to my file at line 16. Then do it again for 15, and so on until you've read the whole file backward.

Why did you use 4.0 for space_in_a_car?

It is mostly so you can then find out what a floating point number is and ask this question. See the Study Drills.

This page intentionally left blank

More Variables and Printing

Now we'll do even more typing of variables and printing them out. Every time you put " (double-quotes) around a piece of text you have been making a *string*. A string is how you make something that your program might give to a human. You print strings, save strings to files, send strings to web servers, and many other things.

Strings are really handy, so in this exercise you will learn how to make strings that have variables embedded in them. You embed variables inside a string by using a special #{} sequence and then put the variable you want inside the {} characters. This tells Ruby, "Hey, this string needs to be formatted. Put these variables in there."

As usual, just type this in (even if you do not understand it) and make it exactly the same.

ex5.rb

```

1  my_name = 'Zed A. Shaw'
2  my_age = 35 # not a lie in 2009
3  my_height = 74 # inches
4  my_weight = 180 # lbs
5  my_eyes = 'Blue'
6  my_teeth = 'White'
7  my_hair = 'Brown'
8
9  puts "Let's talk about #{my_name}."
10 puts "He's #{my_height} inches tall."
11 puts "He's #{my_weight} pounds heavy."
12 puts "Actually that's not too heavy."
13 puts "He's got #{my_eyes} eyes and #{my_hair} hair."
14 puts "His teeth are usually #{my_teeth} depending on the coffee."
15
16 # this line is tricky, try to get it exactly right
17 puts "If I add #{my_age}, #{my_height}, and #{my_weight}"
18     I get #{my_age + my_height + my_weight}."
```

What You Should See

Exercise 5 Session

```
$ ruby ex5.rb
Let's talk about Zed A. Shaw.
He's 74 inches tall.
He's 180 pounds heavy.
```

Actually that's not too heavy.
He's got Blue eyes and Brown hair.
His teeth are usually White depending on the coffee.
If I add 35, 74, and 180 I get 289.

Study Drills

1. Change all the variables so there is no `my_` in front of each one. Make sure you change the name everywhere, not just where you used `=` to set them.
2. Try to write some variables that convert the inches and pounds to centimeters and kilograms. Do not just type in the measurements. Work out the math in Ruby.

Common Student Questions

Can I make a variable like this: `1 = 'Zed Shaw'`?

No, `1` is not a valid variable name. Variable names need to start with a character, so `a1` would work, but `1` will not.

Why does this not make sense to me?

Try making the numbers in this script your measurements. It's weird, but talking about yourself will make it seem more real. Also, you're just starting out so it won't make too much sense. Keep going and more exercises will explain it more.

Strings and Text

While you have been writing strings, you still do not know what they do. In this exercise we create a bunch of variables with complex strings so you can see what they are for. First, we need an explanation of strings.

A string is usually a bit of text you want to display to someone, or "export" out of the program you are writing. Ruby knows you want something to be a string when you put either " (double-quotes) or ' (single-quotes) around the text. You saw this many times with your use of puts, when you put the text you want to go into the string inside " after the puts to print the string.

Strings can contain any number of variables that are in your Ruby script. Remember that a variable is any line of code where you set a name = (equal) to a value. In the code for this exercise, types_of_people = 10 creates a variable named types_of_people and sets it = (equal) to 10. You can put that in any string with #{types_of_people}.

We will now type in a whole bunch of strings, variables, and formats, and print them. We'll also practice using short abbreviated variable names. Programmers love saving time at your expense by using annoyingly short and cryptic variable names, so let's get you started reading and writing them early on.

ex6.rb

```
1  types_of_people = 10
2  x = "There are #{types_of_people} types of people."
3  binary = "binary"
4  do_not = "don't"
5  y = "Those who know #{binary} and those who #{do_not}."
6
7  puts x
8  puts y
9
10 puts "I said: #{x}."
11 puts "I also said: '#{y}'."
12
13 hilarious = false
14 joke_evaluation = "Isn't that joke so funny?! #{hilarious}"
15
16 puts joke_evaluation
17
18 w = "This is the left side of..."
19 e = "a string with a right side."
20
21 puts w + e
```

What You Should See

Exercise 6 Session

```
$ ruby ex6.rb
There are 10 types of people.
Those who know binary and those who don't.
I said: There are 10 types of people..
I also said: 'Those who know binary and those who don't.'.
Isn't that joke so funny?! false
This is the left side of...a string with a right side.
```

Study Drills

1. Go through this program and write a comment above each line explaining it.
2. Find all the places where a string is put inside a string. There are four places.
3. Are you sure there are only four places? How do you know? Maybe I like lying.
4. Explain why adding the two strings w and e with + makes a longer string.
5. What happens when you change the strings to use ' (single-quote) instead of " (double-quote)? Do they still work? Try to guess why.

Common Student Question

If you thought the joke was funny, could you write `hilarious = true`?

Yes, and you'll learn more about these Boolean values in Exercise 27.

More Printing

Now we are going to do a bunch of exercises where you just type code in and make it run. I won't be explaining this exercise because it is more of the same. The purpose is to build up your chops. See you in a few exercises, and *do not skip! Do not paste!*

ex7.rb

```

1  puts "Mary had a little lamb."
2  puts "Its fleece was white as #{'snow'}."
3  puts "And everywhere that Mary went."
4  puts "." * 10 # what'd that do?
5
6  end1 = "C"
7  end2 = "h"
8  end3 = "e"
9  end4 = "e"
10 end5 = "s"
11 end6 = "e"
12 end7 = "B"
13 end8 = "u"
14 end9 = "r"
15 end10 = "g"
16 end11 = "e"
17 end12 = "r"
18
19 # watch that print vs. puts on this line what's it do?
20 print end1 + end2 + end3 + end4 + end5 + end6
21 puts end7 + end8 + end9 + end10 + end11 + end12

```

What You Should See

Exercise 7 Session

```

$ ruby ex7.rb
Mary had a little lamb.
Its fleece was white as snow.
And everywhere that Mary went.
.....
CheeseBurger

```

Study Drills

For these next few exercises, you will have the exact same Study Drills.

1. Go back through and write a comment on what each line does.
2. Read each one backward or out loud to find your errors.
3. From now on, when you make a mistake, write down on a piece of paper what kind of mistake you made.
4. When you go to the next exercise, look at the mistakes you have made and try not to make them in this new one.
5. Remember that everyone makes mistakes. Programmers are like magicians who fool everyone into thinking they are perfect and never wrong, but it's all an act. They make mistakes all the time.

Common Student Questions

Why are you using the variable named 'snow'?

That's actually not a variable: it is just a string with the word snow in it. A variable wouldn't have the single-quotes around it.

Is it normal to write an English comment for every line of code like you say to do in Study Drill 1?

No, you write comments only to explain difficult code or why you did something. Why is usually much more important, and then you try to write the code so that it explains how something is being done on its own. However, sometimes you have to write such nasty code to solve a problem that it does need a comment on every line. In this case it's strictly for you to practice translating code to English.

Can I use single-quotes or double-quotes to make a string, or do they do different things?

In Ruby, the " (double-quote) tells Ruby to replace variables it finds with #{}, but the ' (single-quote) tells Ruby to leave the string alone and ignore any variables inside it.

Printing, Printing

I will now show you how to create a format string, but rather than using variables, we'll use values by their names. Some of this is a bit more advanced, but don't worry you'll learn what all of these mean later. Just type this in, make it work, and write a comment above each line translating it to English.

ex8.rb

```

1  formatter = "%{first} %{second} %{third} %{fourth}"
2
3  puts formatter % {first: 1, second: 2, third: 3, fourth: 4}
4  puts formatter % {first: "one", second: "two", third: "three",
5                  fourth: "four"}
6  puts formatter % {first: true, second: false, third: true, fourth: false}
7  puts formatter % {first: formatter, second: formatter, third: formatter,
8                  fourth: formatter}
9
10 puts formatter % {
11    first: "I had this thing.",
12    second: "That you could type up right.",
13    third: "But it didn't sing.",
14    fourth: "So I said goodnight."
15 }
```

What You Should See

Exercise 8 Session

```

$ ruby ex8.rb
1 2 3 4
one two three four
true false true false
%{first} %{second} %{third} %{fourth} %{first} %{second} %{third} %{fourth}
%{first} %{second} %{third} %{fourth} %{first} %{second} %{third} %{fourth}
I had this thing. That you could type up right. But it didn't sing.
So I said goodnight.
```

Study this carefully and try to see how I put the formatter inside the formatter.

Study Drills

1. Check your work, write down your mistakes, and try not to make the same mistakes on the next exercise.
2. Notice that the last line of output uses both single-quotes and double-quotes for individual pieces. Why do you think that is?

Common Student Questions

Should I use %{} or #{} for formatting?

You will almost always use #{} to format your strings, but there are times when you want to apply the same format to multiple values. That's when %{} comes in handy.

Why do I have to put quotes around "one" but not around true or false?

Ruby recognizes true and false as keywords representing the concept of true and false. If you put quotes around them, then they are turned into strings and won't work. You'll learn more about how these work in Exercise 27.

Printing, Printing, Printing

By now you should realize the pattern for this book is to use more than one exercise to teach you something new. I start with code that you might not understand, then more exercises explain the concept. If you don't understand something now, you will later as you complete more exercises. Write down what you don't understand, and keep going.

ex9.rb

```

1  # Here's some new strange stuff, remember type it exactly.
2
3  days = "Mon Tue Wed Thu Fri Sat Sun"
4  months = "Jan\nFeb\nMar\nApr\nMay\nJun\nJul\nAug"
5
6  puts "Here are the days: #{days}"
7  puts "Here are the months: #{months}"
8
9  puts """
10 There's something going on here.
11 With the three double-quotes.
12 We'll be able to type as much as we like.
13 Even 4 lines if we want, or 5, or 6.
14 """

```

What You Should See

Exercise 9 Session

```

$ ruby ex9.rb
Here are the days: Mon Tue Wed Thu Fri Sat Sun
Here are the months: Jan
Feb
Mar
Apr
May
Jun
Jul
Aug

```

```

There's something going on here.
With the three double-quotes.
We'll be able to type as much as we like.
Even 4 lines if we want, or 5, or 6.

```

Study Drills

1. Check your work, write down your mistakes, and try not to make them on the next exercise.

Common Student Questions

Why do I get an error when I put spaces between the three double-quotes?

You have to type them like """ and not " ", meaning with *no* spaces between each one.

Is it bad that my errors are always spelling mistakes?

Most programming errors in the beginning (and even later) are simple spelling mistakes, typos, or getting simple things out of order.

What Was That?

In Exercise 9, I threw you some new stuff, just to keep you on your toes. I showed you two ways to make a string that goes across multiple lines. In the first way, I put the characters `\n` (backslash n) between the names of the months. These two characters put a new `line` character into the string at that point.

This `\` (backslash) character encodes difficult-to-type characters into a string. There are various "escape sequences" available for different characters you might want to use. We'll try a few of these sequences so you can see what I mean.

An important escape sequence is to escape a single-quote `'` or double-quote `"`. Imagine you have a string that uses double-quotes and you want to put a double-quote inside the string. If you write `"I "understand" joe."` then Ruby will get confused because it will think the `"` around `"understand"` actually *ends* the string. You need a way to tell Ruby that the `"` inside the string isn't a *real* double-quote.

To solve this problem you *escape* double-quotes and single-quotes so Ruby knows to include in the string. Here's an example:

```
"I am 6'2\" tall." # escape double-quote inside string
'I am 6\'2" tall.' # escape single-quote inside string
```

The second way is by using triple-quotes, which is just `"""` and works like a string, but you also can put as many lines of text as you want until you type `"""` again. We'll also play with these.

ex10.rb

```

1 tabby_cat = "\tI'm tabbed in."
2 persian_cat = "I'm split\non a line."
3 backslash_cat = "I'm \\ a \\ cat."
4
5 fat_cat = """
6 I'll do a list:
7 \t* Cat food
8 \t* Fishies
9 \t* Catnip\n\t* Grass
10 """
11
12 puts tabby_cat
13 puts persian_cat
14 puts backslash_cat
15 puts fat_cat
```

What You Should See

Look for the tab characters that you made. In this exercise the spacing is important to get right.

Exercise 10 Session

```
$ ruby ex10.rb
      I'm tabbed in.
I'm split
on a line.
I'm a cat.

I'll do a list:
  * Cat food
  * Fishies
  * Catnip
  * Grass
```

Escape Sequences

These are all of the escape sequences Ruby supports. You may not use many of these, but you should memorize their format and what they do anyway. Try them out in some strings to see if you can make them work.

Escape	What it does.
\\	Backslash ()
\'	Single-quote ('')
\"	Double-quote (")
\a	ASCII bell (BEL)
\b	ASCII backspace (BS)
\f	ASCII formfeed (FF)
\n	ASCII linefeed (LF)
\r ASCII	Carriage Return (CR)
\t ASCII	Horizontal Tab (TAB)
\uxxxx	Character with 16-bit hex value xxxx (Unicode only)
\Uxxxxxxxxx	Character with 32-bit hex valuexxxxxxxx (Unicode only)
\v	ASCII vertical tab (VT)
\ooo	Character with octal value ooo
\xhh	Character with hex value hh

Study Drills

1. Memorize all the escape sequences by putting them on flash cards.
2. Use ''' (triple-single-quote) instead. Can you see why you might use that instead of """"?
3. Combine escape sequences and format strings to create a more complex format.

Common Student Questions

What if I wanted to start the months on a new line?

You simply start the string with \n like this:

```
"\nJan\nFeb\nMar\nApr\nMay\nJun\nJul\nAug"
```

I still haven't completely figured out the last exercise. Should I continue?

Yes, keep going. Instead of stopping, take notes listing things you don't understand for each exercise. Periodically go through your notes and see if you can figure these things out after you've completed more exercises. Sometimes, though, you may need to go back a few exercises and do them again.

What makes \\ special compared to the other ones?

It's simply the way you would write out one backslash (\) character. Think about why you would need this.

When I write // or /n, it doesn't work.

That's because you are using a forward-slash / and not a backslash \. They are different characters that do very different things.

I don't get Study Drill 3. What do you mean by "combine" escape sequences and formats?

One concept I need you to understand is that each of these exercises can be combined to solve problems. Take what you know about format strings and write some new code that uses format strings *and* the escape sequences from this exercise. What's better, ''' or """"? Use ''' when you need a multi-line string that contains #{} formatting, but you don't want them to be processed yet or at all. Use """" for all other multi-line strings.

This page intentionally left blank

Asking Questions

Now it is time to pick up the pace. You are doing a lot of printing to get you familiar with typing simple things, but those simple things are fairly boring. What we want to do now is get data into your programs. This is a little tricky because you have to learn to do two things that may not make sense right away—but trust me and do it anyway. It will make sense in a few exercises.

Most of what software does is the following:

1. Take some kind of input from a person.
2. Change it.
3. Print out something to show how it changed.

So far you have been printing strings, but you haven't been able to get any input from a person. You may not even know what "input" means, but type this code in anyway and make it exactly the same. In the next exercise we'll do more to explain input.

ex11.rb

```
1 print "How old are you? "
2 age = gets.chomp
3 print "How tall are you? "
4 height = gets.chomp
5 print "How much do you weigh? "
6 weight = gets.chomp
7
8 puts "So, you're #{age} old, #{height} tall and #{weight} heavy."
```

NOTE: I use `print` instead of `puts` to print the string without a `\n` (newline) printed, and the prompt stops right where the user should enter the answer.

What You Should See

Exercise 11 Session

```
$ ruby ex11.rb
How old are you? 39
How tall are you? 6'2"
How much do you weigh? 180lbs
So, you're 39 old, 6'2" tall and 180lbs heavy.
```

Study Drills

1. Go online and find out what Ruby's `gets.chomp` does.
2. Can you find other ways to use it? Try some of the samples you find.
3. Write another "form" like this to ask some other questions.

Common Student Question

How do I get a number from someone so I can do math?

That's a little advanced, but try `gets.chomp.to_i` which says, "Get a string from the user, chomp off the \n, and convert it to an integer."

Prompting People for Numbers

You may have noticed in Exercise 11 that you were getting only strings from the user. How do you get numbers? Type this code in to find out how.

ex12.rb

```
1  print "Give me a number: "
2  number = gets.chomp.to_i
3
4  bigger = number * 100
5  puts "A bigger number is #{bigger}."
6
7  print "Give me another number: "
8  another = gets.chomp
9  number = another.to_i
10
11 smaller = number / 100
12 puts "A smaller number is #{smaller}."
```

What You Should See

Exercise 12 Session

```
$ ruby ex12.rb
Give me a number: 10
A bigger number is 1000.
Give me another number: 200
A smaller number is 2.
```

See how I can add a `.to_i` to the `gets.chomp`, and it will convert to an integer? I can also save what `gets.chomp` returns, and call `.to_i` on that, as I did with `number = another.to_i`.

Study Drills

1. Try out the `.to_f` operation. What does `.to_f` do?
2. To play with `.to_f` more, make a small script that asks for some money and gives back 10% of it. If I give your script 103.4 (dollars), your script gives me back 10.34 in change.

This page intentionally left blank

Parameters, Unpacking, Variables

In this exercise we will cover one more input method you can use to pass variables to a script (script being another name for your .rb files). You know how you type `ruby ex13.rb` to run the `ex13.rb` file? Well, the `ex13.rb` part of the command is called an “argument.” What we’ll do now is write a script that also accepts arguments.

Type this program, and I’ll explain it in detail:

`ex13.rb`

```

1 first, second, third = ARGV
2
3 puts "Your first variable is: #{first}"
4 puts "Your second variable is: #{second}"
5 puts "Your third variable is: #{third}"

```

`ARGV` is the “argument variable,” a very standard name in programming, that you will find used in many other languages. This variable *holds* the arguments you pass to your Ruby script when you run it. In the exercises you will get to play with this more and see what happens.

Line 1 “unpacks” `ARGV` so that, rather than holding all the arguments, it gets assigned to four variables you can work with: `first`, `second`, and `third`. This may look strange, but “unpack” is probably the best word to describe what it does. It just says, “Take whatever is in `ARGV`, unpack it, and assign it to all of these variables on the left in order.”

After that, we just print them out like normal.

What You Should See

Run the program like this (and you *must* pass *three* command line arguments):

Exercise 13 Session

```
$ ruby ex13.rb first 2nd 3rd
Your first variable is: first
Your second variable is: 2nd
Your third variable is: 3rd
```

This is what you should see when you do a few different runs with different arguments:

Exercise 13 Session

```
$ ruby ex13.rb stuff things that
Your first variable is: stuff
Your second variable is: things
```

```
$ ruby ex13.rb apple orange grapefruit
Your first variable is: apple
Your second variable is: orange
Your third variable is: grapefruit
```

You can actually replace `first`, `2nd`, and `3rd` with any three things you want.

If you do not run the script correctly, then you will get an error like this:

Exercise 13 Session

```
$ ruby ex13.rb first 2nd
Your first variable is: first
Your second variable is: 2nd
Your third variable is:
```

This happens when you do not put enough arguments in the command when you run it (in this case just `first 2nd`). Notice when I run it I give it `first 2nd`, which caused it to give an error message "need more than 3 values to unpack," telling you that you didn't give it enough parameters.

Study Drills

1. Try giving fewer than three arguments to your script.
2. Write a script that has fewer arguments and one that has more. Make sure you give the unpacked variables good names.
3. Combine `gets.chomp` with `ARGV` to make a script that gets more input from a user.

Common Student Questions

What's the difference between `ARGV` and `gets.chomp`?

The difference has to do with where the user is required to give input. If they give your script inputs on the command line, then you use `ARGV`. If you want them to input using the keyboard while the script is running, then use `gets.chomp`.

Are the command line arguments strings?

Yes, they come in as strings, even if you typed numbers on the command line. Use `.to_i` to convert them, just like with `gets.chomp.to_i`.

How do you use the command line?

You should have learned to use it real quick by now, but if you need to learn it at this stage, then read the Command Line Crash Course in Appendix A.

I can't combine ARGV with `gets.chomp`.

Don't overthink it. Just slap two lines at the end of this script that uses `gets.chomp` to get something and then print it. From that start playing with more ways to use both in the same script.

Why can't I do this: `gets.chomp = x`?

Because that's backward to how it should work. Do it the way I do it, and it'll work.

This page intentionally left blank

Prompting and Passing

Let's do one exercise that uses ARGV and gets.chomp together to ask the user something specific. You will need this for the next exercise, where we learn to read and write files. In this exercise we'll use gets.chomp slightly differently by having it print a simple > prompt. This is similar to a game like Zork or Adventure.

ex14.rb

```

1  user_name = ARGV.first
2  prompt = '> '
3
4  puts "Hi #{user_name}."
5  puts "I'd like to ask you a few questions."
6  puts "Do you like me #{user_name}? ", prompt
7  likes = $stdin.gets.chomp
8
9  puts "Where do you live #{user_name}?", prompt
10 lives = $stdin.gets.chomp
11
12 puts "What kind of computer do you have?", prompt
13 computer = $stdin.gets.chomp
14
15 puts """
16 Alright, so you said #{likes} about liking me.
17 You live in #{lives}. Not sure where that is.
18 And you have a #{computer} computer. Nice.
19 """

```

We make a variable `prompt` that is set to the prompt we want, and we give that to `gets.chomp` instead of typing it over and over. Now if we want to make the prompt something else, we just change it in this one spot and rerun the script.

Very handy.

What You Should See

When you run this, remember that you have to give the script your name for the ARGV arguments.

Exercise 14 Session

```

$ ruby ex14.rb zed
Hi zed.
I'd like to ask you a few questions.

```

```
Do you like me zed?  
>  
Yes  
Where do you live zed?  
>  
San Francisco  
What kind of computer do you have?  
>  
Tandy
```

Alright, so you said Yes about liking me.
You live in San Francisco. Not sure where that is.
And you have a Tandy computer. Nice.

Study Drills

1. Find out what Zork and Adventure were. Try to find copies and play them.
2. Change the `prompt` variable to something else entirely.
3. Add another argument and use it in your script, the same way you did in the previous exercise with `first, second = ARGV`.
4. Make sure you understand how I combined a """ style multiline string with the #{} format activator as the last print.

Common Student Questions

I don't understand what you mean by changing the prompt.

See the variable `prompt = '> '`. Change that to have a different value. You know this, it's just a string and you've done 13 exercises making them, so take the time to figure it out.

Can I use double-quotes for the `prompt` variable?

You totally can. Go ahead and try that.

You have a Tandy computer?

I did when I was little.

I get undefined local variable or method 'prompt' when I run it.

You either spelled the name of the `prompt` variable wrong or forgot that line. Go back and compare each line of code to mine, from the bottom of the script to the top. Any time you see this error, it means you spelled something wrong, or forgot to create the variable.

Reading Files

You know how to get input from a user with `gets.chomp` or `ARGV`. Now you will learn about reading from a file. You may have to play with this exercise the most to understand what's going on, so do the exercise carefully and remember to do your checks. Working with files is an easy way to *erase your work* if you are not careful.

This exercise involves writing two files. One is the usual `ex15.rb` file that you will run, but the *other* is named `ex15_sample.txt`. This second file isn't a script, but a plain text file we'll be reading in our script. Here are the contents of that file:

```
This is stuff I typed into a file.  
It is really cool stuff.  
Lots and lots of fun to have in here.
```

What we want to do is "open" that file in our script and print it out. However, we do not want to just "hard code" the name `ex15_sample.txt` into our script. "Hard coding" means putting some bit of information that should come from the user as a string directly in our source code. That's bad because we want it to load other files later. The solution is to use `ARGV` or `gets.chomp` to ask the user what file to open instead of "hard coding" the file's name.

ex15.rb

```
1 filename = ARGV.first  
2  
3 txt = open(filename)  
4  
5 puts "Here's your file #{filename}:"  
6 print txt.read  
7  
8 print "Type the filename again: "  
9 file_again = $stdin.gets.chomp  
10  
11 txt_again = open(file_again)  
12  
13 print txt_again.read
```

A few fancy things are going on in this file, so let's break it down real quick:

Lines 1-2 uses `ARGV` to get a filename. Next we have line 3 where we use a new command `open`. Right now, run `ri open` and read the instructions. Notice how, just like your own scripts and `gets.chomp`, it takes a parameter and returns a value that you can set to your own variable. You just opened a file.

Line 5 prints a little message, but on line 6 we have something very new and exciting. We call a function in `txt` named `read`. What you get back from `open` is a `File`, and it also has commands you can give it. You give a file a command by using the `.` (dot or period), the name of the command, and parameters, just like with `open` and `gets.chomp`. The difference is that `txt.read` says, "Hey `txt`! Do your `read` command with no parameters!"

The remainder of the file is more of the same, but we'll leave the analysis to you in the Study Drills.

What You Should See

I made a file called `ex15_sample.txt` and ran my script.

Exercise 15 Session

```
$ ruby ex15.rb ex15_sample.txt
Here's your file ex15_sample.txt:
This is stuff I typed into a file.
It is really cool stuff.
Lots and lots of fun to have in here.

Type the filename again: ex15_sample.txt
This is stuff I typed into a file.
It is really cool stuff.
Lots and lots of fun to have in here.
```

Study Drills

This is a big jump, so be sure you do these Study Drills as well as you can before moving on.

1. Above each line, write out in English what that line does.
2. If you are not sure ask someone for help or search online. Many times searching for "ruby THING" will find answers to what that THING does in Ruby. Try searching for "ruby open."
3. I used the word "commands" here, but commands are also called "functions" and "methods." You will learn about functions and methods later in the book.
4. Get rid of lines 8–13 where you use `gets.chomp` and the script again.
5. Use only `gets.chomp` and try the script that way. Why is one way of getting the filename better than another?
6. Start `irb` to start the Ruby shell, and use `open` from the prompt just like in this program. Notice how you can open files and run `read` on them from within `irb`?
7. Have your script also call `close()` on the `txt` and `txt_again` variables. It's important to close files when you are done with them.

Common Student Questions

Does `txt = open(filename)` return the contents of the file?

No, it doesn't. It actually makes something called a "file object." You can think of a file like an old tape drive that you saw on mainframe computers in the 1950s, or even like a DVD player from today. You can move around inside them, and then "read" them, but the DVD player is not the DVD's contents the same way the file object is not the file's contents.

I can't type code into my Terminal/PowerShell like you say in Study Drill 6.

First thing, from the command line just type `irb` and press Enter. Now you are in `irb` as we've done a few other times. Then you can type in code and Ruby will run it in little pieces. Play with that. To get out of it type `quit()` and hit Enter.

Why is there no error when we open the file twice?

Ruby will not restrict you from opening a file more than once, and sometimes this is necessary.

This page intentionally left blank

Reading and Writing Files

If you did the Study Drills from the last exercise you should have seen all sorts of commands (methods/functions) you can give to files. Here's the list of commands I want you to remember:

- `close` – Closes the file. Like `File->Save..` in your editor.
- `read` – Reads the contents of the file. You can assign the result to a variable.
- `readline` – Reads just one line of a text file.
- `truncate` – Empties the file. Watch out if you care about the file.
- `write('stuff')` – Writes "stuff" to the file.

For now, these are the important commands you need to know. Some of them take parameters, but we do not really care about that. You only need to remember that `write` takes a parameter of a string you want to write to the file.

Let's use some of this to make a simple little text editor:

ex16.rb

```
1 filename = ARGV.first
2
3 puts "We're going to erase #{filename}"
4 puts "If you don't want that, hit CTRL-C (^C)."
5 puts "If you do want that, hit RETURN."
6
7 $stdin.gets
8
9 puts "Opening the file..."
10 target = open(filename, 'w')
11
12 puts "Truncating the file. Goodbye!"
13 target.truncate(0)
14
15 puts "Now I'm going to ask you for three lines."
16
17 print "line 1: "
18 line1 = $stdin.gets.chomp
19 print "line 2: "
20 line2 = $stdin.gets.chomp
21 print "line 3: "
22 line3 = $stdin.gets.chomp
23
```

```
24 puts "I'm going to write these to the file."
25
26 target.write(line1)
27 target.write("\n")
28 target.write(line2)
29 target.write("\n")
30 target.write(line3)
31 target.write("\n")
32
33 puts "And finally, we close it."
34 target.close
```

That's a large file, probably the largest you have typed in. So go slow, do your checks, and make it run. One trick is to get bits of it running at a time. Get lines 1–8 running, then five more, then a few more, until it's all done and running.

What You Should See

There are actually two things you will see. First the output of your new script:

Exercise 16 Session

```
$ ruby ex16.rb test.txt
We're going to erase test.txt
If you don't want that, hit CTRL-C (^C).
If you do want that, hit RETURN.

Opening the file...
Truncating the file. Goodbye!
Now I'm going to ask you for three lines.
line 1: I am the first line.
line 2: I am the second line.
line 3: I am the third line.
I'm going to write these to the file.
And finally, we close it.
```

Now, open up the file you made (in my case, `test.txt`) in your editor and check it out. Neat, right?

Study Drills

1. If you do not understand this script, go back through and use the comment trick to get it squared away in your mind. One simple English comment above each line will help you understand or at least let you know what you need to research more.
2. Write a script similar to the last exercise that uses `read` and `ARGV` to read the file you just created.

3. There's too much repetition in this file. Use strings, formats, and escapes to print out `line1`, `line2`, and `line3` with just one `target.write()` command instead of six.
4. Find out why we had to pass a '`w`' as an extra parameter to open. Hint: open tries to be safe by making you explicitly say you want to write a file.
5. If you open the file with '`w`' mode, then do you really need the `target.truncate()`? Read the documentation for Ruby's open function and see if that's true.

Common Student Questions

Is the `truncate()` necessary with the '`w`' parameter?

See Study Drills 5.

What does '`w`' mean?

It's really just a string with a character in it for the kind of mode for the file. If you use '`w`' then you're saying "open this file in 'write' mode," thus the '`w`' character. There's also '`r`' for "read," '`a`' for "append," and modifiers on these.

What modifiers to the file modes can I use?

The most important one to know for now is the '+' modifier, so you can do '`w+`', '`r+`', and '`a+`'. This will open the file in both read and write mode, and depending on the character use position the file in different ways.

Does just doing `open(filename)` open it in '`r`' (read) mode?

Yes, that's the default for the `open()` function.

This page intentionally left blank

More Files

Now let's do a few more things with files. We'll write a Ruby script to copy one file to another. It'll be very short, but will give you ideas about other things you can do with files.

ex17.rb

```

1  from_file, to_file = ARGV
2
3  puts "Copying from #{from_file} to #{to_file}"
4
5  # we could do these two on one line, how?
6  in_file = open(from_file)
7  indata = in_file.read
8
9  puts "The input file is #{indata.length} bytes long"
10
11 puts "Does the output file exist? #{File.exist?(to_file)}"
12 puts "Ready, hit RETURN to continue, CTRL-C to abort."
13 $stdin.gets
14
15 out_file = open(to_file, 'w')
16 out_file.write(indata)
17
18 puts "Alright, all done."
19
20 out_file.close
21 in_file.close

```

The important thing to learn from this script is the function `File.exist?(to_file)` on line 11. This can be broken down as, “File! I want you to use your `exist?` function to tell me if `to_file` exists on the disk.” Yet another way to say this is, “Get the `exist?` function from `File` and call it with the variable `to_file`.” You’ll learn more about this later, but for now you should study how you can call functions inside `File` to do things with files.

What You Should See

Just like your other scripts, run this one with two arguments: the file to copy from and the file to copy it to. I’m going to use a simple test file named `test.txt`:

Exercise 17 Session

```

$ echo "This is a test file." > test.txt
$ cat test.txt

```

```
This is a test file.  
$ ruby ex17.rb test.txt new_file.txt  
Copying from test.txt to new_file.txt  
The input file is 21 bytes long  
Does the output file exist? false  
Ready, hit RETURN to continue, CTRL-C to abort.
```

Alright, all done.

It should work with any file. Try a bunch more and see what happens. Just be careful that you do not blast an important file.

WARNING! Did you see the trick I did with echo to make a file, and cat to show the file? You can learn how to do that in Appendix A.

Study Drills

1. This script is *really* annoying. There's no need to ask you before doing the copy, and it prints too much out on the screen. Try to make the script more friendly to use by removing features.
2. See how short you can make the script. I could make this one line long.
3. Notice at the end of the *What You Should See* I used something called cat? It's an old command that "con*cat*enates" files together, but mostly it's just an easy way to print a file to the screen. Type man cat to read about it.
4. Find out why you had to write out_file.close in the code.

Common Student Questions

Why is the 'w' in quotes?

That's a string. You've been using strings for a while now. Make sure you know what a string is.

No way you can make this one line!

That ; depends ; on ; how ; you ; define ; one ; line ; of ; code.

Is it normal to feel like this exercise was really hard?

Yes, it is totally normal. Programming may not "click" for you until maybe even Exercise 36, or it might not until you finish the book and then make something with Ruby. Everyone is different, so just keep going and keep reviewing exercises that you had trouble with until everything clicks. Be patient.

Names, Variables, Code, Functions

Big title, right? I am about to introduce you to *the function!* Dum dum dah! Every programmer will go on and on about functions and all the different ideas about how they work and what they do, but I will give you the simplest explanation you can use right now.

Functions do three things:

1. They name pieces of code the way variables name strings and numbers.
2. They take arguments the way your scripts take ARGV.
3. Using 1 and 2 they let you make your own “mini-scripts” or “tiny commands.”

You can create a function by using the word `def` in Ruby. I’m going to have you make four different functions that work like your scripts, and I’ll then show you how each one is related.

`ex18.rb`

```

1  # this one is like your scripts with ARGV
2  def print_two(*args)
3      arg1, arg2 = args
4      puts "arg1: #{arg1}, arg2: #{arg2}"
5  end
6
7  # ok, that *args is actually pointless, we can just do this
8  def print_two_again(arg1, arg2)
9      puts "arg1: #{arg1}, arg2: #{arg2}"
10 end
11
12 # this just takes one argument
13 def print_one(arg1)
14     puts "arg1: #{arg1}"
15 end
16
17 # this one takes no arguments
18 def print_none()
19     puts "I got nothin'."
20 end
21
22
23 print_two("Zed", "Shaw")
24 print_two_again("Zed", "Shaw")
25 print_one("First!")
26 print_none()
```

Let's break down the first function, `print_two`, which is the most similar to what you already know from making scripts:

1. First we tell Ruby we want to make a function using `def` for "define."
2. On the same line as `def` we give the function a name. In this case we just called it "`print_two`" but it could also be "`peanuts`." It doesn't matter, except that your function should have a short name that says what it does.
3. Then we tell it we want `*args` (asterisk args) which is a lot like your `ARGV` parameter but for functions. This *has to* go inside `()` parentheses to work.
4. Then we end this line with a newline (ENTER key), and start indenting.
5. After the colon all the lines that are indented four spaces will become attached to this name, `print_two`. Our first indented line is one that unpacks the arguments, the same as with your other scripts.
6. To demonstrate how it works we print these arguments out, just like we would in a script.

The problem with `print_two` is that it's not the easiest way to make a function. In Ruby we can skip the whole unpacking arguments and just use the names we want right inside `()`. That's what `print_two_again` does.

After that, you have an example of how you make a function that takes one argument in `print_one`.

Finally you have a function that has no arguments in `print_none`.

WARNING! This is very important. Do *not* get discouraged right now if this doesn't quite make sense. We're going to do a few exercises that focus on linking functions to your scripts and show you how to make more. For now, just keep thinking "mini-script" when I say "function" and keep playing with them.

What You Should See

If you run `ex18.rb` you should see:

Exercise 18 Session

```
$ ruby ex18.rb
arg1: Zed, arg2: Shaw
arg1: Zed, arg2: Shaw
arg1: First!
I got nothin'.
```

Right away you can see how a function works. Notice that you used your newly created functions the way you use things like `exists`, `open`, and other "commands." In fact, I've been tricking you because in Ruby those "commands" are just functions. This means you can make your own commands and use them in your scripts, too.

Study Drills

Create a *function checklist* for later exercises. Write these checks on an index card and keep it by you while you complete the rest of these exercises, or until you feel you do not need the index card anymore:

1. Did you start your function definition with `def`?
2. Does your function name have only characters and `_` (underscore) characters?
3. Did you put an open parenthesis `(` right after the function name?
4. Did you put your arguments after the parenthesis `(` separated by commas?
5. Did you make each argument unique (meaning no duplicated names)?
6. Did you put a close parenthesis `)` after the arguments?
7. Did you indent all lines of code you want in the function two spaces?
8. Did you end your function with `end` lined up with the `def` above?

When you run ("use" or "call") a function, check these things:

1. Did you call/use/run this function by typing its name?
2. Did you put the `(` character after the name to run it?
3. Did you put the values you want into the parentheses separated by commas?
4. Did you end the function call with a `)` character?
5. Functions that don't have parameters do not need the `()` after them, but would it be clearer if you included them anyway.

Use these two checklists on the remaining lessons until you do not need them anymore.

Finally, repeat this a few times to yourself:

"To 'run', 'call', or 'use' a function all mean the same thing."

Common Student Questions

What's allowed for a function name?

The same as variable names. Anything that doesn't start with a number, and is letters, numbers, and underscores will work.

What does the * in *args do?

That tells Ruby to take all the arguments to the function and then put them in args as a list. It's like ARGV that you've been using, but for functions. It's not normally used too often unless specifically needed.

This feels really boring and monotonous.

That's good. It means you're starting to get better at typing in the code and understanding what it does. To make it less boring, take everything I tell you to type in, and then break it on purpose.

Functions and Variables

Functions may have been a mind-blowing amount of information, but do not worry. Just keep doing these exercises and going through your checklist from the last exercise and you will eventually get it.

There is one tiny point that you might not have realized, which we'll reinforce right now. The variables in your function are not connected to the variables in your script. Here's an exercise to get you thinking about this:

ex19.rb

```
1  def cheese_and_crackers(cheese_count, boxes_of_crackers)
2      puts "You have #{cheese_count} cheeses!"
3      puts "You have #{boxes_of_crackers} boxes of crackers!"
4      puts "Man that's enough for a party!"
5      puts "Get a blanket.\n"
6  end
7
8
9  puts "We can just give the function numbers directly:"
10 cheese_and_crackers(20, 30)
11
12
13 puts "OR, we can use variables from our script:"
14 amount_of_cheese = 10
15 amount_of_crackers = 50
16
17 cheese_and_crackers(amount_of_cheese, amount_of_crackers)
18
19
20 puts "We can even do math inside too:"
21 cheese_and_crackers(10 + 20, 5 + 6)
22
23
24 puts "And we can combine the two, variables and math:"
25 cheese_and_crackers(amount_of_cheese + 100, amount_of_crackers + 1000)
```

This shows all the different ways we're able to give our function `cheese_and_crackers` the values it needs to print them. We can give it straight numbers. We can give it variables. We can give it math. We can even combine math and variables.

In a way, the arguments to a function are kind of like `our = character` when we make a variable. In fact, if you can use `=` to name something, you can usually pass it to a function as an argument.

What You Should See

You should study the output of this script and compare it with what you think you should get for each of the examples in the script.

Exercise 19 Session

```
$ ruby ex19.rb
We can just give the function numbers directly:
You have 20 cheeses!
You have 30 boxes of crackers!
Man that's enough for a party!
Get a blanket.
OR, we can use variables from our script:
You have 10 cheeses!
You have 50 boxes of crackers!
Man that's enough for a party!
Get a blanket.
We can even do math inside too:
You have 30 cheeses!
You have 11 boxes of crackers!
Man that's enough for a party!
Get a blanket.
And we can combine the two, variables and math:
You have 110 cheeses!
You have 1050 boxes of crackers!
Man that's enough for a party!
Get a blanket.
```

Study Drills

1. Go back through the script and type a comment above each line explaining in English what it does.
2. Start at the bottom and read each line backward, saying all the important characters.
3. Write at least one more function of your own design, and run it 10 different ways.

Common Student Questions

How can there possibly be 10 different ways to run a function?

Believe it or not, there's a theoretically infinite number of ways to call any function. See how creative you can get with functions, variables, and input from a user.

Is there a way to analyze what this function is doing so I can understand it better?

There are many different ways, but try putting an English comment above each line describing what the line does. Another trick is to read the code out loud. Yet another is to print the code out and draw on the paper with pictures and comments showing what's going on.

What if I want to ask the user for the numbers of cheese and crackers?

You need to use `.to_i` to convert what you get from `gets.chomp`.

Does making the variable `amount_of_cheese` change the variable `cheese_count` in the function?

No, those variables are separate and live outside the function. They are then passed to the function and temporary versions are made just for the function's run. When the function exits, these temporary variables go away and everything keeps working. Keep going in the book and this should become clearer.

Is it bad to have global variables (like `amount_of_cheese`) with the same name as function variables?

Yes, since then you're not quite sure which one you're talking about. But sometimes necessity means you have to use the same name, or you might do it by accident. Just avoid it whenever you can.

Is there a limit to the number of arguments a function can have?

It depends on the version of Ruby and the computer you're on, but it is fairly large. The practical limit, though, is about five arguments before the function becomes annoying to use.

Can you call a function within a function?

Yes, you'll make a game that does this later in the book.

This page intentionally left blank

Functions and Files

Remember your checklist for functions, then do this exercise paying close attention to how functions and files can work together to make useful stuff.

ex20.rb

```
1  input_file = ARGV.first
2
3  def print_all(f)
4      puts f.read
5  end
6
7  def rewind(f)
8      f.seek(0)
9  end
10
11 def print_a_line(line_count, f)
12     puts "#{line_count}, #{f.gets.chomp}"
13 end
14
15 current_file = open(input_file)
16
17 puts "First let's print the whole file:\n"
18
19 print_all(current_file)
20
21 puts "Now let's rewind, kind of like a tape."
22
23 rewind(current_file)
24
25 puts "Let's print three lines:"
26
27 current_line = 1
28 print_a_line(current_line, current_file)
29
30 current_line = current_line + 1
31 print_a_line(current_line, current_file)
32
33 current_line = current_line + 1
34 print_a_line(current_line, current_file)
```

Pay close attention to how we pass in the current line number each time we run `print_a_line`.

What You Should See

Exercise 20 Session

```
$ ruby ex20.rb test.txt
First let's print the whole file:
This is line 1
This is line 2
This is line 3
Now let's rewind, kind of like a tape.
Let's print three lines:
1, This is line 1
2, This is line 2
3, This is line 3
```

Study Drills

1. Write English comments for each line to understand what that line does.
2. Each time `print_a_line` is run, you are passing a variable `current_line`. Write out what `current_line` is equal to on each function call, and trace how it becomes `line_count` in `print_a_line`.
3. Find each place a function is used, and check its `def` to make sure that you are giving it the right arguments.
4. Research online what the `seek` function for `file` does. Try `ri File` and see if you can figure it out from there.
5. Research the shorthand notation `+=` and rewrite the script to use `+=` instead.

Common Student Questions

What is `f` in the `print_all` and other functions?

The `f` is a variable just like you had in other functions in Exercise 18, except this time it's a file. A file in Ruby is kind of like an old tape drive on a mainframe, or maybe a DVD player. It has a "read head," and you can "seek" this read head around the file to reach specific positions, then work with it there. Each time you do `f.seek(0)`, you're moving to the start of the file. Each time you do `f.readline()`, you're reading a line from the file, and moving the read head to right after the `\n` that ends that file. This will be explained more as you go on.

Why are there empty lines between the lines in the file?

The `readline()` function returns the `\n` that's in the file at the end of that line. Add a `,` at the end of your `print` function calls to avoid adding double `\n` to every line.

Why does `seek(0)` not set the `current_line` to 0?

First, the `seek()` function is dealing in *bytes*, not lines. The code `seek(0)` moves the file to the 0 byte (first byte) in the file. Second, `current_line` is just a variable and has no real connection to the file at all. We are manually incrementing it.

What is `+=?`?

You know how in English I can rewrite "it is" as "it's"? Or I can rewrite "you are" as "you're"? In English this is called a contraction, and this is kind of like a contraction for the two operations `=` and `+`. That means `x = x + y` is the same as `x += y`.

How does `readline()` know where each line is?

Inside `readline()` is code that scans each byte of the file until it finds a `\n` character, then stops reading the file to return what it has found so far. The file `f` is responsible for maintaining the current position in the file after each `readline()` call, so that it will keep reading each line.

This page intentionally left blank

Functions Can Return Something

You have been using the = character to name variables and set them to numbers or strings.

We're now going to blow your mind again by showing you how to use = and a new Ruby word return to set variables to be a *value from a function*. There will be one thing to pay close attention to, but first type this in:

ex21.rb

```
1  def add(a, b)
2      puts "ADDING #{a} + #{b}"
3      return a + b
4  end
5
6  def subtract(a, b)
7      puts "SUBTRACTING #{a} - #{b}"
8      return a - b
9  end
10
11 def multiply(a, b)
12     puts "MULTIPLYING #{a} * #{b}"
13     return a * b
14 end
15
16 def divide(a, b)
17     puts "DIVIDING #{a} / #{b}"
18     return a / b
19 end
20
21
22 puts "Let's do some math with just functions!"
23
24 age = add(30, 5)
25 height = subtract(78, 4)
26 weight = multiply(90, 2)
27 iq = divide(100, 2)
28
29 puts "Age: #{age}, Height: #{height}, Weight: #{weight}, IQ: #{iq}"
30
31
32 # A puzzle for the extra credit, type it in anyway.
33 puts "Here is a puzzle."
34
35 what = add(age, subtract(height, multiply(weight, divide(iq, 2))))
36
37 puts "That becomes: #{what}. Can you do it by hand?"
```

We are now doing our own math functions for add, subtract, multiply, and divide. The important thing to notice is the last line where we say `return a + b` (in add). What this does is the following:

1. Our function is called with two arguments: `a` and `b`.
2. We print out what our function is doing, in this case "ADDING."
3. Then we tell Ruby to do something kind of backward: we return the addition of `a + b`. You might say this as, "I add `a` and `b` and then return them."
4. Ruby adds the two numbers. Then when the function ends, any line that runs it will be able to assign this `a + b` result to a variable.

As with many other things in this book, you should take this real slow, break it down, and try to trace what's going on. To help there is extra credit to solve a puzzle and learn something cool.

What You Should See

Exercise 21 Session

```
$ ruby ex21.rb
Let's do some math with just functions!
ADDING 30 + 5
SUBTRACTING 78 - 4
MULTIPLYING 90 * 2
DIVIDING 100 / 2
Age: 35, Height: 74, Weight: 180, IQ: 50
Here is a puzzle.
DIVIDING 50 / 2
MULTIPLYING 180 * 25
SUBTRACTING 74 - 4500
ADDING 35 + -4426
That becomes: -4391. Can you do it by hand?
```

Study Drills

1. If you aren't really sure what `return` does, try writing a few of your own functions and have them return some values. You can return anything that you can put to the right of an `=`.
2. At the end of the script is a puzzle. I'm taking the return value of one function and *using* it as the argument of another function. I'm doing this in a chain so that I'm kind of creating a formula using the functions. It looks really weird, but if you run the script you can see the results. What you should do is try to figure out the normal formula that would recreate this same set of operations.

3. Once you have the formula worked out for the puzzle, get in there and see what happens when you modify the parts of the functions. Try to change it on purpose to make another value.
4. Do the inverse. Write a simple formula and use the functions in the same way to calculate it.
5. Remove the word `return` and see if the script still works. You'll find that it does because Ruby implicitly returns whatever the last expression calculates. However, this isn't clear so I want you to do it explicitly for my book.

This exercise might really whack your brain out, but take it slow and easy and treat it like a little game. Figuring out puzzles like this is what makes programming fun, so I'll be giving you more little problems like this as we go.

Common Student Questions

Why does Ruby print the formula or the functions "backward"?

It's not really backward; it's "inside out." When you start breaking down the function into separate formulas and function calls you'll see how it works. Try to understand what I mean by "inside out" rather than "backward."

What do you mean by "write out a formula"?

Try `24 + 34 / 100 - 1023` as a start. Convert that to use the functions. Now come up with your own similar math equation and use variables so it's more like a formula.

This page intentionally left blank

What Do You Know So Far?

There won't be any code in this exercise or the next one, so there's no *What You Should See* or *Study Drills* either. In fact, this exercise is like one giant Study Drill. I'm going to have you do a review of what you have learned so far.

First, go back through every exercise you have done so far and write down every word and symbol (another name for "character") that you have used. Make sure your list of symbols is complete.

Next to each word or symbol, write its name and what it does. If you can't find a name for a symbol in this book, then look for it online. If you do not know what a word or symbol does, then read about it again and try using it in some code.

You may run into a few things you can't find out or know, so just keep those on the list and be ready to look them up when you find them.

Once you have your list, spend a few days rewriting the list and double-checking that it's correct. This may get boring but push through and really nail it down.

Once you have memorized the list and what they do, then step it up by writing out tables of symbols, their names, and what they do *from memory*. If you hit some you can't recall from memory, go back and memorize them again.

WARNING! The most important thing when doing this exercise is: "There is no failure, only trying."

What You Are Learning

It's important when you are doing a boring, mindless memorization exercise like this to know why. It helps you focus on a goal and know the purpose of all your efforts.

In this exercise you are learning the names of symbols so that you can read source code more easily. It's similar to learning the alphabet and basic words of English, except this Ruby alphabet has extra symbols you might not know.

Just take it slow and do not hurt your brain. It's best to take 15 minutes at a time with your list and then take a break. Giving your brain a rest will help you learn faster with less frustration.

This page intentionally left blank

Read Some Code

You should have spent the last week getting your list of symbols straight and locked in your mind. Now you get to apply this to another week of reading code on the Internet. This exercise will be daunting at first. I'm going to throw you in the deep end for a few days and have you just try your best to read and understand some source code from real projects. The goal isn't to get you to understand code, but to teach you the following three skills:

1. Finding Ruby source code for things you need.
2. Reading through the code and looking for files.
3. Trying to understand code you find.

At your level, you really do not have the skills to evaluate the things you find, but you can benefit from getting exposure and seeing how things look.

When you do this exercise, think of yourself as an anthropologist, trucking through a new land with just barely enough of the local language to get around and survive. Except, of course, that you will actually get out alive because the Internet isn't a jungle.

Here's what you do:

1. Go to bitbucket.org, github.com, or gitorious.org with your favorite web browser and search for "ruby."
2. Pick a random project and click on it.
3. Click on the Source tab and browse through the list of files and directories until you find a .rb file.
4. Start at the top and read through the code, taking notes on what you think it does.
5. If any symbols or strange words seem to interest you, write them down to research later.

That's it. Your job is to use what you know so far and see if you can read the code and get a grasp of what it does. Try skimming the code first, and then read it in detail. Try taking very difficult parts and read each symbol you know out loud.

Now try some of these other sites:

- launchpad.net
- sourceforge.net

This page intentionally left blank

More Practice

You are getting to the end of this section. You should have enough Ruby "under your fingers" to move on to learning about how programming really works, but you should do some more practice. This exercise is longer and all about building up stamina. The next exercise will be similar. Do them, get them exactly right, and do your checks.

ex24.rb

```
1 puts "Let's practice everything."
2 puts 'You\'d need to know \'bout escapes with \\ that do \n newlines and \t tabs.'
3
4 poem = <<END
5   \tThe lovely world
6   with logic so firmly planted
7   cannot discern \n the needs of love
8   nor comprehend passion from intuition
9   and requires an explanation
10  \n\t\twhere there is none.
11 END
12
13 puts "-----"
14 puts poem
15 puts "-----"
16
17
18 five = 10 - 2 + 3 - 6
19 puts "This should be five: #{five}"
20
21 def secret_formula(started)
22   jelly_beans = started * 500
23   jars = jelly_beans / 1000
24   crates = jars / 100
25   return jelly_beans, jars, crates
26 end
27
28
29 start_point = 10000
30 beans, jars, crates = secret_formula(start_point)
31
32 puts "With a starting point of: #{start_point}"
33 puts "We'd have #{beans} beans, #{jars} jars, and #{crates} crates."
34
35 start_point = start_point / 10
```

What You Should See

Exercise 24 Session

```
$ ruby ex24.rb
Let's practice everything.
You'd need to know 'bout escapes with \ that do \n newlines and \t tabs.
-----
The lovely world
with logic so firmly planted
cannot discern
the needs of love
nor comprehend passion from intuition
and requires an explanation

      where there is none.
-----
This should be five: 5
With a starting point of: 10000
We'd have 5000000 beans, 5000 jars, and 50 crates.
```

Study Drills

1. Make sure to do your checks: read it backward, read it out loud, and put comments above confusing parts.
2. Break the file on purpose, then run it to see what kinds of errors you get. Make sure you can fix it.

Common Student Questions

Why do you call the variable `jelly_beans` but the name `beans` later?

That's part of how a function works. Remember that inside the function the variable is temporary. When you return it, then it can be assigned to a variable for later. I'm just making a new variable named `beans` to hold the return value.

What do you mean by reading the code backward?

Start at the last line. Compare that line in your file to the same line in mine. Once it's exactly the same, move up to the next line. Do this until you get to the first line of the file.

Who wrote that poem?

I did. Not all of my poems suck.

Even More Practice

We're going to do some more practice involving functions and variables to make sure you know them well. This exercise should be straightforward for you to type in, break down, and understand.

However, this exercise is a little different. You won't be running it. Instead, you will import it into Ruby and run the functions yourself.

ex25.rb

```
1 module Ex25
2
3     # This function will break up words for us.
4     def Ex25.break_words(stuff)
5         words = stuff.split(' ')
6         return words
7     end
8
9     # Sorts the words.
10    def Ex25.sort_words(words)
11        return words.sort
12    end
13
14    # Prints the first word after shifting it off.
15    def Ex25.print_first_word(words)
16        word = words.shift
17        puts word
18    end
19
20    # Prints the last word after popping it off.
21    def Ex25.print_last_word(words)
22        word = words.pop
23        puts word
24    end
25
26    # Takes in a full sentence and returns the sorted words.
27    def Ex25.sort_sentence(sentence)
28        words = Ex25.break_words(sentence)
29        return Ex25.sort_words(words)
30    end
31
32    # Prints the first and last words of the sentence.
33    def Ex25.print_first_and_last(sentence)
34        words = Ex25.break_words(sentence)
```

```
35     Ex25.print_first_word(words)
36     Ex25.print_last_word(words)
37   end
38
39   # Sorts the words then prints the first and last one.
40   def Ex25.print_first_and_last_sorted(sentence)
41     words = Ex25.sort_sentence(sentence)
42     Ex25.print_first_word(words)
43     Ex25.print_last_word(words)
44   end
45
46 end
```

First, run this with `ruby ex25.rb` to find any errors you have made. Once you have found all of the errors you can and fixed them, you will then want to follow the *What You Should See* section to complete the exercise.

What You Should See

In this exercise we're going to interact with your `ex25.rb` file inside the `irb` interpreter you used periodically to do calculations. You run `irb` from the terminal like this:

```
>>>
$ irb
irb(main):001:0>
```

Your output should look like mine, and after the `>` character (called the prompt) you can type Ruby code in and it will run immediately. Using this I want you to type each of these lines of Ruby code into `irb` and see what it does:

Exercise 25 Ruby Session

```
1  require "./ex25.rb"
2
3  sentence = "All good things come to those who wait."
4  words = Ex25.break_words(sentence)
5  words
6  sorted_words = Ex25.sort_words(words)
7  sorted_words
8  Ex25.print_first_word(words)
9  Ex25.print_last_word(words)
10 words
11 Ex25.print_first_word(sorted_words)
12 Ex25.print_last_word(sorted_words)
13 sorted_words
14 sorted_words = Ex25.sort_sentence(sentence)
15 sorted_words
16 Ex25.print_first_and_last(sentence)
17 Ex25.print_first_and_last_sorted(sentence)
```

Here's what it looks like when I work with the ex25.rb module in irb:

Exercise 25 Ruby Session

```
>> require "./ex25.rb"
=> true
>>
=?> sentence = "All good things come to those who wait."
=> "All good things come to those who wait."
>> words = Ex25.break_words(sentence)
=> ["All", "good", "things", "come", "to", "those", "who", "wait."]
>> words
=> ["All", "good", "things", "come", "to", "those", "who", "wait."]
>> sorted_words = Ex25.sort_words(words)
=> ["All", "come", "good", "things", "those", "to", "wait.", "who"]
>> sorted_words
=> ["All", "come", "good", "things", "those", "to", "wait.", "who"]
>> Ex25.print_first_word(words)
=> nil
>> Ex25.print_last_word(words)
wait.
=> nil
>> words
=> ["All", "good", "things", "come", "to", "those", "who"]
>> Ex25.print_first_word(sorted_words)
=> nil
>> Ex25.print_last_word(sorted_words)
who
=> nil
>> sorted_words
=> ["All", "come", "good", "things", "those", "to", "wait."]
>> sorted_words = Ex25.sort_sentence(sentence)
=> ["All", "come", "good", "things", "those", "to", "wait.", "who"]
>> sorted_words
=> ["All", "come", "good", "things", "those", "to", "wait.", "who"]
>> Ex25.print_first_and_last(sentence)
wait.
=> nil
>> Ex25.print_first_and_last_sorted(sentence)
who
=> nil
```

As you go through each of these lines, make sure you can find the function that's being run in ex25.rb and you understand how each one works. If you get different results or errors, you'll have to fix your code, exit irb, and start over.

Study Drills

1. Take the remaining lines of the *What You Should See* output and figure out what they are doing. Make sure you understand how you are running your functions in the ex25 module.
2. The Ex25 module doesn't have to be in a file named ex25.rb. Try putting it in a new file with a random name, then import that file and see how you still have Ex25 available, even though the file you made does not have ex25 in it.

Common Student Questions

How can the words.shift be changing the words variable?

That's a complicated question, but in this case words is a list, and because of that you can give it commands and it'll retain the results of those commands. This is similar to how files and many other things worked when you were working with them.

When should I use puts instead of return in a function?

The return from a function gives the line of code that called the function a result. You can think of a function as taking input through its arguments, and returning output through return. The puts is *completely* unrelated to this, and only deals with printing output to the terminal.

Congratulations, Take a Test!

You are almost done with the first half of the book. The second half is where things get interesting. You will learn logic and be able to do useful things like make decisions.

Before you continue, I have a quiz for you. This quiz will be very hard because it requires you to fix someone else's code. When you are a programmer you often have to deal with other programmers' code, and also with their arrogance. Programmers will very frequently claim that their code is perfect.

These programmers are stupid people who care little for others. A good programmer assumes, like a good scientist, that there's always *some* probability their code is wrong. Good programmers start from the premise that their software is broken and then work to rule out all possible ways it could be wrong before finally admitting that maybe it really is the other guy's code.

In this exercise, you will practice dealing with a bad programmer by fixing a bad programmer's code. I have poorly copied Exercises 24 and 25 into a file and removed random characters and added flaws. Most of the errors are things Ruby will tell you, while some of them are math errors you should find. Others are formatting errors or spelling mistakes in the strings.

All of these errors are very common mistakes all programmers make. Even experienced ones.

Your job in this exercise is to correct this file. Use all of your skills to make this file better. Analyze it first, maybe printing it out to edit it like you would a school term paper. Fix each flaw and keep running it and fixing it until the script runs perfectly. Try not to get help. If you get stuck take a break and come back to it later.

Even if this takes days to do, bust through it and make it right.

The point of this exercise isn't to type it in but to fix an existing file. To do that, you must go to this site:

- <http://learnrubythehardway.org/book/exercise26.txt>

Copy-paste the code into a file named `ex26.rb`. This is the only time you are allowed to copy-paste.

Common Student Questions

Do I have to require `ex25.rb` or can I just remove the references to it?

You can do either. This file has the functions from `ex25` though, so first go with removing references to it.

Can I run the code while I'm fixing it?

You most certainly may. The computer is there to help, so use it as much as possible.

This page intentionally left blank

Memorizing Logic

Today is the day you start learning about logic. Up to this point you have done everything you possibly can reading and writing files to the Terminal, and have learned quite a lot about the math capabilities of Ruby.

From now on, you will be learning *logic*. You won't learn complex theories that academics love to study but just the simple basic logic that makes real programs work and that real programmers need every day.

Learning logic has to come after you do some memorization. I want you to do this exercise for an entire week. Do not falter. Even if you are bored out of your mind, keep doing it. This exercise has a set of logic tables you must memorize to make it easier for you to do the later exercises.

I'm warning you this won't be fun at first. It will be downright boring and tedious, but this teaches you a very important skill you will need as a programmer. You *will* need to be able to memorize important concepts in your life. Most of these concepts will be exciting once you get them. You will struggle with them, like wrestling a squid, and then one day you will understand it. All that work memorizing the basics pays off big later.

Here's a tip on how to memorize something without going insane: Do a tiny bit at a time throughout the day and mark down what you need to work on most. Do not try to sit down for two hours straight and memorize these tables. This won't work. Your brain will retain whatever you studied in the first 15 or 30 minutes anyway. Instead, create a bunch of index cards with each column on the left (true or false) on the front, and the column on the right on the back. You should then take them out, see the "true or false," and immediately say "true!" Keep practicing until you can do this.

Once you can do that, start writing out your own truth tables each night into a notebook. Do not just copy them. Try to do them from memory. When you get stuck glance quickly at the ones I have here to refresh your memory. Doing this will train your brain to remember the whole table.

Do not spend more than one week on this, because you will be applying it as you go.

The Truth Terms

In Ruby, we have the following terms (characters and phrases) for determining if something is "true" or "false." Logic on a computer is all about seeing if some combination of these characters and some variables is true at that point in the program.

- `&&` (and)
- `||` (or)
- `!` (not)
- `!=` (not equal)
- `==` (equal)
- `>=` (greater-than-equal)
- `<=` (less-than-equal)
- `true`
- `false`

You actually have run into these characters before, but maybe not the terms. The terms (and, or, not) actually work the way you expect them to, just like in English.

The Truth Tables

We will now use these characters to make the truth tables you need to memorize.

NOT	true?
<code>!false</code>	true
<code>!true</code>	false

OR ()	true?
<code>true false</code>	true
<code>true true</code>	true
<code>false true</code>	true
<code>false false</code>	false

AND (&&)	true?
<code>true && false</code>	false
<code>true && true</code>	true
<code>false && true</code>	false
<code>false && false</code>	false

(Continues)

NOT OR	true?
not (true false)	false
not (true true)	false
not (false true)	false
not (false false)	true

NOT AND	true?
!(true && false)	true
!(true && true)	false
!(false && true)	true
!(false && false)	true

\neq	true?
$1 \neq 0$	true
$1 \neq 1$	false
$0 \neq 1$	true
$0 \neq 0$	false

\equiv	true?
$1 \equiv 0$	false
$1 \equiv 1$	true
$0 \equiv 1$	false
$0 \equiv 0$	true

Now use these tables to write up your own cards and spend the week memorizing them. Remember, though, there is no failing in this book—just trying as hard as you can each day, and then a *little* bit more.

Common Student Question

Can't I just learn the concepts behind Boolean algebra and not memorize this?

Sure, you can do that, but then you'll have to constantly go through the rules for Boolean algebra while you code. If you memorize these first, it not only builds your memorization skills, but it also makes these operations natural. After that, the concept of Boolean algebra is easy. But do whatever works for you.

This page intentionally left blank

Boolean Practice

The logic combinations you learned from the last exercise are called "Boolean" logic expressions. Boolean logic is used *everywhere* in programming. It is an essential, fundamental part of computation and knowing them very well is akin to knowing your scales in music.

In this exercise you will take the logic exercises you memorized and start trying them out in Ruby. Take each of these logic problems and write what you think the answer will be. In each case it will be either true or false. Once you have the answers written down, you will start Ruby on your Terminal and type each logic problem in to confirm your answers.

1. `true && true`
2. `false && true`
3. `1 == 1 && 2 == 1`
4. `"test" == "test"`
5. `1 == 1 || 2 != 1`
6. `true && 1 == 1`
7. `false && 0 != 0`
8. `true || 1 == 1`
9. `"test" == "testing"`
10. `1 != 0 && 2 == 1`
11. `"test" != "testing"`
12. `"test" == 1`
13. `!(true && false)`
14. `!(1 == 1 && 0 != 1)`
15. `!(10 == 1 || 1000 == 1000)`
16. `!(1 != 10 || 3 == 4)`
17. `!("testing" == "testing" && "Zed" == "Cool Guy")`
18. `1 == 1 && !(("testing" == 1 || 1 == 0))`
19. `"chunky" == "bacon" && !(3 == 4 || 3 == 3))`
20. `3 == 3 && !(("testing" == "testing" || "Ruby" == "Fun"))`

I will also give you a trick to help you figure out the more complicated ones toward the end.

Whenever you see these Boolean logic statements, you can solve them easily by this simple process:

1. Find an equality test (`==` or `!=`) and replace it with its truth.
2. Find each `&&/||` inside parentheses and solve those first.
3. Find each `!` and invert it.
4. Find any remaining `&&/||` and solve it.
5. When you are done, you should have true or false.

I will demonstrate with a variation on problem 20:

```
3 != 4 && !("testing" != "test" || "Ruby" == "Ruby")
```

Here's me going through each of the steps and showing you the translation until I've boiled it down to a single result:

1. Solve each equality test:

```
3 != 4 is true: true && !( "testing" != "test" || "Ruby" == "Ruby")  
"testing" != "test" is true: true && !(true || "Ruby" == "Ruby")  
"Ruby" == "Ruby": true && !(true || true)
```

2. Find each `&&/||` in parentheses ():

```
(true || true) is true: true && !(true)
```

3. Find each `!` and invert it:

```
!(true) is false: true && false
```

4. Find any remaining `&&/||` and solve them:

```
true && false is false
```

With that we're done and know the result is false.

WARNING! The more complicated ones may seem very hard at first. You should be able to give a good first stab at solving them, but do not get discouraged. I'm just getting you primed for more of these "logic gymnastics" so that later cool stuff is much easier. Just stick with it, and keep track of what you get wrong, but do not worry that it's not getting in your head quite yet. It'll come.

What You Should See

After you have tried to guess at these, this is what your session with Ruby might look like:

```
$ irb
irb(main):001:0> true && true
=> true
irb(main):002:0> 1 == 1 && 2 == 2
=> true
irb(main):003:0>
```

Study Drills

1. There are a lot of operators in Ruby similar to != and ==. Try to find as many "equality operators" as you can. They should be like < or <=.
2. Write out the names of each of these equality operators. For example, I call != "not equal."
3. Play with the Ruby by typing out new Boolean operators, and before you press Enter try to shout out what it is. Do not think about it. Shout the first thing that comes to mind. Write it down, then press Enter, and keep track of how many you get right and wrong.
4. Throw away the piece of paper from 3 so you do not accidentally try to use it later.

Common Student Questions

Why does "test" && "test" return "test" or 1 && 1 return 1 instead of true?

Ruby and many languages like it return one of the operands to their Boolean expressions rather than just `true` or `false`. This means that if you did `false && 1` you get the first operand (`false`), but if you do `true && 1` you get the second (1). Play with this a bit.

Is there any difference between != and <?

Ruby has deprecated < in favor of !=, so use !=. Other than that, there should be no difference.

Isn't there a shortcut?

Yes. Any `&&` expression that has a `false` is immediately `false`, so you can stop there. Any `||` expression that has a `true` is immediately `true`, so you can stop there. But make sure that you can process the whole expression because later it becomes helpful.

This page intentionally left blank

What If

Here is the next script of Ruby you will enter, which introduces you to the `if`-statement. Type this in, make it run exactly right, and then we'll see if your practice has paid off.

ex29.rb

```
1  people = 20
2  cats = 30
3  dogs = 15
4
5
6  if people < cats
7      puts "Too many cats! The world is doomed!"
8  end
9
10 if people > cats
11    puts "Not many cats! The world is saved!"
12 end
13
14 if people < dogs
15    puts "The world is drooled on!"
16 end
17
18 if people > dogs
19    puts "The world is dry!"
20 end
21
22
23 dogs += 5
24
25 if people >= dogs
26    puts "People are greater than or equal to dogs."
27 end
28
29 if people <= dogs
30    puts "People are less than or equal to dogs."
31 end
32
33
34 if people == dogs
35    puts "People are dogs."
36 end
```

What You Should See

Exercise 29 Session

```
$ ruby ex29.rb
Too many cats! The world is doomed!
The world is dry!
People are greater than or equal to dogs.
People are less than or equal to dogs.
People are dogs.
```

Study Drills

In this Study Drill, try to guess what you think the `if`-statement is and what it does. Try to answer these questions in your own words before moving on to the next exercise:

1. What do you think the `if` does to the code under it?
2. Why does the code under the `if` need to be indented two spaces?
3. What happens if it isn't indented?
4. Can you put other Boolean expressions from Exercise 27 in the `if`-statement? Try it.
5. What happens if you change the initial values for `people`, `cats`, and `dogs`?

Common Student Question

What does `+=` mean?

The code `x += 1` is the same as doing `x = x + 1` but involves less typing. You can call this the "increment by" operator. The same goes for `-=` and many other expressions you'll learn later.

Else and If

In the last exercise you worked out some if-statement and then tried to guess what they are and how they work. Before you learn more I'll explain what everything is by answering the questions you had from Study Drills. You did the Study Drills, right?

1. What do you think the if does to the code under it? An if-statement creates what is called a "branch" in the code. It's kind of like those choose your own adventure books where you are asked to turn to one page if you make one choice, and another if you go a different direction. The if-statement tells your script, "If this Boolean expression is true, then run the code under it, otherwise skip it."
2. Why does the code under the if need to be indented two spaces? In Ruby you indent code under statements like if, else, and others so that other programmers know it is a "block" of code. Blocks can have other blocks in them and are ended with the word end. There are other ways to make a block of code, but for if-statements this is the way.
3. What happens if you don't end it with end? If you don't, then Ruby doesn't know where your if-statement ends and where others might begin, so it will give you a syntax error.
4. Can you put other Boolean expressions from Ex. 27 in the if-statement? Try it. Yes you can, and they can be as complex as you like, although really complex things generally are bad style.
5. What happens if you change the initial values for people, cats, and dogs? Because you are comparing numbers, if you change the numbers, different if-statements will evaluate to true and the blocks of code under them will run. Go back and put different numbers in and see if you can figure out in your head what blocks of code will run.

Compare my answers to your answers, and make sure you *really* understand the concept of a "block" of code. This is important for when you do the next exercise where you write all the parts of if-statements that you can use.

Type this one in and make it work, too.

ex30.rb

```
1 people = 30
2 cars = 40
3 trucks = 15
4
5
6 if cars > people
```

```
7      puts "We should take the cars."  
8  elsif cars < people  
9      puts "We should not take the cars."  
10 else  
11     puts "We can't decide."  
12 end  
13  
14 if trucks > cars  
15   puts "That's too many trucks."  
16 elsif trucks < cars  
17   puts "Maybe we could take the trucks."  
18 else  
19   puts "We still can't decide."  
20 end  
21  
22 if people > trucks  
23   puts "Alright, let's just take the trucks."  
24 else  
25   puts "Fine, let's stay home then."  
26 end
```

What You Should See

Exercise 30 Session

```
$ ruby ex30.rb  
We should take the cars.  
Maybe we could take the trucks.  
Alright, let's just take the trucks.
```

Study Drills

1. Try to guess what `elsif` and `else` are doing.
2. Change the numbers of `cars`, `people`, and `trucks` and then trace through each `if`-statement to see what will be printed.
3. Try some more complex Boolean expressions like `cars > people || trucks < cars`.
4. Above each line write an English description of what the line does.

Common Student Question

What happens if multiple `elsif` blocks are true?

Ruby starts and at the top runs the first block that is true, so it will run only the first one.

Making Decisions

In the first half of this book, you mostly just printed out things called functions, but everything was basically in a straight line. Your scripts ran starting at the top and went to the bottom where they ended. If you made a function you could run that function later, but it still didn't have the kind of branching you need to really make decisions. Now that you have `if`, `else`, and `elsif` you can start to make scripts that decide things.

In the last script you wrote out a simple set of tests asking some questions. In this script you will ask the user questions and make decisions based on their answers. Write this script, and then play with it quite a lot to figure it out.

ex31.rb

```
1 puts "You enter a dark room with two doors. Do you go through door #1 or door #2?"
2
3 print "> "
4 door = $stdin.gets.chomp
5
6 if door == "1"
7   puts "There's a giant bear here eating a cheese cake. What do you do?"
8   puts "1. Take the cake."
9   puts "2. Scream at the bear."
10
11 print "> "
12 bear = $stdin.gets.chomp
13
14 if bear == "1"
15   puts "The bear eats your face off. Good job!"
16 elsif bear == "2"
17   puts "The bear eats your legs off. Good job!"
18 else
19   puts "Well, doing %s is probably better. Bear runs away." % bear
20 end
21
22 elsif door == "2"
23   puts "You stare into the endless abyss at Cthulhu's retina."
24   puts "1. Blueberries."
25   puts "2. Yellow jacket clothespins."
26   puts "3. Understanding revolvers yelling melodies."
27
28 print "> "
29 insanity = $stdin.gets.chomp
30
31 if insanity == "1" || insanity == "2"
32   puts "Your body survives powered by a mind of jello. Good job!"
33 else
```

```
34      puts "The insanity rots your eyes into a pool of muck. Good job!"  
35  end  
36  
37  else  
38      puts "You stumble around and fall on a knife and die. Good job!"  
39  end
```

A key point here is that you are now putting the `if`-statements *inside* `if`-statements as code that can run. This is very powerful and can be used to create “nested” decisions, where one branch leads to another and another.

Make sure you understand this concept of `if`-statements inside `if`-statements. In fact, do the Study Drills to really nail it.

What You Should See

Here is me playing this little adventure game. I do not do so well.

Exercise 31 Session

```
$ ruby ex31.rb  
You enter a dark room with two doors. Do you go through door #1 or door #2?  
> 1  
There's a giant bear here eating a cheese cake. What do you do?  
1. Take the cake.  
2. Scream at the bear.  
> 2  
The bear eats your legs off. Good job!
```

Study Drills

1. Make new parts of the game and change which decisions people can make. Expand the game out as much as you can before it gets ridiculous.
2. Write a completely new game. Maybe you don’t like this one, so make your own. This is your computer, do what you want.

Common Student Questions

Can you replace `elsif` with a sequence of `if-else` combinations?

You can in some situations, but it depends on how each `if/else` is written. It also means that Ruby will check every `if-else` combination, rather than just the first false ones like it would with `if-elsif-else`. Try to make some of these to figure out the differences.

How do I tell if a number is within a range of numbers?

Use `||` (or) to test if the number is `x > 1` or `x < 10`. Also remember you can use `<=` (less-than-equal) and similar tests, too.

What if I wanted more options in the `if-elsif-else` blocks?

Add more `elsif` blocks for each possible choice.

This page intentionally left blank

Loops and Arrays

You should now be able to do some programs that are much more interesting. If you have been keeping up, you should realize that now you can combine all the other things you have learned with if-statements and Boolean expressions to make your programs do smart things.

However, programs also need to do repetitive things very quickly. We are going to use a for-loop in this exercise to build and print various arrays. When you do the exercise, you will start to figure out what they are. I won't tell you right now. You have to figure it out.

Before you can use a for-loop, you need a way to store the results of loops somewhere. The best way to do this is with arrays. Arrays are exactly what their name says: a container of things that are organized in order from first to last. It's not complicated; you just have to learn a new syntax. First, there's how you make arrays:

```
hairs = ['brown', 'blond', 'red']
eyes = ['brown', 'blue', 'green']
weights = [1, 2, 3, 4]
```

You start the array with the [(left bracket) which "opens" the array. Then you put each item you want in the array separated by commas, similar to function arguments. Lastly, end the array with a] (right bracket) to indicate that it's over. Ruby then takes this array and all its contents and assigns them to the variable.

WARNING! This is where things get tricky for people who can't code. Your brain has been taught that the world is flat. Remember in the last exercise where you put if-statements inside if-statements? That probably made your brain hurt because most people do not ponder how to "nest" things inside things. In programming nested structures are all over the place. You will find functions that call other functions that have if-statements that have arrays with arrays inside arrays. If you see a structure like this that you can't figure out, take out a pencil and paper and break it down manually bit by bit until you understand it.

We now will build some arrays using some for-loops and print them out:

ex32.rb

```
1 the_count = [1, 2, 3, 4, 5]
2 fruits = ['apples', 'oranges', 'pears', 'apricots']
3 change = [1, 'pennies', 2, 'dimes', 3, 'quarters']
4
```

```
5 # this first kind of for-loop goes through a list
6 # in a more traditional style found in other languages
7 for number in the_count
8   puts "This is count #{number}"
9 end
10
11 # same as above, but in a more Ruby style
12 # this and the next one are the preferred
13 # way Ruby for-loops are written
14 fruits.each do |fruit|
15   puts "A fruit of type: #{fruit}"
16 end
17
18 # also we can go through mixed lists too
19 # note this is yet another style, exactly like above
20 # but a different syntax (way to write it).
21 change.each { |i| puts "I got #{i}" }
22
23 # we can also build lists, first start with an empty one
24 elements = []
25
26 # then use the range operator to do 0 to 5 counts
27 (0..5).each do |i|
28   puts "adding #{i} to the list."
29   # pushes the i variable on the *end* of the list
30   elements.push(i)
31 end
32
33 # now we can print them out too
34 elements.each { |i| puts "Element was: #{i}" }
```

You should immediately see that Ruby has two kinds of loops that I am calling a **for-loop**. In programming the term **for-loop** just means “a loop that goes through each thing in an array of things.” In Ruby this is both `for number in the_count` style, and the more common `fruits.each` style. You should use the `.each` version as it is more reliable and what other Ruby programmers expect you to write.

WARNING! Ruby programmers are very particular about how their **for-loops** are written and will declare you a bad programmer for simply using this one construct wrong. They went so far as to break the **for-each** version of looping so that there are problems with using it, forcing you to conform to their culture. Heed my warning that you should *always* use `.each` and never `for-each` for fear of being forever branded bad and shunned. Yes, it is as ridiculous as it sounds.

What You Should See

Exercise 32 Session

```
$ ruby ex32.rb
This is count 1
This is count 2
This is count 3
This is count 4
This is count 5
A fruit of type: apples
A fruit of type: oranges
A fruit of type: pears
A fruit of type: apricots
I got 1
I got pennies
I got 2
I got dimes
I got 3
I got quarters
adding 0 to the list.
adding 1 to the list.
adding 2 to the list.
adding 3 to the list.
adding 4 to the list.
adding 5 to the list.
Element was: 0
Element was: 1
Element was: 2
Element was: 3
Element was: 4
Element was: 5
```

Study Drills

1. Take a look at how you used `(0..5)` in the last `for`-loop. Look up Ruby's "range operator" `(.. and ...)` online to see what it does.
2. Change the first `for` number in `the_count` to be a more typical `.each` style loop like the others.
3. Find the Ruby documentation on arrays and read about them. What other operations can you do besides the `push` function? Try `<<`, which is the same as `push` but is an operator. `fruits << x` is the same as `fruits.push(x)`.

Common Student Questions

How do you make a 2-dimensional (2D) array?

That's an array in an array like this: `[[1,2,3],[4,5,6]]`

Aren't lists and arrays the same thing?

It depends on the language and the implementation. In classic terms a list is very different from an array because of how they're implemented. In Ruby they call these arrays. In Python they call them lists. Just call these arrays for now since that's what Ruby calls them.

Why is a for-loop able to use a variable that isn't defined yet?

The variable is defined by the for-loop when it starts, initializing it to the current element of the loop iteration, each time through.

While Loops

Now to totally blow your mind with a new loop, the while-loop. A while-loop will keep executing the code block under it as long as a Boolean expression is true.

Wait, you have been keeping up with the terminology, right? You should know that Ruby has three kinds of code blocks you need to read. You have the kind that if-statements use, where code is started after the if, and the block of code is ended with end. You then have two kinds for .each style blocks. Either you use do ... end or { ... } when making a block of code. When I use the characters ... in that last description I do not mean type ... I mean them in the normal English way of "and then some stuff here". If you don't quite understand this go back and redo the last few exercises until you get it.

Later on we'll have some exercises that will train your brain to read these structures, similar to how we burned Boolean expressions into your brain.

Back to while-loops. What they do is simply a test like an if-statement, but instead of running the code block once, they jump back to the "top" where the while is, and repeat. A while-loop runs until the expression is false.

Here's the problem with while-loops: Sometimes they do not stop. This is great if your intention is to just keep looping until the end of the universe. Otherwise you almost always want your loops to end eventually.

To avoid these problems, there are some rules to follow:

1. Make sure that you use while-loops sparingly. Usually a for-loop is better.
2. Review your while statements and make sure that the Boolean test will become false at some point.
3. When in doubt, print out your test variable at the top and bottom of the while-loop to see what it's doing.

In this exercise, you will learn the while-loop while doing these three checks:

ex33.rb

```
1  i = 0
2  numbers = []
3
4  while i < 6
5    puts "At the top i is #{i}"
6    numbers.push(i)
```

```
7      i += 1
8      puts "Numbers now: ", numbers
9      puts "At the bottom i is #{i}"
10     end
11
12
13    puts "The numbers: "
14
15    # remember you can write this 2 other ways?
16    numbers.each { |num| puts num }
```

What You Should See

Exercise 33 Session

```
$ ruby ex33.rb
At the top i is 0
Numbers now:
0
At the bottom i is 1
At the top i is 1
Numbers now:
0
1
At the bottom i is 2
At the top i is 2
Numbers now:
0
1
2
At the bottom i is 3
At the top i is 3
Numbers now:
0
1
2
3
At the bottom i is 4
At the top i is 4
Numbers now:
0
1
2
3
4
At the bottom i is 5
At the top i is 5
Numbers now:
0
1
2
```

```
3
4
5
At the bottom i is 6
The numbers:
0
1
2
3
4
5
6
```

Study Drills

1. Convert this while-loop to a function that you can call, and replace 6 in the test ($i < 6$) with a variable.
2. Use this function to rewrite the script to try different numbers.
3. Add another variable to the function arguments that you can pass in that lets you change the `+ 1` on line 8 so you can change the size of the increment.
4. Rewrite the script again to use this function to see what effect that has.
5. Write it to use for-loops and the `(0 .. 6)` range operator. Do you need the incrementor in the middle anymore? What happens if you do not get rid of it?

If at any time that you are doing this it goes crazy (it probably will), just hold down CTRL and press c (CTRL-c) and the program will abort.

Common Student Questions

What's the difference between a for-loop and a while-loop?

A for-loop can only iterate (loop) "over" collections of things. A while-loop can do any kind of iteration (looping) you want. However, while-loops are harder to get right and you normally can get many things done with for-loops.

Loops are hard. How do I figure them out?

The main reason people don't understand loops is because they can't follow the "jumping" that the code does. When a loop runs, it goes through its block of code, and at the end it jumps back to the top. To visualize this, put `puts` statements all over the loop, printing out where in the loop Ruby is running and what the variables are set to at those points. Write `puts` lines before the loop, at the top of the loop, in the middle, and at the bottom. Study the output and try to understand the jumping that's going on.

This page intentionally left blank

Accessing Elements of Arrays

Arrays are pretty useful, but unless you can get at the things in them, they aren't all that useful. You can already go through the elements of an array in order, but what if you want the fifth element? You need to know how to access the elements of an array. Here's how you would access the *first* element of an array:

```
animals = ['bear', 'tiger', 'penguin', 'zebra']
bear = animals[0]
```

You take an array of animals, and then you get the first (1st) one using 0?! How does that work? Because of the way math works, Ruby starts its arrays at 0 rather than 1. It seems weird, but there are many advantages to this, even though it is mostly arbitrary.

The best way to explain why is by showing you the difference between how you use numbers and how programmers use numbers.

Imagine you are watching the four animals in our array (`['bear', 'tiger', 'penguin', 'zebra']`) run in a race. They cross the finish line in the *order* we have them in this array. The race was really exciting because the animals didn't eat each other and somehow managed to run a race. Your friend shows up late and wants to know who won. Does your friend say, "Hey, who came in *zeroth*?" No, he says, "Hey, who came in *first*?"

This is because the *order* of the animals is important. You can't have the second animal without the first (1st) animal, and you can't have the third without the second. It's also impossible to have a "zeroth" animal since zero means nothing. How can you have a nothing win a race? It just doesn't make sense. We call these kinds of numbers "ordinal" numbers, because they indicate an ordering of things.

Programmers, however, can't think this way because they can pick any element out of an array at any point. To programmers, the array of animals is more like a deck of cards. If they want the tiger, they grab it. If they want the zebra, they can take it, too. This need to pull elements out of arrays at random means that they need a way to indicate elements consistently by an address, or an "index," and the best way to do that is to start the indices at 0. Trust me on this: the math is way easier for these kinds of accesses. This kind of number is a "cardinal" number and means you can pick at random, so there needs to be a 0 element.

How does this help you work with arrays? Simple, every time you say to yourself, "I want the third animal," you translate this "ordinal" number to a "cardinal" number by subtracting 1. The "third" animal is at index 2 and is the penguin. You have to do this because you have spent your whole life using ordinal numbers, and now you have to think in cardinal. Just subtract 1 and you will be good.

Remember: ordinal == ordered, 1st; cardinal == cards at random, 0.

Let's practice this. Take this array of animals, and follow the exercises where I tell you to write down what animal you get for that ordinal or cardinal number. Remember if I say "first," "second," then I'm using ordinal, so subtract 1. If I give you cardinal (like "The animal at 1"), then use it directly.

```
animals = ['bear', 'ruby', 'peacock', 'kangaroo', 'whale', 'platypus']
```

1. The animal at 1.
2. The third (3rd) animal.
3. The first (1st) animal.
4. The animal at 3.
5. The fifth (5th) animal.
6. The animal at 2.
7. The sixth (6th) animal.
8. The animal at 4.

For each of these, write out a full sentence of the form: "The first (1st) animal is at 0 and is a bear." Then say it backwards: "The animal at 0 is the 1st animal and is a bear."

Use Ruby to check your answers.

Study Drills

1. With what you know of the difference between these types of numbers, can you explain why the year 2010 in "January 1, 2010," really is 2010 and not 2009? (Hint: you can't pick years at random.)
 2. Write some more arrays and work out similar indexes until you can translate them.
 3. Use Ruby to check your answers.
-

WARNING! Programmers will tell you to read this guy named "Dijkstra" on this subject. I recommend you avoid his writings on this unless you enjoy being yelled at by someone who stopped programming at the same time programming started.

Branches and Functions

You have learned if-statements, functions, and arrays. Now it's time to bend your mind.
Type this in, and see if you can figure out what it's doing.

ex35.rb

```
1  def gold_room
2      puts "This room is full of gold. How much do you take?"
3
4      print "> "
5      choice = $stdin.gets.chomp
6
7      # this line has a bug, so fix it
8      if choice.include? ("0") || choice.include? ("1")
9          how_much = choice.to_i
10     else
11         dead("Man, learn to type a number.")
12     end
13
14     if how_much < 50
15         puts "Nice, you're not greedy, you win!"
16         exit(0)
17     else
18         dead("You greedy bastard!")
19     end
20 end
21
22
23 def bear_room
24     puts "There is a bear here."
25     puts "The bear has a bunch of honey."
26     puts "The fat bear is in front of another door."
27     puts "How are you going to move the bear?"
28     bear_moved = false
29
30     while true
31         print "> "
32         choice = $stdin.gets.chomp
33
34         if choice == "take honey"
35             dead("The bear looks at you then slaps your face off.")
36         elsif choice == "taunt bear" && !bear_moved
37             puts "The bear has moved from the door. You can go through it now."
38             bear_moved = true
39     end
40 end
```

```
39      elsif choice == "taunt bear" && bear_moved
40          dead("The bear gets pissed off and chews your leg off.")
41      elsif choice == "open door" && bear_moved
42          gold_room
43      else
44          puts "I got no idea what that means."
45      end
46  end
47 end
48
49
50 def cthulhu_room
51     puts "Here you see the great evil Cthulhu."
52     puts "He, it, whatever stares at you and you go insane."
53     puts "Do you flee for your life or eat your head?"
54
55     print "> "
56     choice = $stdin.gets.chomp
57
58     if choice.include? "flee"
59         start
60     elsif choice.include? "head"
61         dead("Well that was tasty!")
62     else
63         cthulhu_room
64     end
65 end
66
67
68 def dead(why)
69     puts why, "Good job!"
70     exit(0)
71 end
72
73 def start
74     puts "You are in a dark room."
75     puts "There is a door to your right and left."
76     puts "Which one do you take?"
77
78     print "> "
79     choice = $stdin.gets.chomp
80
81     if choice == "left"
82         bear_room
83     elsif choice == "right"
84         cthulhu_room
85     else
86         dead("You stumble around the room until you starve.")
```

```
87      end  
88  end  
89  
90  start
```

What You Should See

Here's me playing the game:

Exercise 35 Session

```
$ ruby ex35.rb  
You are in a dark room.  
There is a door to your right and left.  
Which one do you take?  
> right  
Here you see the great evil Cthulhu.  
He, it, whatever stares at you and you go insane.  
Do you flee for your life or eat your head?  
> cry  
Here you see the great evil Cthulhu.  
He, it, whatever stares at you and you go insane.  
Do you flee for your life or eat your head?  
> flee  
You are in a dark room.  
There is a door to your right and left.  
Which one do you take?  
> left  
There is a bear here.  
The bear has a bunch of honey.  
The fat bear is in front of another door.  
How are you going to move the bear?  
> take honey  
The bear looks at you then slaps your face off.  
Good job!
```

Study Drills

1. Draw a map of the game and indicate how you flow through it.
2. Fix all of your mistakes, including spelling mistakes.
3. Write comments for the functions you do not understand.
4. Add more to the game. What can you do to both simplify and expand it?
5. The `gold_room` has a weird way of getting you to type a number. What are all the bugs in this way of doing it? Can you make it better than what I've written? Look at how `=~` works for clues.

Common Student Questions

Help! How does this program work!?

When you get stuck understanding a piece of code, simply write an English comment above every line explaining what that line does. Keep your comments short and similar to the code. Then either diagram how the code works or write a paragraph describing it. If you do that, you'll get it.

Why did you write `while true`?

That makes an infinite loop.

What does `exit(0)` do?

On many operating systems a program can abort with `exit(0)`, and the number passed in will indicate an error or not. If you do `exit(1)` then it will be an error, but `exit(0)` will be a good exit. The reason it's backward from normal Boolean logic (with `0==false` is that you can use different numbers to indicate different error results. You can do `exit(100)` for a different error result than `exit(2)` or `exit(1)`.

Designing and Debugging

Now that you know if-statements, I'm going to give you some rules for for-loops and while-loops that will keep you out of trouble. I'm also going to give you some tips on debugging so that you can figure out problems with your program. Finally, you will design a little game similar to the last exercise, but with a slight twist.

Rules for If-Statements

1. Every if-statement must have an else.
2. If this else should never run because it doesn't make sense, then you must use a die function in the else that prints out an error message and dies, just like we did in the last exercise. This will find *many* errors.
3. Never nest if-statements more than two deep and always try to do them one deep.
4. Treat if-statements like paragraphs, where each if-elsif-else grouping is like a set of sentences. Put blank lines before and after.
5. Your Boolean tests should be simple. If they are complex, move their calculations to variables earlier in your function and use a good name for the variable.

If you follow these simple rules, you will start writing better code than most programmers. Go back to the last exercise and see if I followed all of these rules. If not, fix my mistakes.

WARNING! Never be a slave to the rules in real life. For training purposes you need to follow these rules to make your mind strong, but in real life sometimes these rules are just stupid. If you think a rule is stupid, try not using it.

Rules for Loops

1. Use a while-loop only to loop forever, and that means probably never. This only applies to Ruby; other languages are different.
2. Use a for-loop for all other kinds of looping, especially if there is a fixed or limited number of things to loop over.

Tips for Debugging

1. Do not use a "debugger." A debugger is like doing a full-body scan on a sick person. You do not get any specific useful information, and you find a whole lot of information that doesn't help and is just confusing.
2. The best way to debug a program is to use `puts` to print out the values of variables at points in the program to see where they go wrong.
3. Make sure parts of your programs run as you work on them. Do not write massive files of code before you try to run them. Code a little, run a little, fix a little.

Homework

Now write a similar game to the one that I created in the last exercise. It can be any kind of game you want in the same flavor. Spend a week on it making it as interesting as possible. For Study Drills, use arrays, functions, and modules (remember those from Exercise 13?) as much as possible, and find as many new pieces of Ruby as you can to make the game work.

Before you start coding you must draw a map for your game. Create the rooms, monsters, and traps that the player must go through on paper before you code.

Once you have your map, try to code it. If you find problems with the map, then adjust it and make the code match.

The best way to work on a piece of software is in small chunks like this:

1. On a sheet of paper or an index card, write a list of tasks you need to complete to finish the software. This is your to do list.
2. Pick the easiest thing you can do from your list.
3. Write out English comments in your source file as a guide for how you would accomplish this task in your code.
4. Write some of the code under the English comments.
5. Quickly run your script to see if that code worked.
6. Keep working in a cycle of writing some code, running it to test it, and fixing it until it works.
7. Cross this task off your list, then pick your next easiest task and repeat.

This process will help you work on software in a methodical and consistent manner. As you work, update your list by removing tasks you don't really need and adding ones you do.

Symbol Review

It's time to review the symbols and Ruby words you know and try to pick up a few more for the next few lessons. I have written out all the Ruby symbols and keywords that are important to know.

In this lesson take each keyword and first try to write out what it does from memory. Next, search online for it and see what it really does. This may be difficult because some of these are difficult to search for, but try anyway.

If you get one of these wrong from memory, make an index card with the correct definition and try to "correct" your memory.

Finally, use each of these in a small Ruby program, or as many as you can get done. The goal is to find out what the symbol does, make sure you got it right, correct it if you do not, then use it to lock it in.

Keywords

Keyword	Description	Example
BEGIN	Run this block when the script starts.	BEGIN { puts "hi" }
END	Run this block when the script is done.	END { puts "hi" }
alias	Create another name for a function.	alias X Y
and	Logical and, but lower priority than &&.	put "Hello" and "Goodbye"
begin	Start a block, usually for exceptions.	begin end
break	Break out of a loop right now.	while true; break; end
case	Case style conditional, like an if.	case X; when Y; else; end
class	Define a new class.	class X; end
def	Define a new function.	def X(); end
defined?	Is this class/function/etc. defined already?	defined? Class == "constant"
do	Create a block that maybe takes a parameter.	(0..5).each do x puts x end
else	Else conditional.	if X; else; end

(Continues)

Keyword	Description	Example
<code>elsif</code>	Else if conditional	<code>if X; elsif Y; else; end</code>
<code>end</code>	Ends blocks, functions, classes, everything.	<code>begin end # many others</code>
<code>ensure</code>	Run this code whether an exception happens or not.	<code>begin ensure end</code>
<code>for</code>	For loop syntax. The <code>.each</code> syntax is preferred.	<code>for X in Y; end</code>
<code>if</code>	If conditional.	<code>if X; end</code>
<code>in</code>	In part of <code>for-loops</code> .	<code>for X in Y; end</code>
<code>module</code>	Define a new module.	<code>module X; end</code>
<code>next</code>	Skip to the next element of a <code>.each</code> iterator.	<code>(0..5).each { y next }</code>
<code>not</code>	Logical not. But use <code>!</code> instead.	<code>not true == false</code>
<code>or</code>	Logical or.	<code>puts "Hello" or "Goodbye"</code>
<code>redo</code>	Rerun a code block exactly the same.	<code>(0..5).each { i redo if i > 2 }</code>
<code>rescue</code>	Run this code if an exception happens.	<code>begin rescue X; end</code>
<code>retry</code>	In a rescue clause, says to try the block again.	<code>(0..5).each { i retry if i > 2 }</code>
<code>return</code>	Returns a value from a function. Mostly optional.	<code>return X</code>
<code>self</code>	The current object, class, or module.	<code>defined? self == "self"</code>
<code>super</code>	The parent class of this class.	<code>super</code>
<code>then</code>	Can be used with <code>if</code> optionally.	<code>if true then puts "hi" end</code>
<code>undef</code>	Remove a function definition from a class.	<code>undef X</code>
<code>unless</code>	Inverse of <code>if</code> .	<code>unless false then puts "not" end</code>
<code>until</code>	Inverse of <code>while</code> , execute block as long as <code>false</code> .	<code>until false; end</code>
<code>when</code>	Part of case conditionals.	<code>case X; when Y; else; end</code>
<code>while</code>	While loop.	<code>while true; end</code>
<code>yield</code>	Pause and transfer control to the code block.	<code>yield</code>

Data Types

For data types, write out what makes up each one. For example, with strings write out how you create a string. For numbers, write out a few numbers.

Type	Description	Example
true	True Boolean value.	true or false == true
false	False Boolean value.	false and true == false
nil	Represents "nothing" or "no value."	x = nil
strings	Stores textual information.	x = "hello"
numbers	Stores integers.	i = 100
floats	Stores decimals.	i = 10.389
arrays	Stores a list of things.	j = [1,2,3,4]
hashes	Stores a key=value mapping of things.	e = {'x' => 1, 'y' => 2}

String Escape Sequences

For string escape sequences, use them in strings to make sure they do what you think they do.

Escape	Description
\\	Backslash
\'	Single-quote
\"	Double-quote
\a	Bell
\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage
\t	Tab
\v	Vertical tab

Operators

Some of these may be unfamiliar to you, but look them up anyway. Find out what they do, and if you still can't figure it out, save it for later.

Operator	Description	Example
+	Add	2 + 4 == 6
-	Subtract	2 - 4 == 2
*	Multiply	2 * 4 == 8
**	Power of	2 ** 4 == 16
/	Divide	2 / 4.0 == 0.5
%	Modulus	2 % 4 == 2
.	Dot access	"1".to_i == 1
::	Colon access	Module::Class
[]	List brackets	[1,2,3,4]
!	Not	!true == false
<	Less than	4 < 4 == false
>	Greater than	4 > 4 == false
>=	Greater than equal	4 >= 4 == true
<=	Less than equal	4 <= 4 == true
<=>	Comparison	4 <=> 4 == 0
==	Equal	4 == 4 == true
====	Equality	4 === 4 == true
!=	Not equal	4 != 4 == false
&&	Logical and (higher precedence)	true && false == false
	Logical or (higher precedence)	true false == true
..	Range inclusive	(0..3).to_a == [0, 1, 2, 3]
...	Range non-inclusive	(0....3).to_a == [0, 1, 2]
@	Object scope	@var ; @@classvar
@@	Class scope	@var ; @@classvar
\$	Global scope	\$stdin

Spend about a week on this, but if you finish faster that's great. The point is to try to get coverage on all these symbols and make sure they are locked in your head. What's also important is to find out what you *do not* know so you can fix it later.

Reading Code

Now find some Ruby code to read. You should be reading any Ruby code you can and trying to steal ideas that you find. You actually should have enough knowledge to be able to read, but maybe not understand what the code does. What this lesson teaches is how to apply things you have learned to understand other people's code.

First, print out the code you want to understand. Yes, print it out, because your eyes and brain are more used to reading paper than computer screens. Make sure you print a few pages at a time.

Second, go through your printout and take notes of the following:

1. Functions and what they do.
2. Where each variable is first given a value.
3. Any variables with the same names in different parts of the program. These may be trouble later.
4. Any if-statements without else clauses. Are they right?
5. Any while-loops that might not end.
6. Any parts of code that you can't understand for whatever reason.

Third, once you have all of this marked up, try to explain it to yourself by writing comments as you go. Explain the functions, how they are used, what variables are involved, and anything you can to figure this code out.

Lastly, on all of the difficult parts, trace the values of each variable line by line, function by function. In fact, do another printout and write in the margin the value of each variable that you need to "trace."

Once you have a good idea of what the code does, go back to the computer and read it again to see if you find new things. Keep finding more code and doing this until you do not need the printouts anymore.

Study Drills

1. Find out what a "flow chart" is and draw a few.
2. If you find errors in code you are reading, try to fix them and send the author your changes.

3. Another technique for when you are not using paper is to put # comments with your notes in the code. Sometimes, these could become the actual comments to help the next person.

Common Student Question

How would I search for these things online?

Simply put "ruby" before anything you want to find. For example, to find yield search for ruby yield.

Doing Things to Arrays

You have learned about arrays. When you learned about while-loops you “appended” numbers to the end of an array and printed them out. There were also Study Drills where you were supposed to find all the other things you can do to arrays in the Ruby documentation. That was a while back, so review those topics if you do not know what I’m talking about.

Found it? Remember it? Good. When you did this you had an array, and you “called” the function push on it. However, you may not really understand what’s going on so let’s see what we can do to arrays.

When you write `mystuff.push('hello')`, you are actually setting off a chain of events inside Ruby to cause something to happen to the `mystuff` array. Here’s how it works:

1. Ruby sees you mentioned `mystuff` and looks up that variable. It might have to look backward to see if you created with `=`, if it is a function argument, or if it’s a global variable. Either way it has to find the `mystuff` first.
2. Once it finds `mystuff`, Ruby reads the `.` (period) operator and starts to look at *variables* that are a part of `mystuff`. Since `mystuff` is an array, it knows that `mystuff` has a bunch of functions.
3. It then hits `push` and compares the name to all the names that `mystuff` says it owns. If `push` is in there (it is), then Ruby grabs *that* to use.
4. Next Ruby sees the `(` (opening parenthesis) and realizes, “Oh hey, this should be a function.” At this point it *calls* (runs, executes) the function just like normally, but instead it calls the function with an *extra* argument.
5. That *extra* argument is ... `mystuff`! I know, weird, right? But that’s how Ruby works so it’s best to just remember it and assume that’s the result. What happens, at the end of all this, is a function call that looks like: `push(mystuff, 'hello')` instead of what you read which is `mystuff.append('hello')`.

This might be a lot to take in, but we’re going to spend a few exercises getting this concept firm in your brain. To kick things off, here’s an exercise that mixes strings and arrays for all kinds of fun.

`ex38.rb`

```
1  ten_things = "Apples Oranges Crows Telephone Light Sugar"
2
3  puts "Wait there are not 10 things in that list. Let's fix that."
4
5  stuff = ten_things.split(' ')
```

```
6 more_stuff = ["Day", "Night", "Song", "Frisbee", "Corn", "Banana", "Girl", "Boy"]
7 # using math to make sure there's 10 items
8
9
10 while stuff.length != 10
11   next_one = more_stuff.pop
12   puts "Adding: #{next_one}"
13   stuff.push(next_one)
14   puts "There are #{stuff.length} items now."
15 end
16
17 puts "There we go: #{stuff}"
18
19 puts "Let's do some things with stuff."
20
21 puts stuff[1]
22 puts stuff[-1] # whoa! fancy
23 puts stuff.pop()
24 puts stuff.join(' ')
25 puts stuff[3..5].join("#")
```

What You Should See

Exercise 38 Session

```
$ ruby ex38.rb
Wait there are not 10 things in that list. Let's fix that.
Adding: Boy
There are 7 items now.
Adding: Girl
There are 8 items now.
Adding: Banana
There are 9 items now.
Adding: Corn
There are 10 items now.
There we go: ["Apples", "Oranges", "Crows", "Telephone", "Light", "Sugar",
             "Boy", "Girl", "Banana", "Corn"]
Let's do some things with stuff.
Oranges
Corn
Corn
Apples Oranges Crows Telephone Light Sugar Boy Girl Banana
Telephone#Light#Sugar
```

What Arrays Can Do

Let's say you want to create a computer game based on Go Fish. If you don't know what Go Fish is, take the time now to go read up on it on the Internet. To do this you would need to have some way of taking the concept of a "deck of cards" and put it into your Ruby program. You then have to write Ruby code that knows how to work this imaginary version of a deck of cards so that a person playing your game thinks that it's real, even if it isn't. What you need is a "deck of cards" structure, and programmers call this kind of thing a "data structure."

What's a data structure? If you think about it, a "data structure" is just a formal way to *structure* (organize) some *data* (facts). It really is that simple, even though some data structures can get insanely complex, all they are is just a way to store facts inside a program so you can access them in different ways. They structure data.

I'll be getting into this more in the next exercise, but arrays are one of the most common data structures programmers use. They are simply ordered lists of facts you want to store and access randomly or linearly by an index. What?! Remember what I said though, just because a programmer said "array is a list" doesn't mean that it's any more complex than what an array already is in the real world. Let's look at the deck of cards as an example of an array:

1. You have a bunch of cards with values.
2. Those cards are in a stack, list, or array from the top card to the bottom card.
3. You can take cards off the top, the bottom, the middle at random.
4. If you want to find a specific card, you have to grab the deck and go through it one at a time.

Let's look at what I said:

"An ordered array" Yes, a deck of cards is in order with a first card and a last card.

"of things you want to store" Yes, cards are things I want to store.

"and access randomly" Yes, I can grab a card from anywhere in the deck.

"or linearly" Yes, if I want to find a specific card I can start at the beginning and go in order.

"by an index" Almost, since with a deck of cards if I told you to get the card at index 19 you'd have to count until you found that one. In our Ruby arrays the computer can just jump right to any index you give it.

That is all an array does, and this should give you a way to figure out concepts in programming. Every concept in programming usually has some relationship to the real world. At least the useful ones do. If you can figure out what the analog in the real world is, then you can use that to figure out what the data structure should be able to do.

When to Use Arrays

You use an array whenever you have something that matches the array data structure's useful features:

1. If you need to maintain order. Remember, this is listed order, not *sorted* order. Arrays do not sort for you.

2. If you need to access the contents randomly by a number. Remember, this is using *cardinal* numbers starting at 0.
3. If you need to go through the contents linearly (first to last). Remember, that's what `for`-loops are for.

Then that's when you use an array.

Study Drills

1. Take each function that is called, and go through the steps for function calls to translate them to what Ruby does. For example, `more_stuff.pop()` is `pop(more_stuff)`.
2. Translate these two ways to view the function calls in English. For example, `more_stuff.pop()` reads as, "Call `pop` on `more_stuff`." Meanwhile, `pop(more_stuff)` means, "Call `pop` with argument `more_stuff`." Understand how they are really the same thing.
3. Go read about "object-oriented programming" online. Confused? I was too. Do not worry. You will learn enough to be dangerous, and you can slowly learn more later.
4. Read up on what a "class" is in Ruby. *Do not read about how other languages use the word "class."* That will only mess you up.
5. Do not worry If you do not have any idea what I'm talking about. Programmers like to feel smart so they invented object-oriented programming, named it OOP, and then used it way too much. If you think that's hard, you should try to use "functional programming."
6. Find 10 examples of things in the real world that would fit in an array. Try writing some scripts to work with them.

Common Student Questions

Why did you use a `while`-loop?

Try rewriting it with a `for`-loop and see if that's easier.

What does `stuff[3..5]` do?

That extracts a "slice" from the `stuff` array that is from element 3 to element 4, meaning it does *not* include element 5. It's similar to how `(3..5)` would work.

Hashes, Oh Lovely Hashes

Now I have to hurt you with another container you can use, because once you learn this container a massive world of ultra-cool will be yours. It is the most useful container ever: the dictionary.

Ruby calls them "hashes", which is short for "hashmaps." Other languages call them "dictionaries." I tend to use both names, but it doesn't matter. What does matter is what hashes do when compared to arrays. You see, an array lets you do this:

Exercise 39 Ruby Session

```
?> things = ['a', 'b', 'c', 'd']
=> ["a", "b", "c", "d"]
>> puts things[1]
b
=> nil
>> things[1] = 'z'
=> "z"
>> puts things[1]
z
=> nil
>> things
=> ["a", "z", "c", "d"]
```

You can use numbers to "index" into a array, meaning you can use numbers to find out what's in arrays. You should know this about arrays by now, but make sure you understand that you can *only* use numbers to get items out of a array.

What a Hash does is let you use *anything*, not just numbers as your index. Yes, a Hash associates one thing to another, no matter what it is. Take a look:

Exercise 39 Ruby Session

```
?> stuff = {'name' => 'Zed', 'age' => 39, 'height' => 6 * 12 + 2}
=> {"name"=>"Zed", "age"=>39, "height"=>74}
>> puts stuff['name']
Zed
=> nil
>> puts stuff['age']
39
=> nil
>> puts stuff['height']
74
=> nil
>> stuff['city'] = "San Francisco"
```

```
=> "San Francisco"
=> print stuff['city']
San Francisco=> nil
```

You will see that instead of just numbers we're using strings to say what we want from the stuff hash. We can also put new things into the hash with strings. You aren't limited to strings for keys or values. We can also do this:

Exercise 39 Ruby Session

```
?> stuff[1] = "Wow"
=> "Wow"
=> stuff[2] = "Neato"
=> "Neato"
=> puts stuff[1]
Wow
=> nil
=> puts stuff[2]
Neato
=> nil
=> stuff
=> {"name"=>"Zed", "age"=>39, "height"=>74,
     "city"=>"San Francisco", 1=>"Wow", 2=>"Neato"}
```

In this code I used numbers, and you can see numbers and strings as keys in the hash when I print it. I could use anything. Well, almost anything but just pretend you can use anything for now.

Of course, a hash that you can only put things in is pretty stupid, so here's how you delete things, with the `delete` function:

Exercise 39 Ruby Session

```
?> stuff.delete('city')
=> "San Francisco"
=> stuff.delete(1)
=> "Wow"
=> stuff.delete(2)
=> "Neato"
=> stuff
=> {"name"=>"Zed", "age"=>39, "height"=>74}
```

A Hash Example

We'll now do an exercise that you *must* study very carefully. I want you to type this code in and try to understand what's going on. Take note of when you put things in a hash, get things from a hash, and all the operations you use. Notice how this example is mapping states to their abbreviations, and then the abbreviations to cities in the states. Remember, "mapping" or "associating" is the key concept in a hash.

ex39.rb

```
1 # create a mapping of state to abbreviation
2 states = {
3     'Oregon' => 'OR',
4     'Florida' => 'FL',
5     'California' => 'CA',
6     'New York' => 'NY',
7     'Michigan' => 'MI'
8 }
9
10 # create a basic set of states and some cities in them
11 cities = {
12     'CA' => 'San Francisco',
13     'MI' => 'Detroit',
14     'FL' => 'Jacksonville'
15 }
16
17 # add some more cities
18 cities['NY'] = 'New York'
19 cities['OR'] = 'Portland'
20
21 # puts out some cities
22 puts '-' * 10
23 puts "NY State has: #{cities['NY']}"
24 puts "OR State has: #{cities['OR']}"
25
26 # puts some states
27 puts '-' * 10
28 puts "Michigan's abbreviation is: #{states['Michigan']}"
29 puts "Florida's abbreviation is: #{states['Florida']}"
30
31 # do it by using the state then cities dict
32 puts '-' * 10
33 puts "Michigan has: #{cities[states['Michigan']]}"
34 puts "Florida has: #{cities[states['Florida']]}"
35
36 # puts every state abbreviation
37 puts '-' * 10
38 states.each do |state, abbrev|
39     puts "#{state} is abbreviated #{abbrev}"
40 end
41
42 # puts every city in state
43 puts '-' * 10
44 cities.each do |abbrev, city|
45     puts "#{abbrev} has the city #{city}"
46 end
```

```
47
48 # now do both at the same time
49 puts '-' * 10
50 states.each do |state, abbrev|
51   city = cities[abbrev]
52   puts "#{state} is abbreviated #{abbrev} and has city #{city}"
53 end
54
55 puts '-' * 10
56 # by default ruby says "nil" when something isn't in there
57 state = states['Texas']
58
59 if !state
60   puts "Sorry, no Texas."
61 end
62
63 # default values using ||= with the nil result
64 city = cities['TX']
65 city ||= 'Does Not Exist'
66 puts "The city for the state 'TX' is: #{city}"
```

What You Should See

Exercise 39 Session

```
$ ruby ex39.rb
-----
NY State has: New York
OR State has: Portland
-----
Michigan's abbreviation is: MI
Florida's abbreviation is: FL
-----
Michigan has: Detroit
Florida has: Jacksonville
-----
Oregon is abbreviated OR
Florida is abbreviated FL
California is abbreviated CA
New York is abbreviated NY
Michigan is abbreviated MI
-----
CA has the city San Francisco
MI has the city Detroit
FL has the city Jacksonville
NY has the city New York
OR has the city Portland
```

```
-----  
Oregon is abbreviated OR and has city Portland  
Florida is abbreviated FL and has city Jacksonville  
California is abbreviated CA and has city San Francisco  
New York is abbreviated NY and has city New York  
Michigan is abbreviated MI and has city Detroit  
-----  
Sorry, no Texas.  
The city for the state 'TX' is: Does Not Exist
```

What Hashes Can Do

Hashes are another example of a data structure, and like arrays they are one of the most commonly used data structures in programming. A hash is used to *map* or *associate* things you want to store to keys you need to get them.

Let's say you want to find out what the word "honorificabilitudinitatibus" means. Today you would simply copy-paste that word into a search engine and then find out the answer, and we could say a search engine is like a really huge super complex version of the *Oxford English Dictionary* (OED). Before search engines what you would do is this:

1. Go to your library and get "the dictionary." Let's say it's the OED.
2. You know "honorificabilitudinitatibus" starts with the letter 'h' so you look on the side of the book for the little tab that has 'H' on it.
3. Then you'd skim the pages until you are close to where "hon" started.
4. Then you'd skim a few more pages until you found "honorificabilitudinitatibus" or hit the beginning of the "hp" words and realize this word isn't in the OED.
5. Once you found the entry, you'd read the definition to figure out what it means.

This process is nearly exactly the way a hash works, and you are basically "mapping" the word "honorificabilitudinitatibus" to its definition. A hash in Ruby is just like a dictionary in the real world like the OED.

In fact, we can implement a hash like this using just arrays.

Making Your Own Hash Module

The final piece of code in this exercise will show you how to make a hash data structure using nothing but the arrays you already know how to use. This code may be a little difficult to understand, so don't worry if it takes you a while to fully grasp what it's doing. There are no new concepts in this code. It is slightly more complex and has a few more things you will need to look up.

To keep things from stepping on Ruby's hash I'm going to call my data structure a Dict, which is another name for a hash data structure since it maps one thing to another. You should type this code into a file named dict.rb so we can run it in another file named ex39_test.rb next.

dict.rb

```
1  module Dict
2      def Dict.new(num_buckets=256)
3          # Initializes a Dict with the given number of buckets.
4          aDict = []
5          (0...num_buckets).each do |i|
6              aDict.push([])
7          end
8
9          return aDict
10     end
11
12     def Dict.hash_key(aDict, key)
13         # Given a key this will create a number and then convert it to
14         # an index for the aDict's buckets.
15         return key.hash % aDict.length
16     end
17
18     def Dict.get_bucket(aDict, key)
19         # Given a key, find the bucket where it would go.
20         bucket_id = Dict.hash_key(aDict, key)
21         return aDict[bucket_id]
22     end
23
24     def Dict.get_slot(aDict, key, default=nil)
25         # Returns the index, key, and value of a slot found in a bucket.
26         bucket = Dict.get_bucket(aDict, key)
27
28         bucket.each_with_index do |kv, i|
29             k, v = kv
30             if key == k
31                 return i, k, v
32             end
33         end
34
35         return -1, key, default
36     end
37
38     def Dict.get(aDict, key, default=nil)
39         # Gets the value in a bucket for the given key, or the default.
40         i, k, v = Dict.get_slot(aDict, key, default=default)
41         return v
42     end
43
```

```

44  def Dict.set(aDict, key, value)
45      # Sets the key to the value, replacing any existing value.
46      bucket = Dict.get_bucket(aDict, key)
47      i, k, v = Dict.get_slot(aDict, key)
48
49      if i >= 0
50          bucket[i] = [key, value]
51      else
52          bucket.push([key, value])
53      end
54  end
55
56  def Dict.delete(aDict, key)
57      # Deletes the given key from the Dict.
58      bucket = Dict.get_bucket(aDict, key)
59
60      (0...bucket.length).each do |i|
61          k, v = bucket[i]
62          if key == k
63              bucket.delete_at(i)
64              break
65          end
66      end
67  end
68
69  def Dict.list(aDict)
70      # Prints out what's in the Dict.
71      aDict.each do |bucket|
72          if bucket
73              bucket.each { |k, v| puts k, v}
74          end
75      end
76  end
77 end

```

This will create a module named `Dict`, which you should be able to import and work with using this `ex39_test.rb` file:

ex39_test.rb

```

1  require './dict.rb'
2
3  # create a mapping of state to abbreviation
4  states = Dict.new()
5  Dict.set(states, 'Oregon', 'OR')
6  Dict.set(states, 'Florida', 'FL')
7  Dict.set(states, 'California', 'CA')
8  Dict.set(states, 'New York', 'NY')
9  Dict.set(states, 'Michigan', 'MI')
10

```

```
11 # create a basic set of states and some cities in them
12 cities = Dict.new()
13 Dict.set(cities, 'CA', 'San Francisco')
14 Dict.set(cities, 'MI', 'Detroit')
15 Dict.set(cities, 'FL', 'Jacksonville')
16
17 # add some more cities
18 Dict.set(cities, 'NY', 'New York')
19 Dict.set(cities, 'OR', 'Portland')
20
21
22 # puts out some cities
23 puts '-' * 10
24 puts "NY State has: #{Dict.get(cities, 'NY')}"
25 puts "OR State has: #{Dict.get(cities, 'OR')}"
26
27 # puts some states
28 puts '-' * 10
29 puts "Michigan's abbreviation is: #{Dict.get(states, 'Michigan')}"
30 puts "Florida's abbreviation is: #{Dict.get(states, 'Florida')}"
31
32 # do it by using the state then cities dict
33 puts '-' * 10
34 puts "Michigan has: #{Dict.get(cities, Dict.get(states, 'Michigan'))}"
35 puts "Florida has: #{Dict.get(cities, Dict.get(states, 'Florida'))}"
36
37 # puts every state abbreviation
38 puts '-' * 10
39 Dict.list(states)
40
41 # puts every city in state
42 puts '-' * 10
43 Dict.list(cities)
44
45 puts '-' * 10
46 # by default ruby says "nil" when something isn't in there
47 state = Dict.get(states, 'Texas')
48
49 if !state
50   puts "Sorry, no Texas."
51 end
52
53 # default values using ||= with the nil result
54 city = Dict.get(cities, 'TX', 'Does Not Exist')
55 puts "The city for the state 'TX' is: #{city}"
```

You should recognize that this is nearly the same code as we created at the top, except that it's using your new implementation of `Dict` instead. Go through and confirm you understand how each line is the same operation as the same line in `ex39.rb`.

The Code Description

This Dict is nothing more than "an array of buckets, which are an array of slots, which have key/value pairs in them." Take a minute to break that down to see what I mean:

"an array of buckets..." In the Dict function I create the aDict variable which is an array, and then I fill it with other arrays

"which are an array of slots..." These bucket arrays are empty at first, but as we add key/value pairs to the data structure they will fill with "slots" or

"which have key/value pairs in them." Meaning each slot inside a bucket contains a (key, value) tuple or "pair."

If this structure still doesn't make sense, take the time to draw it out on paper until you're sure you get it. In fact, doing algorithms manually on paper is a good way to make sure you understand them.

Now that you know how the data is structured, you just need to know the *algorithm* for each operation. An algorithm is the steps you take to do something to or with a data structure. It's the code that makes the data structure work. We'll go through each of these operations using the code, but a general pattern in the Dict algorithms is this:

1. Convert a key to an integer using a hashing function: hash_key.
2. Convert this hash to a bucket number using the % (modulus) operator.
3. Get this bucket from the aDict array of buckets, and then traverse it to find the slot that contains the key we want.

In the case of set we do this, to replace duplicate keys, or we append to add new ones.

I will now walk through the code for the Dict so you can understand what's going on, following function by function. Follow along and make sure you understand every line. Write an English comment above each line to make sure you understand what it's doing. This is so deceptively simple that I recommend you take the time to play with any lines of code mentioned in the following description either in the Ruby shell, or on paper until you get it.

new First I start by creating a function that makes a Dict for you, also known as an initializer. What I've done is created an aDict variable that has an array, and then I put num_buckets arrays inside it. These buckets will be used to hold the contents of the Dict as I set them. Later I use aDict.length in other functions to find out how many buckets there are. Be sure you understand that!

hash_key This deceptively simple function is the core of how a hash works. What it does is uses the built-in Ruby hash function to convert a string to a number. Ruby uses this function for its own hash data structure, and I'm just reusing it. You should fire up a Ruby console to see how it works. Once I have a number for the key, I then use the % (modulus) operator and the `aDict.length` to get a bucket where this key can go. As you should know, the % (modulus) operator will divide any number and give me the remainder. I can also use this as a way of limiting giant numbers to a fixed smaller set of other numbers. If you don't get this, then use Ruby to explore it.

get_bucket This function then uses `hash_key` to find the bucket that a key *could* be in. Since I did `% aDict.length` in the `hash_key` function, I know whatever `bucket_id` I get will fit into the `aDict` array. Using the `bucket_id` I can then get the bucket where the key *could* be.

get_slot This function uses `get_bucket` to get the bucket a key could be in, and then it simply rolls through every element of that bucket until it finds a matching key. Once it does it returns the tuple of `(i, k, v)` which is the index the key was found in, the key itself, and the value set for that key. You now know enough to see how this data structure works. It takes keys, hashes and uses the modulus operator on them to find a bucket, and then searches that bucket to find the item. This effectively cuts the amount of searching necessary to a fraction of what it would be normally.

get This is a "convenience function" which does what most people want a `Dict` to do. It uses `get_slot` to get the `(i, k, v)` and then just returns the `v` (value) only. Make sure you understand how the `default` variable works, and how the `(i, k, v)` in `get_slot` is assigned to the `i, k, v` variables in `get`.

set To set a key/value pair all I need to do is get the bucket, and append the new (key, value) to it so it can be found later. However, I want my `Dict` to allow one key at a time. To do that, first I have to find the bucket then check if this key already exists. If it does then I replace it in the bucket with the new one, and if it doesn't then I append. This is actually slower than simply appending, but more likely what a user of `Dict` wants. If you wanted to allow multiple values for a key you could simply have `get` go through every slot in the bucket and return an array of everything it found. This is a good example of tradeoffs in design. The current version is faster on `get`, but slower on `set`.

delete To delete a key, I simply get the bucket and search for the key in it, and delete it from the array. However, because I chose to make `set` store only one key/value pair I can stop when I have found one. If I had decided to allow multiple values for each key by simply appending I would have also made `delete` slower because I would have needed to go through every slot on `delete` just in case it had a key/value pair that matched.

list The last function is simply a little debug function that prints out what's in the Dict and should be trivial for you to understand. It just gets each bucket, then goes through each slot in the bucket.

After all of these functions I just have a little bit of testing code that makes sure they work.

Three Levels of Arrays

As I mentioned in the discussion, by deciding that set will overwrite (replace) keys with new values, I have made set slower but this assumption makes all of the other functions faster. What if I wanted a Dict that allowed multiple values for each key but still keep everything fast?

What I need to do then is establish that every slot in a bucket is an array of values. This means that I add a third level of buckets, then slots, then values. This also matches the description of this kind of Dict because I am saying, "For every key there can be *multiple* values." Another way to say that is "Every key has an array of values." Since keys go in slots, then slots have arrays of values and I'm done.

If you want to take this code further, then change it to support multiple values for each key.

What You Should See (Again)

Exercise 39_dict Session

```
$ ruby ex39_test.rb
-----
NY State has: New York
OR State has: Portland
-----
Michigan's abbreviation is: MI
Florida's abbreviation is: FL
-----
Michigan has: Detroit
Florida has: Jacksonville
-----
Florida
FL
California
CA
Michigan
MI
New York
NY
Oregon
```

```
OR
-----
OR
Portland
CA
San Francisco
MI
Detroit
FL
Jacksonville
NY
New York
-----
Sorry, no Texas.
The city for the state 'TX' is: Does Not Exist
```

When to Use Hashes or Arrays

As I mentioned in Exercise 38, arrays have specific properties that help you contain and organize things that need to be in an array structure. Hashes are the same, but the properties of a hash are different than arrays because they work with mapping keys to values. That means you use a hash when:

1. You have to retrieve things based on some identifier, like names, addresses, or anything that can be a key.
2. You don't need things to be in order. Hashes do not normally have any notion of order, so you have to use an array for that.
3. You are going to be adding and removing elements and their keys.

This means that if you have to use a non-numeric key, use a hash. If you need things in order, use an array.

Study Drills

1. Do this same kind of mapping with cities and states/regions in your country or some other country.
2. Find the Ruby documentation for hashes and try to do even more things to them.
3. Find out what you *can't* do with hashes. A big one is that they do not have order, so try playing with that.

4. Read about Ruby's assert feature and then take the Dict code and add assertions for each of the tests I've done instead of print. For example, you can assert that the first get operation returns "*Flamenco Sketches*" instead of just printing that out.
5. Did you notice that the `list` function listed the items I added in a *different* order than they were added? This is an example of how hashes don't maintain order, and if you analyze the code you'll understand why.
6. Create a dump function that is like `list` but which dumps the full contents of every bucket so you can debug it.
7. Make sure you know what the hash function does in that code. It's a special function that converts strings to a consistent integer. Find the documentation for it online. Read about what a "hash function" is in programming.

Common Student Questions

What is the difference between an array and a hash?

An array is for an ordered array of items. A hash (or hash) is for matching some items (called "keys") to other items (called "values").

What would I use a hash for?

When you have to take one value and "look up" another value. In fact you could call hashes "look up tables."

What would I use an array for?

Use an array for any sequence of things that need to be in order, and you only need to look them up by a numeric index.

This page intentionally left blank

Modules, Classes, and Objects

Ruby is called an “object-oriented programming language.” This means there is a construct in Ruby called a class that lets you structure your software in a particular way. Using classes, you can add consistency to your programs so that they can be used in a cleaner way. At least that’s the theory.

I am now going to teach you the beginnings of object-oriented programming, classes, and objects using what you already know about hashes and modules. My problem is that Object-Oriented Programming (OOP) is just plain weird. You have to struggle with this, try to understand what I say here, type in the code, and in the next exercise I’ll hammer it in.

Here we go.

Modules Are Like Hashes

You know how a hash is created and used and that it is a way to map one thing to another. That means if you have a hash with a key apple and you want to get it, then you do this:

ex40a.rb

```
1 mystuff = {'apple' => "I AM APPLES!"}
2 puts mystuff['apple']
```

Keep this idea of “get X from Y” in your head, and now think about modules. You’ve made a few so far, and you should know they are:

1. A Ruby file with some functions or variables in it inside a module ... end block..
2. You import that file.
3. And you can access the functions or variables in that module with the . (dot) operator.

Imagine I have a module that I decide to name `mystuff.rb` and I put a function in it called `apple`. Here’s the module `mystuff.rb`:

ex40a.rb

```
1 # this goes in mystuff.rb
2 module MyStuff
3   def MyStuff.apple()
4     puts "I AM APPLES!"
5   end
6 end
```

Once I have this code, I can use the module MyStuff with require and then access the apple function:

ex40a.rb

```
1  require "./mystuff.rb"
2  MyStuff.apple()
```

I could also put a variable in it named tangerine:

ex40a.rb

```
1  module MyStuff
2    def MyStuff.apple()
3      puts "I AM APPLES!"
4    end
5
6    # this is just a variable
7    TANGERINE = "Living reflection of a dream"
8  end
```

I can access that the same way:

ex40a.rb

```
1  require "./mystuff.rb"
2
3  MyStuff.apple()
4  puts MyStuff::TANGERINE
```

Refer back to the hash, and you should start to see how this is similar to using a hash, but the syntax is different. Let's compare:

ex40a.rb

```
1  mystuff['apple'] # get apple from dict
2  MyStuff.apple() # get apple from the module
3  MyStuff::TANGERINE # same thing, it's just a variable
```

This means we have a *very common pattern* in Ruby:

1. Take a key-value style container.
2. Get something out of it by the key's name.

In the case of the hash, the key is a string and the syntax is [key]. In the case of the module, the key is an identifier, and the syntax is .key. Other than that, they are nearly the same thing.

Classes Are Like Modules

You can think about a module as a specialized hash that can store Ruby code so you can access it with the `.` operator. Ruby also has another construct that serves a similar purpose called a class. A class is a way to take a grouping of functions and data and place them inside a container so you can access them with the `.` (dot) operator.

If I were to create a class just like the `mystuff` module, I'd do something like this:

ex40a.rb

```
1  class MyStuff
2
3      def initialize()
4          @tangerine = "And now a thousand years between"
5      end
6
7      attr_reader :tangerine
8
9      def apple()
10         puts "I AM CLASSY APPLES!"
11     end
12
13 end
```

That looks complicated compared to modules, and there is definitely a lot going on by comparison, but you should be able to make out how this is like a “mini-module” with `MyStuff` having an `apple()` function in it. What is probably confusing is the `initialize()` function and use of `self.tangerine` for setting the `tangerine` variable.

Here's why classes are used instead of modules: You can take this `MyStuff` class and use it to craft many of them, millions at a time if you want, and each one won't interfere with each other. When you import a module there is only one for the entire program unless you do some monster hacks.

Before you can understand this though, you need to know what an “object” is and how to work with `MyStuff` just like you do with the `mystuff.rb` module.

Objects Are Like Require

If a class is like a “mini-module,” then there has to be a similar concept to `require` but for classes. That concept is called “instantiate”—which is just a fancy, obnoxious, overly smart way to say “create.” When you instantiate a class what you get is called an object.

You instantiate (create) a class by calling the class's `new` function, like this:

ex40a.rb

```
1  thing = MyStuff.new()
2  thing.apple()
3  puts thing.tangerine
```

The first line is the “instantiate” operation, and it’s a lot like calling a function. However, Ruby coordinates a sequence of events for you behind the scenes. I’ll go through these steps using the above code for MyStuff:

1. Ruby looks for MyStuff and sees that it is a class you’ve defined.
2. Ruby crafts an empty object with all the functions you’ve specified in the class using def because you did MyStuff.new().
3. Ruby then looks to see if you made a “magic” initialize function, and if you have it calls that function to initialize your newly created empty object.
4. In the MyStuff function initialize I then use @tangerine which is telling Ruby, “I want the tangerine variable that is part of this object.” Ruby uses operators like @ and \$ to say where a variable is located. When you did \$stdin you were saying, “the global stdin,” because \$ means global. When you do @tangerine you are saying, “the object’s tangerine”, because @ means “this object.”
5. In this case, I set @tangerine to a song lyric and then I’ve initialized this object.
6. Now Ruby can take this newly minted object and assign it to the thing variable for me to work with.

That’s the basics of how Ruby does this “mini-import” when you call a class like a function. Remember that this is *not* giving you the class, but instead is using the class as a *blueprint* for building a copy of that type of thing.

Keep in mind that I’m giving you a slightly inaccurate idea for how these work so that you can start to build up an understanding of classes based on what you know about modules. The truth is, classes and objects suddenly diverge from modules at this point. If I were being totally honest, I’d say something more like this:

- Classes are like blueprints or definitions for creating new mini-modules.
- Instantiation is how you make one of these mini-modules *and* require it at the same time. “Instantiate” just means to create an object from the class.
- The resulting created mini-module is called an object and you then assign it to a variable to work with it.

At this point objects behave differently from modules and this should only serve as a way for you to bridge over to understanding classes and objects.

Getting Things from Things

I now have three ways to get things from things:

ex40a.rb

```
1 # dict style
2 mystuff['apples']
3
4 # module style
5 MyStuff.apples()
6 puts MyStuff::TANGERINE
7
8 # class style
9 thing = MyStuff.new()
10 thing.apples()
11 puts thing.tangerine
```

A First Class Example

You should start seeing the similarities in these three key-value container types and probably have a bunch of questions. Hang on with the questions, as the next exercise will hammer home your "object-oriented vocabulary." In this exercise, I just want you to type in this code and get it working so that you have some experience before moving on.

ex40.rb

```
1 class Song
2
3     def initialize(lyrics)
4         @lyrics = lyrics
5     end
6
7     def sing_me_a_song()
8         @lyrics.each{|line| puts line }
9     end
10    end
11
12    happy_bday = Song.new(["Happy birthday to you",
13                            "I don't want to get sued",
14                            "So I'll stop right there"])
15
16    bulls_on_parade = Song.new(["They rally around tha family",
17                                "With pockets full of shells"])
18
19    happy_bday.sing_me_a_song()
20
21    bulls_on_parade.sing_me_a_song()
```

What You Should See

Exercise 40 Session

```
$ ruby ex40.rb
Happy birthday to you
I don't want to get sued
So I'll stop right there
They rally around tha family
With pockets full of shells
```

Study Drills

1. Write some more songs using this and make sure you understand that you're passing an array of strings as the lyrics.
2. Put the lyrics in a separate variable, then pass that variable to the class to use instead.
3. See if you can hack on this and make it do more things. Don't worry if you have no idea how, just give it a try, see what happens. Break it, trash it, thrash it; you can't hurt it.
4. Search online for "object-oriented programming" and try to overflow your brain with what you read. Don't worry if it makes absolutely no sense to you. Half of that stuff makes no sense to me too.

Common Student Question

Why do I need @ on @tangerine when I make initialize or other functions?

If you don't have the @ on a variable then Ruby doesn't know which variable you are referring to. Do you mean a tangerine in your function, in the script, or in the object your setting up? The @ on @tangerine makes your usage specific so Ruby knows where to look.

Learning to Speak Object Oriented

In this exercise I'm going to teach you how to speak "object oriented." What I'll do is give you a small set of words with definitions you need to know. Then I'll give you a set of sentences with holes in them that you'll have to understand. Finally, I'm going to give you a large set of exercises that you have to complete to make these sentences solid in your vocabulary.

Word Drills

class Tell Ruby to make a new type of thing.

object Two meanings: the most basic type of thing, and any instance of some thing.

instance What you get when you tell Ruby to create a class.

def How you define a function inside a class.

@ Inside the functions in a class, self is a variable for the instance/object being accessed.

inheritance The concept that one class can inherit traits from another class, much like you and your parents.

composition The concept that a class can be composed of other classes as parts, much like how a car has wheels.

attribute A property classes have that are from composition and are usually variables.

is-a A phrase to say that something inherits from another, as in a "salmon" is-a "fish."

has-a A phrase to say that something is composed of other things or has a trait, as in "a salmon has-a mouth."

Take some time to make flash cards for these terms and memorize them. As usual this won't make too much sense until after you are finished with this exercise, but you need to know the base words first.

Phrase Drills

Next I have a list of Ruby code snippets on the left, and the English sentences for them:

class X(Y) "Make a class named X that is-a Y."

class X(object): def initialize(J) "class X has-a initialize that takes a J parameter."

```
class X(object): def M(self, J) "class X has-a function named M that takes a J parameter."  
foo = X() "Set foo to an instance of class X."  
foo.M(J) "From foo get the M function, and call it with parameter J."  
foo.K = Q "From foo get the K attribute and set it to Q."
```

In each of these where you see X, Y, M, J, K, Q, and foo you can treat those like blank spots. For example, I can also write these sentences as follows:

1. "Make a class named ??? that is-a Y."
2. "class ??? has-a initialize that takes ??? parameters."
3. "class ??? has-a function named ??? that takes ??? parameters."
4. "Set foo to an instance of class ???."
5. "From foo get the ??? function, and call it with parameters ???."
6. "From foo get the ??? attribute and set it to ???."

Again, write these on some flash cards and drill them. Put the Ruby code snippet on the front and the sentence on the back. You *have* to be able to say the sentence exactly the same every time whenever you see that form. Not sort of the same, but exactly the same.

Combined Drills

The final preparation for you is to combine the word drills with the phrase drills. What I want you to do for this drill is this:

1. Take a phrase card and drill it.
2. Flip it over and read the sentence, and for each word in the sentence that is in your word drills, get that card.
3. Drill those words for that sentence.
4. Keep going until you are bored, then take a break and do it again.

A Reading Test

I now have a little Ruby hack script that will drill you on these words you know in an infinite manner. This is a simple script you should be able to figure out, and the only thing it does is use a

library called `urllib` to download a list of words I have. Here's the script, which you should enter into `oop_test.rb` to work with it:

ex41.rb

```
1  require 'open-uri'
2
3  WORD_URL = "http://learncodethehardway.org/words.txt"
4  WORDS = []
5
6  PHRASES = {
7      "class ### < ###\nend" =>
8          "Make a class named ### that is-a ###.",
9      "class ###\n\ndef initialize(@@@)\n\tend\nend" =>
10         "class ### has-a initialize that takes @@@ parameters.",
11      "class ###\n\ndef ***(@@@)\n\tend\nend" =>
12         "class ### has-a function named *** that takes @@@ parameters.",
13      "*** = ###.new()" =>
14         "Set *** to an instance of class ###.",
15      "***.***(@@@)" =>
16         "From *** get the *** function, and call it with parameters @@@.",
17      "***.*** = '***'" =>
18         "From *** get the *** attribute and set it to '***'."
19  }
20
21  PHRASE_FIRST = ARGV[0] == "english"
22
23  open(WORD_URL) { |f|
24      f.each_line { |word| WORDS.push(word.chomp)}
25  }
26
27  def craft_names(rand_words, snippet, pattern, caps=false)
28      names = snippet.scan(pattern).map do
29          word = rand_words.pop()
30          caps ? word.capitalize : word
31      end
32
33      return names * 2
34  end
35
36  def craft_params(rand_words, snippet, pattern)
37      names = (0...snippet.scan(pattern).length).map do
38          param_count = rand(3) + 1
39          params = (0...param_count).map { |x| rand_words.pop() }
40          params.join(', ')
41      end
42
43      return names * 2
44  end
```

```
45
46  def convert(snippet, phrase)
47    rand_words = WORDS.sort_by {rand}
48    class_names = craft_names(rand_words, snippet, /###/, caps=true)
49    other_names = craft_names(rand_words, snippet, /\*\*\*/)
50    param_names = craft_params(rand_words, snippet, /@@@/)
51
52    results = []
53
54    [snippet, phrase].each do |sentence|
55      # fake class names, also copies sentence
56      result = sentence.gsub(/###/) {|x| class_names.pop }
57
58      # fake other names
59      result.gsub!(/\*\*\*/) {|x| other_names.pop }
60
61      # fake parameter lists
62      result.gsub!(/@@@/) {|x| param_names.pop }
63
64      results.push(result)
65    end
66
67    return results
68  end
69
70  # keep going until they hit CTRL-D
71  loop do
72    snippets = PHRASES.keys().sort_by {rand}
73
74    for snippet in snippets
75      phrase = PHRASES[snippet]
76      question, answer = convert(snippet, phrase)
77
78      if PHRASE_FIRST
79        question, answer = answer, question
80      end
81
82      print question, "\n\n> "
83
84      exit(0) unless $stdin.gets
85
86      puts "\nANSWER: %s\n\n" % answer
87    end
88  end
```

Run this script and try to translate the “object-oriented phrases” into English translations. You should see that the PHRASES dict has both forms and you just have to enter the correct one.

Practice English to Code

Next you should run the script with the “english” option so that you drill the inverse operation:

```
$ ruby oop_test.rb english
```

Remember that these phrases are using nonsense words. Part of learning to read code well is to stop placing so much meaning on the names used for variables and classes. Too often people will read a word like “Cork” and suddenly get derailed because that word will confuse them about the meaning. In the above example, “Cork” is just an arbitrary name chosen for a class. Don’t place any other meaning on it, and instead treat it like the patterns I’ve given you.

Reading More Code

You are now to go on a new quest to read even more code, to read the phrases you just learned in the code you read. You will look for all the files with classes, and then do the following:

1. For each class give its name and what other classes it inherits from.
2. Under that, list every function it has and the parameters they take.
3. List all of the attributes it uses on self.
4. For each attribute, give the class it is.

The goal is to go through real code and start learning to “pattern match” the phrases you just learned against how they’re used. If you drill this enough you should start to see these patterns shout at you in the code whereas before they just seemed like vague blank spots you didn’t know.

Common Student Questions

This script is hard to get running!

By this point you should be able to type this in and get it working. It does have a few little tricks here and there but there’s nothing complex about it. Just do all the things you’ve learned so far to debug scripts. Type each line in, confirm that it’s *exactly* like mine, and research anything you don’t know online.

It’s still too hard!

You can do this. Take it very slow, character by character if you have to, but type it in exactly and figure out what it does.

This page intentionally left blank

Is-A, Has-A, Objects, and Classes

An important concept that you have to understand is the difference between a class and an object. The problem is, there is no real "difference" between a class and an object. They are actually the same thing at different points in time. I will demonstrate by a Zen koan:

What is the difference between a Fish and a Salmon?

Did that question sort of confuse you? Really sit down and think about it for a minute. I mean, a Fish and a Salmon are different but, wait, they are the same thing, right? A Salmon is a *kind* of Fish, so I mean it's not different. But at the same time, because a Salmon is a particular type of Fish so it's actually different from all other Fish. That's what makes it a Salmon and not a Halibut. So a Salmon and a Fish are the same but different. Weird.

This question is confusing because most people do not think about real things this way, but they intuitively understand them. You do not need to think about the difference between a Fish and a Salmon because you *know* how they are related. You know a Salmon is a *kind* of Fish and that there are other kinds of Fish without having to understand that.

Let's take it one step further. Let's say you have a bucket full of three Salmon and because you are a nice person, you have decided to name them Frank, Joe, and Mary. Now, think about this question:

What is the difference between Mary and a Salmon?

Again this is a weird question, but it's a bit easier than the Fish versus Salmon question. You know that Mary is a Salmon, so she's not really different. She's just a specific "instance" of a Salmon. Joe and Frank are also instances of Salmon. What do I mean when I say instance? I mean they were created from some other Salmon and now represent a real thing that has Salmon-like attributes.

Now for the mind-bending idea: Fish is a class, and Salmon is a class, and Mary is an object. Think about that for a second. Let's break it down slowly and see if you get it.

A Fish is a class, meaning it's not a *real* thing, but rather a word we attach to instances of things with similar attributes. Got fins? Got gills? Lives in water? Alright it's probably a Fish.

Someone with a Ph.D. then comes along and says, "No, my young friend, *this* Fish is actually *Salmo salar*, affectionately known as a Salmon." This professor has just clarified the Fish further; and made a new class called "Salmon" that has more specific attributes. Longer nose, reddish flesh, big, lives in the ocean or fresh water, tasty? Probably a Salmon.

Finally, a cook comes along and tells the Ph.D., "No, you see this Salmon right here, I'll call her Mary and I'm going to make a tasty fillet out of her with a nice sauce." Now you have this *instance* of a Salmon (which also is an instance of a Fish) named Mary turned into something real that is filling your belly. It has become an object.

There you have it: Mary is a kind of Salmon that is a kind of Fish. Object is a class is a class.

How This Looks in Code

This is a weird concept, but to be very honest you only have to worry about it when you make new classes, and when you use a class. I will show you two tricks to help you figure out whether something is a class or object.

First, you need to learn two catch phrases: "is-a" and "has-a." You use the phrase is-a when you talk about objects and classes being related to each other by a class relationship. You use has-a when you talk about objects and classes that are related only because they *reference* each other.

Now, go through this piece of code and replace each ##?? comment with a comment that says whether the next line represents an is-a or a has-a relationship and what that relationship is. In the beginning of the code, I've laid out a few examples, so you just have to write the remaining ones.

Remember, is-a is the relationship between Fish and Salmon, while has-a is the relationship between Salmon and Gills.

ex42.rb

```
1  ## Animal is-a object look at the extra credit
2  class Animal
3  end
4
5  ## ??
6  class Dog < Animal
7
8      def initialize(name)
9          ## ??
10         @name = name
11     end
12 end
13
14 ## ??
15 class Cat < Animal
16
17     def initialize(name)
18         ## ??
19         @name = name
```

```
20      end
21  end
22
23  ## ??
24  class Person
25
26    def initialize(name)
27      ## ??
28      @name = name
29
30      ## Person has-a pet of some kind
31      @pet = nil
32  end
33
34  attr_accessor :pet
35
36
37  ## ??
38  class Employee < Person
39
40    def initialize(name, salary)
41      ## ?? hmm what is this strange magic?
42      super(name)
43      ## ??
44      @salary = salary
45  end
46
47
48
49  ## ??
50  class Fish
51  end
52
53  ## ??
54  class Salmon < Fish
55  end
56
57  ## ??
58  class Halibut < Fish
59  end
60
61
62  ## rover is-a Dog
63  rover = Dog.new("Rover")
64
65  ## ??
66  satan = Cat.new("Satan")
67
```

```
68      ## ??
69  mary = Person.new("Mary")
70
71  ## ??
72  mary.pet = satan
73
74  ## ??
75  frank = Employee.new("Frank", 120000)
76
77  ## ??
78  frank.pet = rover
79
80  ## ??
81  flipper = Fish.new()
82
83  ## ??
84  crouse = Salmon.new()
85
86  ## ??
87  harry = Halibut.new()
```

Study Drills

1. Research why Ruby added this strange object class, and what that means.
2. Is it possible to use a class like it's an object?
3. Fill out the animals, fish, and people in this exercise with functions that make them do things. See what happens when functions are in a "base class" like Animal versus in say Dog.
4. Find other people's code and work out all the is-a and has-a relationships.
5. Make some new relationships that are arrays and hashes so you can also have "has-many" relationships.
6. Do you think there's such a thing as an "is-many" relationship? Read about "multiple inheritance," then avoid it if you can.

Common Student Questions

What are these ## ?? comments for?

Those are "fill-in-the-blank" comments that you are supposed to fill in with the right "is-a" or "has-a" concepts. Read this exercise again and look at the other comments to see what I mean.

What is the point of `@pet = nil`?

That gives a default to a Person's pet that is `nil` or "not set to anything."

What does `super(name)` do?

That's how you can run the `initialize` of the parent class `Person` before you do what you need inside `Employee`. Go search for "ruby super" online and read the various advice on it.

This page intentionally left blank

Basic Object-Oriented Analysis and Design

I'm going to describe a process to use when you want to build something using Ruby, specifically with object-oriented programming (OOP). What I mean by a "process" is that I'll give you a set of steps that you do in order, but that you aren't meant to be a slave to or that will totally always work for every problem. They are just a good starting point for many programming problems and shouldn't be considered the *only* way to solve these types of problems. This process is just one way to do it that you can follow.

The process is as follows:

1. Write or draw about the problem.
2. Extract key concepts from 1 and research them.
3. Create a class hierarchy and object map for the concepts.
4. Code the classes and do a test to run them.
5. Repeat and refine.

The way to look at this process is that it is "top down," meaning it starts from the very abstract loose idea and then slowly refines it until the idea is solid and something you can code.

I start by just writing about the problem and trying to think up anything I can about it. Maybe I'll even draw a diagram or two, maybe a map of some kind, or even write myself a series of emails describing the problem. This gives me a way to express the key concepts in the problem and also explore what I might already know about it.

Then I go through these notes, drawings, and descriptions and I pull out the key concepts. There's a simple trick to doing this: Simply make a list of all the *nouns* and *verbs* in your writing and drawings, and then write out how they're related. This gives me a good list of names for classes, objects, and functions in the next step. I take this list of concepts and then research any that I don't understand so I can refine them further if I needed.

Once I have my list of concepts, I create a simple outline/tree of the concepts and how they are related as classes. You can usually take your list of nouns and start asking "Is this one like other concept nouns? That means they have a common parent class, so what is it called?" Keep doing this until you have a class hierarchy that's just a simple tree list or a diagram. Then take the *verbs* you have and see if those are function names for each class and put them in your tree.

With this class hierarchy figured out, I sit down and write some basic skeleton code that has just the classes, their functions, and nothing more. I then write a test that runs this code and makes sure the classes I've made make sense and work right. Sometimes I may write the test first though, and other times I might write a little test, a little code, a little test, etc., until I have the whole thing built.

Finally, I keep cycling over this process repeating it and refining as I go and making it as clear as I can before doing more implementation. If I get stuck at any particular part because of a concept or problem I haven't anticipated, then I sit down and start the process over on just that part to figure it out more before continuing.

I will now go through this process while coming up with a game engine and a game for this exercise.

The Analysis of a Simple Game Engine

The game I want to make is called "Gothons from Planet Percal #25" and it will be a small space adventure game. With nothing more than that concept in my mind, I can explore the idea and figure out how to make the game come to life.

Write or Draw about the Problem

I'm going to write a little paragraph for the game:

"Aliens have invaded a space ship and our hero has to go through a maze of rooms defeating them so he can escape into an escape pod to the planet below. The game will be more like a Zork or Adventure type game with text outputs and funny ways to die. The game will involve an engine that runs a map full of rooms or scenes. Each room will print its own description when the player enters it and then tell the engine which room to run next out of the map."

At this point I have a good idea for the game and how it would run, so now I want to describe each scene:

Death This is when the player dies and should be something funny.

Central Corridor This is the starting point and has a Gothon already standing there that the player has to defeat with a joke before continuing.

Laser Weapon Armory This is where the hero gets a neutron bomb to blow up the ship before getting to the escape pod. It has a keypad the hero has to guess the number for.

The Bridge Another battle scene with a Gothon where the hero places the bomb.

Escape Pod Where the hero escapes to, but only after guessing the right escape pod.

At this point I might draw out a map of these, maybe write more descriptions of each room—whatever comes to mind as I explore the problem.

Extract Key Concepts and Research Them

I now have enough information to extract some of the nouns and analyze their class hierarchy. First I make a list of all the nouns:

- Alien
- Player
- Ship
- Maze
- Room
- Scene
- Gothon
- Escape Pod
- Planet
- Map
- Engine
- Death
- Central Corridor
- Laser Weapon Armory
- The Bridge

I would also possibly go through all the verbs and see if there are any that might be good function names, but I'll skip that for now.

At this point you might also research each of these concepts and anything you don't know right now. For example, I might play a few of these types of games and make sure I know how they work. I might research how ships are designed or how bombs work. Maybe I'll research some technical issue, like how to store the game's state in a database. After I've done this research, I might start over at step 1 based on new information I have and rewrite my description and extract new concepts.

Create a Class Hierarchy and Object Map for the Concepts

Once I have that I turn it into a class hierarchy by asking, "What is similar to other things?" I also ask, "What is basically just another word for another thing?"

Right away, I see that "Room" and "Scene" are basically the same thing depending on how I want to do things. I'm going to pick "Scene" for this game. Then I see that all the specific rooms like "Central Corridor" are basically just Scenes. I see also that Death is basically a Scene, which confirms my choice of "Scene" over "Room" since you can have a death scene, but a death room is kind of odd. "Maze" and "Map" are basically the same so I'm going to go with "Map" since I used it more often. I don't want to do a battle system so I'm going to ignore "Alien" and "Player" and save that for later. The "Planet" could also just be another scene instead of something specific.

After going through that thought process, I start to make a class hierarchy that looks like this in my text editor:

- * Map
- * Engine
- * Scene
 - * Death
 - * Central Corridor
 - * Laser Weapon Armory
 - * The Bridge
 - * Escape Pod

I would then go through and figure out which actions are needed on each thing based on verbs in the description. For example, I know from the description I'm going to need a way to "run" the engine, "get the next scene" from the map, get the "opening scene," and "enter" a scene. I'll add those like this:

- * Map
 - next_scene
 - opening_scene
- * Engine
 - play
- * Scene
 - enter

- * Death
- * Central Corridor
- * Laser Weapon Armory
- * The Bridge
- * Escape Pod

Notice how I just put `-enter` under `Scene` because I know that all the scenes under it will inherit this action and have to override it later.

Code the Classes and a Test to Run Them

Once I have this tree of classes and some of the functions, I open up a source file in my editor and try to write the code for it. Usually I'll just copy-paste the tree into the source file and then edit it into classes. Here's a small example of how this might look at first, with a simple little test at the end of the file.

ex43_classes.rb

```
1  class Scene
2    def enter()
3    end
4  end
5
6
7  class Engine
8
9    def initialize(scene_map)
10   end
11
12   def play()
13   end
14 end
15
16 class Death < Scene
17
18   def enter()
19   end
20 end
21
22 class CentralCorridor < Scene
23
24   def enter()
25   end
26 end
27
28 class LaserWeaponArmory < Scene
29
30   def enter()
```

```
31      end
32  end
33
34  class TheBridge < Scene
35
36      def enter()
37      end
38  end
39
40  class EscapePod < Scene
41
42      def enter()
43      end
44  end
45
46
47  class Map
48
49      def initialize( start_scene)
50      end
51
52      def next_scene( scene_name)
53      end
54
55      def opening_scene()
56      end
57  end
58
59
60  a_map = Map.new('central_corridor')
61  a_game = Engine.new(a_map)
62  a_game.play()
```

In this file you can see that I simply replicated the hierarchy I wanted, and then added a little bit of code at the end to run it and see if it all works in this basic structure. In the later sections of this exercise, you'll fill in the rest of this code and make it work to match the description of the game.

Repeat and Refine

The last step in my little process isn't so much a step as it is a `while`-loop. You don't ever do this as a one-pass operation. Instead, you go back over the whole process again and refine it based on information you've learned from later steps. Sometimes I'll get to step 3 and realize that I need to work on 1 and 2 more, so I'll stop and go back and work on those. Sometimes I'll get a flash of inspiration and jump to the end to code up the solution in my head while I have it there, but then I'll go back and do the previous steps to make sure I cover all the possibilities I have.

The other idea in this process is that it's not just something you do at one single level but something that you can do at every level when you run into a particular problem. Let's say I don't know how to

write the Engine.play method yet. I can stop and do this whole process on *just* that one function to figure out how to write it.

Top Down Versus Bottom Up

The process is typically labeled "top down" since it starts at the most abstract concepts (the top) and works its way down to actual implementation. I want you to use this process when analyzing problems in the book from now on, but you should know that there's another way to solve problems in programming that starts with code and then goes "up" to the abstract concepts. This other way is labeled "bottom up." Here are the general steps you follow to do this:

1. Take a small piece of the problem; hack on some code and get it to run barely.
2. Refine the code into something more formal with classes and automated tests.
3. Extract the key concepts you're using and try to find research for them.
4. Write a description of what's really going on.
5. Go back and refine the code, possibly throwing it out and starting over.
6. Repeat, moving on to some other piece of the problem.

I find the bottom up process is better once you're more solid at programming, and are naturally thinking in code about problems. This process is very good when you know small pieces of the overall puzzle, but maybe don't have enough information yet about the overall concept. Breaking it down in little pieces and exploring with code then helps you slowly grind away at the problem until you've solved it. However, remember that your solution will probably be meandering and weird, so that's why my version of this process involves going back and finding research, then cleaning things up based on what you've learned.

The Code for "Gothons from Planet Percal #25"

Stop! I'm going to show you my final solution to the above problem but I don't want you to just jump in and type this up. I want you to take the rough skeleton code I did and try to make it work based on the description. Once you have your solution then you can come back and see how I did it.

I'm going to break this final file ex43.rb down into sections and explain each one rather than dump all the code at once.

ex43.rb

```
1 class Scene
2   def enter()
3     puts "This scene is not yet configured. Subclass it and implement enter()."
```

```
4     exit(1)
5   end
6 end
```

As you saw in the skeleton code, I have a base class for Scene that will have the common things that all scenes do. In this simple program, they don't do much so this is more a demonstration of what you would do to make a base class.

ex43.rb

```
1  class Engine
2
3    def initialize(scene_map)
4      @scene_map = scene_map
5    end
6
7    def play()
8      current_scene = @scene_map.opening_scene()
9      last_scene = @scene_map.next_scene('finished')
10
11    while current_scene != last_scene
12      next_scene_name = current_scene.enter()
13      current_scene = @scene_map.next_scene(next_scene_name)
14    end
15
16    # be sure to print out the last scene
17    current_scene.enter()
18  end
19 end
```

I also have my Engine class, and you can see how I'm already using the methods for Map.opening_scene and Map.next_scene. Because I've done a bit of planning, I can just assume I'll write those and then use them before I've written the Map class.

ex43.rb

```
1  class Death < Scene
2
3  @@quips = [
4    "You died. You kinda suck at this.",
5    "Your mom would be proud...if she were smarter.",
6    "Such a luser.",
7    "I have a small puppy that's better at this."
8  ]
9
10 def enter()
11   puts @@quips[rand(0..@@quips.length - 1)]
12   exit(1)
13 end
14 end
```

My first scene is the odd scene named Death, which shows you the simplest kind of scene you can write.

ex43.rb

```
1 class CentralCorridor < Scene
2
3   def enter()
4     puts "The Gothons of Planet Percal #25 have invaded your ship and destroyed"
5     puts "your entire crew. You are the last surviving member and your last"
6     puts "mission is to get the neutron destruct bomb from the Weapons Armory,"
7     puts "put it in the bridge, and blow the ship up after getting into an "
8     puts "escape pod."
9     puts "\n"
10    puts "You're running down the central corridor to the Weapons Armory when"
11    puts "a Gothon jumps out, red scaly skin, dark grimy teeth, and evil clown costume"
12    puts "flowing around his hate filled body. He's blocking the door to the"
13    puts "Armory and about to pull a weapon to blast you."
14    print "> "
15
16    action = $stdin.gets.chomp
17
18    if action == "shoot!"
19      puts "Quick on the draw you yank out your blaster and fire it at the Gothon."
20      puts "His clown costume is flowing and moving around his body, which throws"
21      puts "off your aim. Your laser hits his costume but misses him entirely. This"
22      puts "completely ruins his brand new costume his mother bought him, which"
23      puts "makes him fly into an insane rage and blast you repeatedly in the face until"
24      puts "you are dead. Then he eats you."
25      return 'death'
26
27    elsif action == "dodge!"
28      puts "Like a world class boxer you dodge, weave, slip and slide right"
29      puts "as the Gothon's blaster cranks a laser past your head."
30      puts "In the middle of your artful dodge your foot slips and you"
31      puts "bang your head on the metal wall and pass out."
32      puts "You wake up shortly after only to die as the Gothon stomps on"
33      puts "your head and eats you."
34      return 'death'
35
36    elsif action == "tell a joke"
37      puts "Lucky for you they made you learn Gothon insults in the academy."
38      puts "You tell the one Gothon joke you know:"
39      puts "Lbhe zbgure vf fb sng, jura fur fvgf nebhaq gur ubhfr, fur fvgf nebhaq gur ubhfr."
40      puts "The Gothon stops, tries not to laugh, then busts out laughing and can't move."
41      puts "While he's laughing you run up and shoot him square in the head"
42      puts "putting him down, then jump through the Weapon Armory door."
43      return 'finished'
44
45    else
46      puts "DOES NOT COMPUTE!"
47      return 'central_corridor'
48    end
49  end
50 end
```

After that I've created the CentralCorridor, which is the start of the game. I'm doing the scenes for the game before the Map because I need to reference them later.

ex43.rb

```
1  class LaserWeaponArmory < Scene
2
3  def enter()
4      puts "You do a dive roll into the Weapon Armory, crouch and scan the room"
5      puts "for more Gothons that might be hiding. It's dead quiet, too quiet."
6      puts "You stand up and run to the far side of the room and find the"
7      puts "neutron bomb in its container. There's a keypad lock on the box"
8      puts "and you need the code to get the bomb out. If you get the code"
9      puts "wrong 10 times then the lock closes forever and you can't"
10     puts "get the bomb. The code is 3 digits."
11     code = "#{rand(1..9)}#{rand(1..9)}#{rand(1..9)}"
12     print "[keypad]> "
13     guess = $stdin.gets.chomp
14     guesses = 0
15
16     while guess != code && guesses < 10
17         puts "BZZZZEDDD!"
18         guesses += 1
19         print "[keypad]> "
20         guess = $stdin.gets.chomp
21     end
22
23     if guess == code
24         puts "The container clicks open and the seal breaks, letting gas out."
25         puts "You grab the neutron bomb and run as fast as you can to the"
26         puts "bridge where you must place it in the right spot."
27         return 'the_bridge'
28     else
29         puts "The lock buzzes one last time and then you hear a sickening"
30         puts "melting sound as the mechanism is fused together."
31         puts "You decide to sit there, and finally the Gothons blow up the"
32         puts "ship from their ship and you die."
33         return 'death'
34     end
35   end
36 end
37
38
39 class TheBridge < Scene
40
41   def enter()
42       puts "You burst onto the Bridge with the netron destruct bomb"
43       puts "under your arm and surprise 5 Gothons who are trying to"
44       puts "take control of the ship. Each of them has an even uglier"
45       puts "clown costume than the last. They haven't pulled their"
46       puts "weapons out yet, as they see the active bomb under your"
47       puts "arm and don't want to set it off."
48       print "> "
```

```
49      action = $stdin.gets.chomp
50
51      if action == "throw the bomb"
52          puts "In a panic you throw the bomb at the group of Goths"
53          puts "and make a leap for the door. Right as you drop it a"
54          puts "Gothon shoots you right in the back killing you."
55          puts "As you die you see another Gothon frantically try to disarm"
56          puts "the bomb. You die knowing they will probably blow up when"
57          puts "it goes off."
58          return 'death'
59
60      elsif action == "slowly place the bomb"
61          puts "You point your blaster at the bomb under your arm"
62          puts "and the Goths put their hands up and start to sweat."
63          puts "You inch backward to the door, open it, and then carefully"
64          puts "place the bomb on the floor, pointing your blaster at it."
65          puts "You then jump back through the door, punch the close button"
66          puts "and blast the lock so the Goths can't get out."
67          puts "Now that the bomb is placed you run to the escape pod to"
68          puts "get off this tin can."
69          return 'escape_pod'
70
71      else
72          puts "DOES NOT COMPUTE!"
73          return "the_bridge"
74      end
75  end
76 end
77
78
79 class EscapePod < Scene
80
81     def enter()
82         puts "You rush through the ship desperately trying to make it to"
83         puts "the escape pod before the whole ship explodes. It seems like"
84         puts "hardly any Goths are on the ship, so your run is clear of"
85         puts "interference. You get to the chamber with the escape pods, and"
86         puts "now need to pick one to take. Some of them could be damaged"
87         puts "but you don't have time to look. There's 5 pods, which one"
88         puts "do you take?"
89
90     good_pod = rand(1..5)
91     print "[pod #]> "
92     guess = $stdin.gets.chomp.to_i
93
94     if guess != good_pod
95         puts "You jump into pod %s and hit the eject button." % guess
96         puts "The pod escapes out into the void of space, then"
97         puts "implodes as the hull ruptures, crushing your body"
98         puts "into jam jelly."
99         return 'death'
100    else
101        puts "You jump into pod %s and hit the eject button." % guess
```

```
102     puts "The pod easily slides out into space heading to"
103     puts "the planet below. As it flies to the planet, you look"
104     puts "back and see your ship implode then explode like a"
105     puts "bright star, taking out the Gothon ship at the same"
106     puts "time. You won!"
107
108
109     return 'finished'
110   end
111 end
112 end
113
114 class Finished < Scene
115   def enter()
116     puts "You won! Good job."
117   end
118 end
```

This is the rest of the game's scenes, and since I know I need them and have thought about how they'll flow together I'm able to code them up directly.

Incidentally, I wouldn't just type all this code in. Remember I said to try and build this incrementally, one little bit at a time. I'm just showing you the final result.

ex43.rb

```
1  class Map
2    @@scenes = {
3      'central_corridor' => CentralCorridor.new(),
4      'laser_weapon_armory' => LaserWeaponArmory.new(),
5      'the_bridge' => TheBridge.new(),
6      'escape_pod' => EscapePod.new(),
7      'death' => Death.new()
8    }
9
10
11  def initialize(start_scene)
12    @start_scene = start_scene
13  end
14
15
16  def next_scene(scene_name)
17    val = @@scenes[scene_name]
18    return val
19  end
20
21  def opening_scene()
22    return next_scene(@start_scene)
23  end
24 end
```

After that I have my Map class, and you can see it is storing each scene by name in a hash, and then I refer to that hash with `@@scenes`. This is also why the map comes after the scenes—because the hash has to refer to the scenes so they have to exist.

ex43.rb

```
1 a_map = Map.new('central廊道')
2 a_game = Engine.new(a_map)
3 a_game.play()
```

Finally I've got my code that runs the game by making a Map and then handing that map to an Engine before calling play to make the game work.

What You Should See

Make sure you understand the game and that you tried to solve it yourself first. One thing to do if you're stumped, is cheat a little by reading my code, then continue trying to solve it yourself.

When I run my game it looks like this:

Exercise 43 Session

```
$ ruby ex43.rb
```

The Goths of Planet Percal #25 have invaded your ship and destroyed your entire crew. You are the last surviving member and your last mission is to get the neutron destruct bomb from the Weapons Armory, put it in the bridge, and blow the ship up after getting into an escape pod.

You're running down the central corridor to the Weapons Armory when a Goth jumps out, red scaly skin, dark grimy teeth, and evil clown costume flowing around his hate filled body. He's blocking the door to the Armory and about to pull a weapon to blast you.

> dodge!

Like a world class boxer you dodge, weave, slip and slide right as the Goth's blaster cranks a laser past your head.

In the middle of your artful dodge your foot slips and you bang your head on the metal wall and pass out.

You wake up shortly after only to die as the Goth stomps on your head and eats you.

I have a small puppy that's better at this.

Study Drills

1. Change it! Maybe you hate this game. Could be too violent, or you aren't into sci-fi. Get the game working, then change it to what you like. This is your computer, you make it do what you want.

2. I have a bug in this code. Why is the door lock guessing 11 times?
3. Explain how returning the next room works.
4. Add cheat codes to the game so you can get past the more difficult rooms. I can do this with two words on one line.
5. Go back to my description and analysis, and try to build a small combat system for the hero and the various Gothons he encounters.
6. This is actually a small version of something called a "finite state machine." Read about them. They might not make sense but try anyway.

Common Student Question

Where can I find stories for my own games?

You can make them up, just like you would tell a story to a friend. Or you can take simple scenes from a book or movie you like.

Inheritance Versus Composition

In the fairy tales about heroes defeating evil villains there's always a dark forest of some kind. It could be a cave, a forest, another planet, or just some place that everyone knows the hero shouldn't go. Of course, shortly after the villain is introduced, you find out, yes, the hero has to go to that stupid forest to kill the bad guy. It seems the hero just keeps getting into situations that require him to risk his life in this evil forest.

You rarely read fairy tales about the heroes who are smart enough to just avoid the whole situation entirely. You never hear a hero say, "Wait a minute, if I leave to make my fortunes on the high seas leaving Buttercup behind, I could die and then she'd have to marry some ugly prince named Humperdink. Humperdink! I think I'll stay here and start a Farm Boy for Rent business." If he did that, there'd be no fire swamp, dying, reanimation, sword fights, giants, or any kind of story really. Because of this, the forest in these stories seems to exist like a black hole that drags the hero in no matter what he does.

In object-oriented programming, Inheritance is the evil forest. Experienced programmers know to avoid this evil because they know that deep inside the Dark Forest Inheritance is the Evil Queen Multiple Inheritance. She likes to eat software and programmers with her massive complexity teeth, chewing on the flesh of the fallen. But the forest is so powerful and so tempting that nearly every programmer has to go into it, and try to make it out alive with the Evil Queen's head before they can call themselves real programmers. You just can't resist the Inheritance Forest's pull, so you go in. After the adventure, you learn to just stay out of that stupid forest and bring an army if you are ever forced to go in again.

This is basically a funny way to say that I'm going to teach you something you should use carefully called Inheritance. Programmers who are currently in the forest battling the Queen will probably tell you that you have to go in. They say this because they need your help, because what they've created is probably too much for them to handle. But you should always remember this:

Most of the uses of inheritance can be simplified or replaced with composition, and I'll show you how in this exercise.

What Is Inheritance?

Inheritance is used to indicate that one class will get most or all of its features from a parent class. This happens implicitly whenever you write `class Foo(Bar)`, which says "Make a class Foo that inherits from Bar." When you do this, the language makes any action that you do on instances of Foo also work as if it was done to an instance of Bar. Doing this lets you put common functionality in the Bar class, then specialize that functionality in the Foo class as needed.

When you are creating this kind of specialization, there are three ways that the parent and child classes can interact:

1. Actions on the child imply an action on the parent.
2. Actions on the child override the action on the parent.
3. Actions on the child alter the action on the parent.

I will now demonstrate each of these in order and show you code for them.

Implicit Inheritance

First I will show you the implicit actions that happen when you define a function in the parent, but *not* in the child.

ex44a.rb

```
1 class Parent
2
3   def implicit()
4     puts "PARENT implicit()"
5   end
6 end
7
8 class Child < Parent
9 end
10
11 dad = Parent.new()
12 son = Child.new()
13
14 dad.implicit()
15 son.implicit()
```

The class `Child` is an empty class that inherits all of its behavior from `Parent`. When you run this code you get the following:

Exercise 44a Session

```
$ ruby ex44a.rb
PARENT implicit()
PARENT implicit()
```

Notice how even though I'm calling `son.implicit()` on line 15, and even though `Child` does *not* have an `implicit` function defined, it still works and it calls the one defined in `Parent`. This shows you that, if you put functions in a base class (i.e., `Parent`), then all subclasses (i.e., `Child`) will automatically get those features. Very handy for repetitive code you need in many classes.

Override Explicitly

The problem with having functions called implicitly is that sometimes you want the child to behave differently. In this case you want to override the function in the child, effectively replacing the functionality. To do this, just define a function with the same name in Child. Here's an example:

ex44b.rb

```
1 class Parent
2
3   def override()
4     puts "PARENT override()"
5   end
6 end
7
8 class Child < Parent
9   def override()
10    puts "CHILD override()"
11  end
12 end
13
14 dad = Parent.new()
15 son = Child.new()
16
17 dad.override()
18 son.override()
```

In this example I have a function named `override` in both classes, so let's see what happens when you run it.

Exercise 44b Session

```
$ ruby ex44b.rb
PARENT override()
CHILD override()
```

As you can see, when line 17 runs, it runs the `Parent.override` function because that variable (`dad`) is a `Parent`. But when line 18 runs, it prints out the `Child.override` message because `son` is an instance of `Child` and `Child` overrides that function by defining its own version.

Take a break right now and try playing with these two concepts before continuing.

Alter Before or After

The third way to use inheritance is a special case of overriding where you want to alter the behavior before or after the `Parent` class's version runs. You first override the function just like in the last example, but then you use a Ruby built-in function named `super` to get the `Parent` version to call. Here's the example of doing that so you can make sense of this description:

ex44c.rb

```
1  class Parent
2    def altered()
3      puts "PARENT altered()"
4    end
5  end
6
7  class Child < Parent
8    def altered()
9      puts "CHILD, BEFORE PARENT altered()"
10     super()
11     puts "CHILD, AFTER PARENT altered()"
12   end
13
14 end
15
16 dad = Parent.new()
17 son = Child.new()
18
19 dad.altered()
20 son.altered()
```

The important lines here are 9-11, where in the `Child` I do the following when `son.altered()` is called:

1. Because I've overridden `Parent.altered`, the `Child.altered` version runs, and line 9 executes like you'd expect.
2. In this case I want to do a before and after, so after line 9, I want to use `super` to get the `Parent.altered` version.
3. On line 10, I call `super()`, which is aware of inheritance and will get the `Parent` class for you.
4. At this point, the `Parent.altered` version of the function runs, and that prints out the `Parent` message.
5. The script then returns from the `Parent.altered`, and the `Child.altered` function continues to print out the after message.

If you run the script, you should see this:

Exercise 44c Session

```
$ ruby ex44c.rb
PARENT altered()
CHILD, BEFORE PARENT altered()
PARENT altered()
CHILD, AFTER PARENT altered()
```

All Three Combined

To demonstrate all of these, I have a final version that shows each kind of interaction from inheritance in one file:

[ex44d.rb](#)

```
1  class Parent
2
3      def override()
4          puts "PARENT override()"
5      end
6
7      def implicit()
8          puts "PARENT implicit()"
9      end
10
11     def altered()
12         puts "PARENT altered()"
13     end
14 end
15
16 class Child < Parent
17
18     def override()
19         puts "CHILD override()"
20     end
21
22     def altered()
23         puts "CHILD, BEFORE PARENT altered()"
24         super()
25         puts "CHILD, AFTER PARENT altered()"
26     end
27 end
28
29 dad = Parent.new()
30 son = Child.new()
31
32 dad.implicit()
33 son.implicit()
34
35 dad.override()
36 son.override()
37
38 dad.altered()
39 son.altered()
```

Go through each line of this code, and write a comment explaining what that line does and whether it's an override. Then run it and confirm you get what you expected:

Exercise 44d Session

```
$ ruby ex44d.rb
PARENT implicit()
PARENT implicit()
PARENT override()
CHILD override()
PARENT altered()
CHILD, BEFORE PARENT altered()
PARENT altered()
CHILD, AFTER PARENT altered()
```

Using super() with initialize

The most common use of `super()` is actually in `initialize` functions in base classes. This is usually the only place where you need to do some things in a child, then complete the initialization in the parent. Here's a quick example of doing that in the `Child` from these examples:

```
class Child < Parent
  def initialize(stuff)
    @stuff = stuff
    super()
  end
end
```

This is the same as the `Child.altered` example above, except I'm setting some variables in the `initialize` before having the `Parent.initialize` with its `Parent.initialize`.

Composition

Inheritance is useful, but another way to do the exact same thing is just to *use* other classes and modules, rather than rely on implicit inheritance. If you look at the three ways to exploit inheritance, two of the three involve writing new code to replace or alter functionality. This can easily be replicated by just calling functions in a module. Here's an example of doing this:

ex44e.rb

```
1 class Other
2
3   def override()
4     puts "OTHER override()"
5   end
6
7   def implicit()
8     puts "OTHER implicit()"
9   end
```

```
10
11     def altered()
12         puts "OTHER altered()"
13     end
14 end
15
16 class Child
17
18     def initialize()
19         @other = Other.new()
20     end
21
22     def implicit()
23         @other.implicit()
24     end
25
26     def override()
27         puts "CHILD override()"
28     end
29
30     def altered()
31         puts "CHILD, BEFORE OTHER altered()"
32         @other.altered()
33         puts "CHILD, AFTER OTHER altered()"
34     end
35 end
36
37 son = Child.new()
38
39 son.implicit()
40 son.override()
41 son.altered()
```

In this code I'm not using the name `Parent`, since there is *not* a parent-child `is-a` relationship. This is a `has-a` relationship, where `Child` has-a `Other` that it uses to get its work done. When I run this I get the following output:

Exercise 44e Session

```
$ ruby ex44e.rb
OTHER implicit()
CHILD override()
CHILD, BEFORE OTHER altered()
OTHER altered()
CHILD, AFTER OTHER altered()
```

You can see that most of the code in `Child` and `Other` is the same to accomplish the same thing. The only difference is that I had to define a `Child.implicit` function to do that one action. I could then ask myself if I need this `Other` to be a class, and could I just make it into a module named `other.rb`?

Ruby has another way to do composition using modules and a concept called mixins. You simply create a module with functions that are common to classes and then include them in your class similar to using a `require`. Here's this same composition example done using modules and mixins.

ex44f.rb

```
1  module Other
2
3      def override()
4          puts "OTHER override()"
5      end
6
7      def implicit()
8          puts "OTHER implicit()"
9      end
10
11     def Other.altered()
12         puts "OTHER altered()"
13     end
14 end
15
16 class Child
17     include Other
18
19     def override()
20         puts "CHILD override()"
21     end
22
23     def altered()
24         puts "CHILD, BEFORE OTHER altered()"
25         Other.altered()
26         puts "CHILD, AFTER OTHER altered()"
27     end
28 end
29
30 son = Child.new()
31
32 son.implicit()
33 son.override()
34 son.altered()
```

This is similar to the previous composition example. Mixins are much more powerful and an advanced topic I won't cover in this book.

When to Use Inheritance or Composition

The question of "inheritance versus composition" comes down to an attempt to solve the problem of reusable code. You don't want to have duplicated code all over your software, since that's not

clean and efficient. Inheritance solves this problem by creating a mechanism for including implied features in base classes. Composition solves this by giving you modules and the ability to call functions in other classes.

If both solutions solve the problem of reuse, then which one is appropriate in which situations? The answer is incredibly subjective, but I'll give you my three guidelines for when to use which:

1. Avoid something called "meta-programming" at all costs, as it is too complex to be reliably useful. If you're stuck with it, then be prepared to know the class hierarchy and spend time determining where everything is coming from.
2. Use composition to package code into modules that are used in many different unrelated places and situations.
3. Use inheritance only when there are clearly related reusable pieces of code that fit under a single common concept or if you have to because of something you're using.

Do not be a slave to these rules. The thing to remember about object-oriented programming is that it is entirely a social convention programmers have created to package and share code. Because it's a social convention, but one that's codified in Ruby, you may be forced to violate these rules because of the people you work with. In that case, find out how they use things and then just adapt to the situation.

Study Drills

There is only one Study Drill for this exercise because it is a big exercise. Go and read <https://github.com/bbatsov/ruby-style-guide> and start trying to use it in your code. You'll notice that some of it is different from what you've been learning in this book, but now you should be able to understand their recommendations and use them in your own code. The rest of the code in this book may or may not follow these guidelines depending on if it makes the code more confusing. I suggest you also do this, as comprehension is more important than impressing everyone with your knowledge of esoteric style rules.

Common Student Questions

How do I get better at solving problems that I haven't seen before?

The only way to get better at solving problems is to solve as many problems as you can *by yourself*. Typically people hit a difficult problem and then rush out to find an answer. This is fine when you have to get things done, but if you have the time to solve it yourself, then take that time. Stop and bang your head against the problem for as long as possible, trying every possible option, until you solve it or give up. After that, the answers you find will be more satisfying and you'll eventually get better at solving problems.

Aren't objects just copies of classes?

In some languages (like JavaScript) that is true. These are called prototype languages and there are not many differences between objects and classes other than usage. In Ruby, however, classes act as templates that "mint" new objects, similar to how coins are minted using a die (template).

You Make a Game

You need to start learning to feed yourself. Hopefully as you have worked through this book, you have learned that all the information you need is on the Internet—you just have to go search for it. The only thing you have been missing are the right words and what to look for when you search. Now you should have a sense of it, so it's about time you struggled through a big project and tried to get it working.

Here are your requirements:

1. Make a different game from the one I made.
2. Use more than one file, and use require to use them. Make sure you know what that is.
3. Use *one class per room* and give the classes names that fit their purpose (like GoldRoom, KoiPondRoom).
4. Your runner will need to know about these rooms, so make a class that runs them and knows about them. There's plenty of ways to do this, but consider having each room return which room is next or setting a variable of which room is next.

Other than that, I leave it to you. Spend a whole week on this exercise and make it the best game you can. Use classes, functions, dicts, arrays, and anything else you can to make it nice. The purpose of this lesson is to teach you how to structure classes that need other classes inside other files.

Remember, I'm not telling you *exactly* how to do this because you have to do this yourself. Go figure it out. Programming is problem solving, and that means trying things, experimenting, failing, scrapping your work, and trying again. When you get stuck, ask for help and show people your code. If they are mean to you, ignore them, focus on the people who are not mean, and offer to help. Keep working it and cleaning it until it's good, and then show it to some more people.

Good luck, and see you in a week with your game.

Evaluating Your Game

In this exercise, you will evaluate the game you just made. Maybe you got partway through it and then got stuck. Maybe you got it working, but just barely. Either way, we're going to go through a bunch of things you should know now and make sure you covered them in your game. We're going to study how to properly format a class, common conventions in using classes, and a lot of "textbook" knowledge.

Why would I have you try to do it yourself and then show you how to do it right? From now on in the book, I'm going to try to make you self-sufficient. I've mostly been holding your hand this whole time, and I can't do that for much longer. I'm now, instead, going to give you things to do, have you do them on your own, and then give you ways to improve what you did.

You will struggle at first and probably be very frustrated—but stick with it and eventually you will build a mind for solving problems. You will start to find creative solutions to problems rather than just copy solutions out of textbooks.

Function Style

All the other rules I've taught you about how to make a function nice apply here, but add these things:

- For various reasons, programmers call functions that are part of classes “methods.” It’s mostly marketing but just be warned that every time you say “function,” they’ll annoyingly correct you and say “method.” If they get too annoying, just ask them to demonstrate the mathematical basis that determines how a “method” is different from a “function” and they’ll shut up.
- When you work with classes, much of your time is spent talking about making the class “do things.” Instead of naming your functions after what the function does, instead name it as if it’s a command you are giving to the class. For example, `pop` is saying, “Hey array, pop this off.” It isn’t called `remove_from_end_of_list` because even though that’s what it does, that’s not a *command* to an array.
- Keep your functions small and simple. For some reason, when people start learning about classes they forget this.

Class Style

- Your class should use “camel case” like `SuperGoldFactory` rather than `super_gold_factory`.
- Try not to do too much in your `initialize` functions. It makes them harder to use.
- Your other functions should use “underscore format,” so write `my_awesome_hair` and not `myawesomехair` or `MyAwesomeHair`.
- Be consistent in how you organize your function arguments. If your class has to deal with users, dogs, and cats, keep that order throughout unless it really doesn’t make sense. If you have one function that takes `(dog, cat, user)` and another function that takes `(user, cat, dog)`, it’ll be hard to use.

- Try not to use variables that come from the module or globals. They should be fairly self-contained.
- A foolish consistency is the hobgoblin of little minds. Consistency is good, but foolishly following some idiotic mantra because everyone else does is bad style. Think for yourself.

Code Style

- Give your code vertical space so people can read it. You will find some very bad programmers who are able to write reasonable code but who do not add *any* spaces. This is bad style in any language because the human eye and brain use space and vertical alignment to scan and separate visual elements. Not having space is the same as giving your code an awesome camouflage paint job.
- If you can't read it out loud, it's probably hard to read. If you are having a problem making something easy to use, try reading it out loud. Not only does this force you to slow down and really read the code, but it also helps you find difficult passages and things to change for readability.
- Try to do what other people are doing in Ruby until you find your own style.
- Once you find your own style, do not be a jerk about it. Working with other people's code is part of being a programmer, and other people have really bad taste. Trust me, you will probably have really bad taste, too, and not even realize it.
- If you find someone who writes code in a style you like, try writing something that mimics that style.

Good Comments

- Programmers will tell you that your code should be readable enough that you do not need comments. They'll then tell you in their most official-sounding voice, "Ergo, one should never write comments or documentation. QED." Those programmers are either consultants who get paid more if other people can't use their code, or incompetents who tend to never work with other people. Ignore them and write comments.
- When you write comments, describe *why* you are doing what you are doing. The code already says how, but why you did things the way you did is more important.
- When you write doc comments for your functions, make the comments documentation for someone who will have to use your code. You do not have to go crazy, but a nice little sentence about what someone can do with that function helps a lot.

- While comments are good, too many are bad, and you have to maintain them. Keep your comments relatively short and to the point, and if you change a function, review the comment to make sure it's still correct.

Evaluate Your Game

I want you to now pretend you are me. Adopt a very stern look, print out your code, and take a red pen and mark every mistake you find, including anything from this exercise and from other guidelines you've read so far. Once you are done marking your code up, I want you to fix everything you came up with. Then repeat this a couple of times, looking for anything that could be better. Use all the tricks I've given you to break your code down into the smallest, tiniest little analysis you can.

The purpose of this exercise is to train your attention to detail on classes. Once you are done with this bit of code, find someone else's code and do the same thing. Go through a printed copy of some part of it and point out all the mistakes and style errors you find. Then fix it and see if your fixes can be made without breaking that program.

I want you to do nothing but evaluate and fix code for the week—your own code and other people's. It'll be pretty hard work, but when you are done your brain will be wired tight like a boxer's hands.

A Project Skeleton

This exercise is where you start learning how to set up a good project “skeleton” directory. This skeleton directory will hold all the basics you need to get a new project up and running. It will hold your project layout, automated tests, modules, and install scripts. When you go to make a new project, just copy this directory to a new name and edit the files to get started.

Creating the Skeleton Project Directory

First, create the structure of your skeleton directory with these commands:

```
$ mkdir projects
$ cd projects
$ mkdir skeleton
$ cd skeleton
$ mkdir bin data doc ext lib tests lib/NAME
```

I use a directory named `projects` to store all the various things I’m working on. Inside that directory I have my `skeleton` directory, where I put the basis of my projects.

Next we need to set up some initial files. Here’s how you do that on Linux/OS X:

```
$ touch bin/NAME
$ touch lib/NAME.rb
```

Here’s the same thing on Windows PowerShell:

```
$ new-item -type file bin/NAME
$ new-item -type file lib/NAME.rb
```

That creates an empty Ruby module directory we can put our code into. Then we need to create a sample `gemspec` file named `NAME.gemspec` that will describe our project:

`NAME.gemspec`

```
# coding: utf-8
lib = File.expand_path('../lib', __FILE__)
$LOAD_PATH.unshift(lib) unless $LOAD_PATH.include?(lib)

Gem::Specification.new do |spec|
  spec.name          = "NAME"
  spec.version       = '1.0'
  spec.authors       = ["Your Name Here"]
  spec.email         = ["youremail@yourdomain.com"]
  spec.summary        = %q{Short summary of your project}
```

```
spec.description = %q{Longer description of your project.}
spec.homepage = "http://domainforproject.com/"
spec.license = "MIT"

spec.files = ['lib/NAME.rb']
spec.executables = ['bin/NAME']
spec.test_files = ['tests/test_NAME.rb']
spec.require_paths = ["lib"]
end
```

Edit this file so that it has your contact information and is ready to go when you copy it.

Next you want to create a `Rakefile` file that will help you automate common tasks when working with Ruby. Write this code for your `Rakefile` to make it run your tests for you:

Rakefile

```
1 require 'rake/testtask'
2
3 Rake::TestTask.new do |t|
4   t.libs << "tests"
5   t.test_files = FileList['tests/test*.rb']
6   t.verbose = true
7 end
```

Finally you will want a simple skeleton file for tests named `tests/test_NAME.rb`:

test_NAME.rb

```
1 require "./lib/NAME.rb"
2 require "test/unit"
3
4 class TestNAME < Test::Unit::TestCase
5
6   def test_sample
7     assert_equal(4, 2+2)
8   end
9
10 end
```

Final Directory Structure

When you are done setting all this up, your directory should look like mine here:

```
skeleton/
  NAME.gemspec
  Rakefile
  data
  ext/
  tests/
```

```
bin/
    NAME
doc/
lib/
    NAME
    NAME.rb
lib/NAME
tests/
    test_NAME.rb
```

From now on, you should run your commands from that work with this directory. If you can't do `ls -R` and see this same structure, then you are in the wrong place. For example, people commonly go into the `tests/` directory and try to run files there, which won't work. To run your application's tests, you would need to be *above* `tests/` in the hierarchy of the directory. So, if you try this:

```
$ cd tests/  # WRONG! WRONG! WRONG!
$ rake test
```

Then that is *wrong!* You have to be *above* `tests`, in the `skeleton/` directory itself, not in directories below `skeleton/`. If you made this mistake, just do this:

```
$ cd .. # get out of tests/
$ ls      # CORRECT! ~ you are now in the right spot
NAME.gemspec      data          ext          tests
bin              doc           lib
$ rake test
Loaded suite tests/test_NAME
Started
.
Finished in 0.000226 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
```

Testing Your Setup

After you get all this installed, you should be able to do this:

```
$ rake test
# A bunch of load junk here, ignore it.
Loaded suite tests/test_NAME
Started
.
Finished in 0.000226 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
```

I'll explain Ruby's `Test::Unit` in the next exercise, but for now if you don't see this output or get syntax errors, then you made a mistake in one of the files or you are in the wrong place.

Using the Skeleton

You are now done with most of your yak shaving. Whenever you want to start a new project, just do this:

1. Make a copy of your skeleton directory. Name it after your new project.
2. Rename (move) the `lib/NAME.rb` file and `lib/NAME/` directory to be the name of your project or whatever you want to call your root module.
3. Edit your `Gemspec` to have all the information for your project.
4. Rename `tests/test_NAME.rb` to match your module name.
5. Double check it's all working by using `ruby tests/test_yourproject.rb` again.
6. Start coding.

Required Quiz

This exercise doesn't have Study Drills but has a quiz you should complete:

1. Read about how to use all of the things you installed.
2. Read about the `NAME.gemspec` file, commonly called a "Gemspec," and all it has to offer.
3. Make a project, put code into the module, and then get the module working. This means you have to change all the files, directories, and modules with `NAME` in them to the name of your project. If you get stuck, watch the video for this exercise to see how I did it.
4. Put a script in the `bin` directory that you can run. Read about how you can make a Ruby script that's runnable for your system.
5. Mention the `bin` script you created in your `Gemspec` so that it gets installed.
6. Use your `Gemspec` to install your own module and make sure it works, then use `gem` to uninstall it.

Common Student Questions

Do these instructions work on Windows?

They should, but depending on the version of Windows, you may need to struggle with the setup a bit to get it working. Just keep researching and trying it until you get it, or see if you can ask a more experienced Ruby+Windows friend to help out.

Why do we need a `bin/` folder?

This is just a standard place to put scripts that are run on the command line, not a place to put modules.

Do you have a real-world example project?

There are many projects written in Ruby that do this, but try this simple one I created <https://rubygems.org/gems/mongrel> with source at <https://github.com/mongrel/mongrel> for you to browse.

My `ruby tests/test_NAME.rb` run shows only one test being run. Is that right?

Yes, that's what my output shows too.

This page intentionally left blank

Automated Testing

Having to type commands into your game over and over to make sure it's working is annoying. Wouldn't it be better to write little pieces of code that test your code? Then when you make a change, or add a new thing to your program, you just "run your tests" and the tests make sure things are still working. These automated tests won't catch all your bugs, but they will cut down on the time you spend repeatedly typing and running your code.

Every exercise after this one will not have a *What You Should See* section, but instead will have a *What You Should Test* section. You will be writing automated tests for all of your code starting now, and this will hopefully make you an even better programmer.

I won't try to explain why you should write automated tests. I will only say that you are trying to be a programmer, and programmers automate boring and tedious tasks. Testing a piece of software is definitely boring and tedious, so you might as well write a little bit of code to do it for you.

That should be all the explanation you need because *your* reason for writing unit tests is to make your brain stronger. You have gone through this book writing code to do things. Now you are going to take the next leap and write code that knows about other code you have written. This process of writing a test that runs some code you have written *forces* you to understand clearly what you have just written. It solidifies in your brain exactly what the code does and why it works and gives you a new level of attention to detail.

Writing a Test Case

We're going to take a very simple piece of code and write one simple test. We're going to base this little test on a new project from your project skeleton.

First, make a ex47 project from your project skeleton. Here are the steps you would take. I'm going to give these instructions in English rather than show you how to type them so that *you* have to figure it out.

1. Copy skeleton to ex47.
2. Rename everything with NAME to ex47.
3. Change the word NAME in all the files to ex47.

Here's me doing it:

Exercise 47setup Session

```
$ cp -r skeleton ex47
$ cd ex47
$ ls
NAME.gemspec      bin          doc          lib
Rakefile          data         ext          tests
$ mv NAME.gemspec ex47.gemspec
$ mv bin/NAME bin/ex47
$ mv tests/test_NAME.rb tests/test_ex47.rb
$ mv lib/NAME lib/ex47
$ mv lib/NAME.rb lib/ex47.rb
$ find . -name "*NAME*" -print
$
```

That last command will find any other files with "NAME" in them so you can change them if you missed some. Refer back to Exercise 46 if you get stuck. If you can't do this easily, then maybe practice it a few times.

Next, create a simple file lib/ex47/game.rb where you can put the code to test. This will be a very silly little class that we want to test with this code in it:

game.rb

```
1  class Room
2
3    def initialize(name, description)
4      @name = name
5      @description = description
6      @paths = {}
7    end
8
9    # these make it easy for you to access the variables
10   attr_reader :name
11   attr_reader :paths
12   attr_reader :description
13
14   def go(direction)
15     return @paths[direction]
16   end
17
18   def add_paths(paths)
19     @paths.update(paths)
20   end
21
22 end
```

Once you have that file, change the unit test in `tests/test_ex47.rb` to this:

`test_ex47.rb`

```
1  require "ex47/game.rb"
2  require "test/unit"
3
4  class TestGame < Test::Unit::TestCase
5
6      def test_room()
7          gold = Room.new("GoldRoom",
8              """This room has gold in it you can grab. There's a
9              door to the north."""")
10         assert_equal(gold.name, "GoldRoom")
11         assert_equal(gold.paths, {})
12     end
13
14     def test_room_paths()
15         center = Room.new("Center", "Test room in the center.")
16         north = Room.new("North", "Test room in the north.")
17         south = Room.new("South", "Test room in the south.")
18
19         center.add_paths({'north'=> north, 'south'=> south})
20         assert_equal(center.go('north'), north)
21         assert_equal(center.go('south'), south)
22
23     end
24
25     def test_map()
26         start = Room.new("Start", "You can go west and down a hole.")
27         west = Room.new("Trees", "There are trees here, you can go east.")
28         down = Room.new("Dungeon", "It's dark down here, you can go up.")
29
30         start.add_paths({'west'=> west, 'down'=> down})
31         west.add_paths({'east'=> start})
32         down.add_paths({'up'=> start})
33
34         assert_equal(start.go('west'), west)
35         assert_equal(start.go('west').go('east'), start)
36         assert_equal(start.go('down').go('up'), start)
37     end
38 end
```

This file imports the `Room` class you made in the `ex47/game.rb` file so that you can do tests on it. There is then a set of tests that are functions starting with `test_`. Inside each test case there's a bit of code that makes a room or a set of rooms, and then makes sure the rooms work the way you expect them to work. It tests out the basic room features, then the paths, then tries out a whole map.

The important function here is `assert_equal`, which makes sure that variables you have set or paths you have built in a Room are actually what you think they are. If you get the wrong result, then `rake test` (actually `Test::Unit`) will print out an error message so you can figure it out.

Testing Guidelines

Follow this general set of guidelines when making your tests:

1. Test files go in `tests/` and must be named `test_BLAH.rb`; otherwise, `rake test` won't run them. This also keeps your tests from clashing with your other code.
2. Write one test file for each module you make.
3. Keep your test cases (functions) short, but do not worry if they are a bit messy. Test cases are usually kind of messy.
4. Even though test cases are messy, try to keep them clean and remove any repetitive code you can. Create helper functions that get rid of duplicate code. You will thank me later when you make a change and then have to change your tests. Duplicated code will make changing your tests more difficult.
5. Do not get too attached to your tests. Sometimes, the best way to redesign something is to just delete it and start over.

What You Should See

Exercise 47 Session

```
$ rake test
(in /Users/zedshaw/projects/books/learn-pyrb-the-hard-way/ruby/ex47)
/System/Library/Frameworks/Ruby.framework/Versions/1.8/usr/bin/ruby-I
"lib:tests" "/System/Library/Frameworks/Ruby.framework/Versions/1.8/
usr/lib/ruby/1.8/rake/rake_test_loader.rb" "tests/test_ex47.rb"
Loaded suite /System/Library/Frameworks/Ruby.framework/Versions/1.8/
usr/lib/ruby/1.8/rake/rake_test_loader
Started
...
Finished in 0.000398 seconds.

3 tests, 7 assertions, 0 failures, 0 errors
```

That's what you should see if everything is working right. Try causing an error to see what that looks like and then fix it.

Study Drills

1. Go read about Ruby's Test::Unit more, and also read about alternatives.
2. Write a new test case in tests/test_ex47.rb that creates a miniature version of your game from Exercise 45. This is one function that is similar to the current functions, but using your game's room names and abbreviated descriptions. Remember to use Room.add_paths to create the map, and use assertions to confirm everything works as expected.

Common Student Questions

I get a syntax error when I run `rake test`.

If you get this error then look at what the error says and fix that line of code or the ones above it. Tools like Test::Unit (which `rake test` runs) are running your code and the test code, so they will find syntax errors the same as running Ruby will.

I can't import `ex47/game.rb`.

Make sure you create the `Rakefile` file as specified in Exercise 46. You most likely got this wrong. You must also *not* be in the `tests` directory. Again, refer back to Exercise 46.

This page intentionally left blank

Advanced User Input

In past games you handled the user's input by simply expecting set strings. If the user typed "run" and exactly "run," then the game worked. If they typed in a similar phrase like "run fast," it would fail. What we need is a device that lets users type phrases in various ways and then converts that input into something the computer understands. For example, we'd like to have all of these phrases work the same way:

- open door
- open the door
- go THROUGH the door
- punch bear
- Punch The Bear in the FACE

It should be alright for a user to write something a lot like English for your game, and have your game figure out what it means. In this exercise, we're going to write a module that does just that. This module will have a few classes that work together to handle user input and convert it into something your game can work with reliably.

A simple version of English has the following elements:

- Words separated by spaces.
- Sentences composed of the words.
- Grammar that structures the sentences into meaning.

That means the best place to start is figuring out how to get words from the user and which kinds of words those are.

Our Game Lexicon

In our game we have to create a list of allowed words called a "lexicon":

- Direction words: north, south, east, west, down, up, left, right, back
- Verbs: go, stop, kill, eat
- Stop words: the, in, of, from, at, it

- Nouns: door, bear, princess, cabinet
- Numbers: any string of 0 through 9 characters

When we get to nouns, we have a slight problem since each room could have a different set of nouns, but let's just pick this small set to work with for now and improve it later.

Breaking Up a Sentence

Once we have our lexicon we need a way to break up sentences so that we can figure out what they are. In our case, we've defined a sentence as "words separated by spaces," so we really just need to do this:

```
stuff = $stdin.gets.chomp
words = stuff.split
```

That's all we'll worry about for now, but this will work really well for quite a while.

Lexicon Tuples

Once we know how to break up a sentence into words, we just have to go through the list of words and figure out what "type" they are. To do that, we're going to use a simple array that has just two elements, and create a new array that contains many of these. Here's an example of building one by hand:

```
first_word = ['verb', 'go']
second_word = ['direction', 'north']
third_word = ['direction', 'west']
sentence = [first_word, second_word, third_word]
```

This creates a pair [TYPE, WORD] that lets you look at the word and do things with it.

This is just an example, but that's basically the end result. You want to take raw input from the user, carve it into words with `split`, analyze those words to identify their type, and finally make a sentence out of them.

Scanning Input

Now you are ready to write your scanner. This scanner will take a string of raw input from a user and return a sentence that's composed of an array of arrays with the (TOKEN, WORD) pairings. If a word isn't part of the lexicon, then it should still return the WORD but set the TOKEN to an error token. These error tokens will tell users they messed up.

Here's where it gets fun. I'm not going to tell you how to do this. Instead I'm going to write a "unit test" and you are going to write the scanner so that the unit test works.

Exceptions and Numbers

There is one tiny thing I will help you with first, and that's converting numbers. In order to do this though, we're going to cheat and use exceptions. An exception is an error message that you get from some function you may have run. What happens is your function "raises" an exception when it encounters an error, and then you have to handle that exception. For example, if you type this into Ruby:

Exercise 48 Ruby Session

```
>> Integer("hell")
ArgumentError: invalid value for Integer(): "hell"
  from (irb):1:in 'Integer'
  from (irb):1
  from /usr/local/bin/irb:11:in '<main>'
```

That ArgumentError is an exception that the Integer() function threw because what you handed Integer() is not a number. The Integer() function could have returned a value to tell you it had an error, but since it only returns integers, it would have a hard time doing that. It can't return -1 since that's a number. Instead of trying to figure out what to return when there's an error, the Integer() function raises the ArgumentError exception and then you deal with it.

You deal with an exception by using the begin and rescue keywords:

ex48_convert.rb

```
1  def convert_number(object)
2    begin
3      return Integer(object)
4    rescue
5      return nil
6    end
7  end
```

You put the code you want to "try" inside the begin block, and then you put the code to run for the error inside the rescue. In this case, we want to "try" to call Integer() on something that might be a number. If that has an error, then we "catch" it and return nil.

In the scanner that you write, you should use this function to test if something is a number. You should also do it as the last thing you check for before declaring that word to be an error word.

A Test First Challenge

Test first is a programming tactic where you write an automated test that pretends the code works, *then* you write the code to make the test actually work. This method works when you can't visualize how the code is implemented, but you can imagine how you have to work with it. For example, if you know how you need to use a new class in another module, but you don't quite know how to implement that class yet, then write the test first.

You are going to take a test I give you and use it to write the code that makes it work. To do this exercise, you'll follow this procedure:

1. Create one small part of the test I give you.
2. Make sure it runs and *fails* so you know that the test is actually confirming a feature works.
3. Go to your source file `lexicon.rb` and write the code that makes this test pass.
4. Repeat until you have implemented everything in the test.

When you get to 3 it's also a good idea to combine our other method of writing code:

1. Make the "skeleton" function or class that you need.
2. Write comments inside describing how that function works.
3. Write the code that does what the comments describe.
4. Remove any comments that just repeat the code.

This method of writing code is called "pseudocode" and works well if you don't know how to implement something, but you can describe it in your own words.

Combining the "test first" with the "pseudocode" tactics, we have this simple process for programming:

1. Write a bit of test that fails.
2. Write the skeleton function/module/class the test needs.
3. Fill the skeleton with comments in your own words explaining how it works.
4. Replace the comments with code until the test passes.
5. Repeat.

In this exercise you will practice this method of working by making a test I give you run against the `lexicon.rb` module.

What You Should Test

Here is the test case `tests/test_lexicon.rb` that you should use, but *don't type this in yet!*

`lexicon_tests.rb`

```
1  require 'ex48/lexicon.rb'  
2  require "test/unit"  
3
```

```
4  class TestNAME < Test::Unit::TestCase
5  def test_directions()
6      assert_equal(Lexicon.scan("north"), [['direction', 'north']])
7      result = Lexicon.scan("north south east")
8
9      assert_equal(result, [['direction', 'north'],
10                  ['direction', 'south'],
11                  ['direction', 'east']])
12  end
13
14  def test_verbs()
15      assert_equal(Lexicon.scan("go"), [['verb', 'go']])
16      result = Lexicon.scan("go kill eat")
17      assert_equal(result, [['verb', 'go'],
18                      ['verb', 'kill'],
19                      ['verb', 'eat']])
20  end
21
22
23  def test_stops()
24      assert_equal(Lexicon.scan("the"), [['stop', 'the']])
25      result = Lexicon.scan("the in of")
26      assert_equal(result, [['stop', 'the'],
27                      ['stop', 'in'],
28                      ['stop', 'of']])
29  end
30
31
32  def test_nouns()
33      assert_equal(Lexicon.scan("bear"), [['noun', 'bear']])
34      result = Lexicon.scan("bear princess")
35      assert_equal(result, [['noun', 'bear'],
36                      ['noun', 'princess']])
37  end
38
39  def test_numbers()
40      assert_equal(Lexicon.scan("1234"), [['number', 1234]])
41      result = Lexicon.scan("3 91234")
42      assert_equal(result, [['number', 3],
43                      ['number', 91234]])
44  end
45
46
47  def test_errors()
48      assert_equal(Lexicon.scan("ASDFADFASDF"), [['error', 'ASDFADFASDF']])
49      result = Lexicon.scan("bear IAS princess")
50      assert_equal(result, [['noun', 'bear'],
51                      ['error', 'IAS'],
```

```
52      ['noun', 'princess']])
53  end
54
55  end
```

You will want to create a new project using the project skeleton just like you did in Exercise 47. Then you'll need to create this test case and the `lexicon.rb` file it will use. Look at the top of the test case to see how it's being required to figure out where it goes.

Next, follow the procedure I gave you earlier and write a little bit of the test case at a time. For example, here's how I'd do it:

1. Write the require at the top. Get that to work.
2. Create an empty version of the first test case `test_directions`. Make sure that runs.
3. Write the first line of the `test_directions` test case. Make it fail.
4. Go to the `lexicon.rb` file, and create an empty `scan` function.
5. Run the test and make sure `scan` is at least running, even though it fails.
6. Fill in pseudocode comments for how `scan` should work to make `test_directions` pass.
7. Write the code that matches the comments until `test_directions` passes.
8. Go back to `test_directions` and write the rest of the lines.
9. Go back to `scan` in `lexicon.rb` and work on it to make this new test code pass.
10. Once you've done that, you have your first passing test, and you move on to the next test.

As long as you keep following this procedure one little chunk at a time, you can successfully turn a large problem into smaller solvable problems. It's like climbing a mountain by turning it into a bunch of little hills.

Study Drills

1. Improve the unit test to make sure you test more of the lexicon.
2. Add more words to the lexicon and then update the unit test.
3. Make sure your scanner handles user input in any capitalization and case. Update the test to make sure this actually works.
4. Find another way to convert the number.
5. My solution was 37 lines long. Is yours longer? Shorter?

Common Student Questions

What's the difference between `begin-rescue` and `if-else`?

The `begin-rescue` construct is only used for handling exceptions that modules can throw. It should never be used as an alternative to `if-else`.

Is there a way to keep the game running while the user is waiting to type?

I'm assuming you want to have a monster attack users if they don't react quickly enough. It is possible but it involves modules and techniques that are outside of this book's domain.

This page intentionally left blank

Making Sentences

What we should be able to get from our little game lexicon scanner is an array that looks like this:

Exercise 49 Ruby Session

```
>> require './ex48/lexicon.rb'
=> true
=> Lexicon.scan("go north")
=> [["verb", "go"], ["direction", "north"]]
=> Lexicon.scan("kill the princess")
=> [["verb", "kill"], ["stop", "the"], ["noun", "princess"]]
=> Lexicon.scan("eat the bear")
=> [["verb", "eat"], ["stop", "the"], ["noun", "bear"]]
=> Lexicon.scan("open the door and smack the bear in the nose")
=> [["error", "open"], ["stop", "the"], ["error", "door"], ["error", "and"],
     ["error", "smack"], ["stop", "the"], ["noun", "bear"], ["stop", "in"],
     ["stop", "the"], ["error", "nose"]]
```

Now let us turn this into something the game can work with, which would be some kind of Sentence class.

If you remember from grade school, a sentence can be a simple structure like:

Subject Verb Object

Obviously, grammar gets more complex than that, and you probably did many days of annoying sentence graphs for English class. What we want is to turn the above arrays of tuples into a nice Sentence object that has a subject, verb, and object.

Match and Peek

To do this we need four tools:

1. A way to loop through the array of scanned words. That's easy.
2. A way to "match" different types of tuples that we expect in our Subject Verb Object setup.
3. A way to "peek" at a potential tuple so we can make some decisions.
4. A way to "skip" things we do not care about, like stop words.
5. A Sentence object to put the results in.

We will be putting these functions in a file named `./lib/ex48/parser.rb` in order to test it. We use the `peek` function to say look at the next element in our tuple array, and then match to take one off and work with it.

The Sentence Grammar

Before you can write the code you need to understand how basic English sentence grammar works. In our parser we want to produce a `Sentence` object that has three attributes:

Sentence.subject This is the subject of any sentence, but could default to "player" most of the time since a sentence of, "run north" is implying "player run north." This will be a noun.

Sentence.verb This is the action of the sentence. In "run north" it would be "run." This will be a verb.

Sentence.object This is another noun that refers to what the verb acts on. In our game we separate out directions, which would also be objects. In "run north," the word "north" would be the object. In "hit bear," the word "bear" would be the object.

Our parser then has to use the functions we described and, given a scanned sentence, convert it into an array of `Sentence` objects to match the input.

A Word on Exceptions

You briefly learned about exceptions, but not how to raise them. This code demonstrates how to do that with the `ParserError` at the top. Notice that it uses classes to give it the type of Exception. Also notice the use of the `raise` keyword to raise the exception.

In your tests, you will want to work with these exceptions, which I'll show you how to do.

The Parser Code

If you want an extra challenge, stop right now and try to write the parser code based on just my description. If you get stuck, you can come back and see how I did it, but trying to implement the parser yourself is good practice. I will now walk through the code so you can enter it into your `ex48/parser.rb`. We start the parser with the exception we need for a parsing error:

`parser.rb`

```
1 class ParserError < Exception
2 end
```

This is how you make your own `ParserError` exception class you can throw. Next we need the `Sentence` object we'll create:

parser.rb

```
1  class Sentence
2
3      def initialize(subject, verb, obj)
4          # remember we take ['noun', 'princess'] pairs and convert them
5          @subject = subject[1]
6          @verb = verb[1]
7          @object = obj[1]
8      end
9
10     attr_reader :subject
11     attr_reader :verb
12     attr_reader :object
13 end
```

There's nothing special about this code so far. You're just making simple classes.

In our description of the problem, we need a function that can peek at a list of words and return what type of word it is:

parser.rb

```
1  def peek(word_list)
2      if word_list
3          word = word_list[0]
4          return word[0]
5      else
6          return nil
7      end
8  end
```

We need this function because we'll have to make decisions about which kind of sentence we're dealing with based on what the next word is; then we can call another function to consume that word and carry on.

To consume a word we use the `match` function, which confirms that the expected word is the right type, takes it off the list, and returns the word.

parser.rb

```
1  def match(word_list, expecting)
2      if word_list
3          word = word_list.shift
4
5          if word[0] == expecting
6              return word
7          else
```

```
8         return nil
9     end
10    else
11        return nil
12    end
13 end
```

Again, this code is fairly simple—but make sure you understand not only this code but also *why* I'm doing it this way. I need to peek at words in the list to decide which kind of sentence I'm dealing with, and then I need to match those words to create my Sentence.

The last thing I need is a way to skip words that aren't useful to the Sentence. These are the words labeled "stop words" (type 'stop'): words like "the," "and," and "a."

parser.rb

```
1 def skip(word_list, word_type)
2     while peek(word_list) == word_type
3         match(word_list, word_type)
4     end
5 end
```

Remember that skip doesn't skip just one word, it skips as many words of that type as it finds. For example, if someone types "scream at the bear," you get "scream" and "bear,"

That's our basic set of parsing functions, and with them we can actually parse just about any text we want. Our parser is very simple, though, so the remaining functions are short.

First we can handle parsing a verb:

parser.rb

```
1 def parse_verb(word_list)
2     skip(word_list, 'stop')
3
4     if peek(word_list) == 'verb'
5         return match(word_list, 'verb')
6     else
7         raise ParserError.new("Expected a verb next.")
8     end
9 end
```

We skip any stop words, then peek ahead to make sure the next word is a "verb" type. If it's not, then we raise the ParserError to say why. If it is a "verb," then we match it, which takes it off the list. A similar function handles sentence objects:

parser.rb

```
1 def parse_object(word_list)
2     skip(word_list, 'stop')
3     next_word = peek(word_list)
4
```

```
5      if next_word == 'noun'
6          return match(word_list, 'noun')
7      elsif next_word == 'direction'
8          return match(word_list, 'direction')
9      else
10         raise ParserError.new("Expected a noun or direction next.")
11     end
12   end
```

Again, skip the stop words, peek ahead, and decide if the sentence is correct based on what's there. In the `parse_object` function, though, we need to handle both "noun" and "direction" words as possible objects. Subjects are then similar again, but since we want to handle the implied "player" noun, we have to use peek:

parser.rb

```
1  def parse_subject(word_list)
2      skip(word_list, 'stop')
3      next_word = peek(word_list)
4
5      if next_word == 'noun'
6          return match(word_list, 'noun')
7      elsif next_word == 'verb'
8          return ['noun', 'player']
9      else
10         raise ParserError.new("Expected a verb next.")
11     end
12   end
```

With that all out of the way and ready, our final `parse_sentence` function is very simple:

parser.rb

```
1  def parse_sentence(word_list)
2      subj = parse_subject(word_list)
3      verb = parse_verb(word_list)
4      obj = parse_object(word_list)
5
6      return Sentence.new(subj, verb, obj)
7  end
```

Playing with the Parser

To see how this works, you can play with it like this:

Exercise 49a Ruby Session

```
>> require './ex48/parser.rb'
=> true
>> x = parse_sentence(['verb', 'run'], ['direction', 'north'])
=> #<Sentence:0x007fd7a30b6df0 @subject="player", @verb="run", @object="north">
>> x.subject
```

```
=> "player"
=> x.verb
=> "run"
=> x.object
=> "north"
=> x = parse_sentence([['noun', 'bear'], ['verb', 'eat'],
?=> ['stop', 'the'], ['noun', 'honey']])
=> #<Sentence:0x007fd7a30869e8 @subject="bear", @verb="eat", @object="honey">
=> x.subject
=> "bear"
=> x.verb
=> "eat"
=> x.object
=> "honey"
```

What You Should Test

For Exercise 49, write a complete test that confirms everything in this code is working. Put the test in `tests/test_parser.rb`, similar to the test file from the last exercise. That includes making exceptions happen by giving the parser bad sentences.

Check for an exception by using the function `assert_raise` from the `Test::Unit` documentation. Learn how to use this function so you can write a test that is *expected* to fail, which is very important in testing. Learn about this function (and others) by reading the `Test::Unit` documentation.

When you are done, you should know how this bit of code works and understand how to write a test for other people's code even if they do not want you to. Trust me, it's a very handy skill to have.

Study Drills

1. Change the `parse_` methods and try to put them into a class rather than have them be just methods. Which design do you like better?
2. Make the parser more error-resistant so that you can avoid annoying your users if they type words that your lexicon doesn't understand.
3. Improve the grammar by handling more things like numbers.
4. Think about how you might use this `Sentence` class in your game to do more fun things with a user's input.

Common Student Question

I can't seem to make `assert_raise` work right.

Look at the example in the Ruby Documentation at http://www.ruby-doc.org/stdlib-2.1.2/libdoc/test/unit/rdoc/Test/Unit/Assertions.html#method-i-assert_raise

Your First Website

These final three exercises will be very hard and you should take your time with them. In this first one you'll build a simple web version of one of your games. Before you attempt this exercise, you *must* have completed Exercise 46 successfully and have a working gem installed such that you can install packages and know how to make a skeleton project directory. If you don't remember how to do this, go back to Exercise 46 and do it all over again.

Installing Sinatra

Before creating your first web application, you'll first need to install the "web framework" called Sinatra. The term "framework" generally means "some package that makes it easier for me to do something." In the world of web applications, people create "web frameworks" to compensate for the difficult problems they've encountered when making their own sites. They share these common solutions in the form of a package you can download to bootstrap your own projects.

In our case, we'll be using the Sinatra framework, but there are many, many, *many* others you can choose from. For now, learn Sinatra, then branch out to another one when you're ready (or just keep using Sinatra since it's good enough).

Using gem, install Sinatra:

```
$ sudo gem install sinatra
Password:
Fetching: rack-1.5.2.gem (100%)
Successfully installed rack-1.5.2
Fetching: tilt-1.4.1.gem (100%)
Successfully installed tilt-1.4.1
Fetching: rack-protection-1.5.3.gem (100%)
Successfully installed rack-protection-1.5.3
Fetching: sinatra-1.4.5.gem (100%)
Successfully installed sinatra-1.4.5
Parsing documentation for rack-1.5.2
Installing ri documentation for rack-1.5.2
Parsing documentation for rack-protection-1.5.3
Installing ri documentation for rack-protection-1.5.3
Parsing documentation for sinatra-1.4.5
Installing ri documentation for sinatra-1.4.5
Parsing documentation for tilt-1.4.1
Installing ri documentation for tilt-1.4.1
Done installing documentation for rack, rack-protection, sinatra, tilt after
 303 seconds
4 gems installed
$
```

This will work on Linux and Mac OS X computers. On Windows, just drop the sudo part of the gem install command and it should work. If not, go back to Exercise 46 and make sure you can do it reliably. You may also see different versions and names of packages installed but should not see any errors.

Make a Simple “Hello World” Project

Now you’re going to make an initial, very simple “Hello World” web application and project directory using Sinatra. First, make your project directory based on the skeleton you have:

```
$ cd projects
$ cp -r skeleton gothonweb
$ cd gothonweb
$ mv bin/NAME bin/app.rb
$ mv lib/NAME lib/gothonweb
$ mv lib/NAME.rb lib/gothonweb.rb
$ mv NAME.gemspec gothonweb.gemspec
$ mv tests/test_NAME.rb tests/test_gothonweb.rb
$ rm tests/NAME_tests.rb
$ mkdir views
$ mkdir static
```

You’ll be taking the game from Exercise 43 and turning it into a web application, so that’s why you’re calling it gothonweb. Before you do that, we need to create the most basic Sinatra application possible. Put the following code into bin/app.rb:

app.rb

```
1  require 'sinatra'
2
3  set :port, 8080
4  set :static, true
5  set :public_folder, "static"
6  set :views, "views"
7
8  get '/' do
9    return 'Hello world'
10 end
11
12 get '/hello/' do
13   greeting = params[:greeting] || "Hi There"
14   erb :index, :locals => {'greeting' => greeting}
15 end
```

Then run the application like this:

```
$ ruby bin/app.rb
[2014-07-03 16:02:44] INFO  WEBrick 1.3.1
```

```
[2014-07-03 16:02:44] INFO  ruby 2.1.2 (2014-05-08) [x86_64-darwin11.0]
== Sinatra/1.4.5 has taken the stage on 8080 for development with backup
   from WEBrick
[2014-07-03 16:02:44] INFO  WEBrick::HTTPServer#start: pid=20305 port=8080
```

The output you see is Sinatra running your little “Hello World” application. However, if you did this:

```
$ cd bin/  # WRONG! WRONG! WRONG!
$ ruby bin/app.rb # WRONG! WRONG! WRONG!
```

Then you are doing it *wrong*. In all Ruby projects, you do not cd into a lower directory to run things. You stay at the top level and run everything from there so that all of the system can access all the modules and files. Go reread Exercise 46 to understand a project layout and how to use it if you did this.

Finally, use your web browser and go to <http://localhost:8080> and you should see two things. First, in your browser you’ll see Hello, world!. Second, you’ll see your Terminal with new output like this:

```
::1 -- [03/Jul/2014:16:07:10] "GET / HTTP/1.1" 200 11 0.0136
localhost -- [03/Jul/2014:16:07:10 PDT] "GET / HTTP/1.1" 200 11
- -> /
```

Those are log messages that Sinatra prints out so you can see whether the server is working, and what the browser is doing behind the scenes. The log messages help you debug your scripts and figure out when you have problems.

I haven’t explained the way *any* of this web stuff works yet, because I want to get you setup and ready to roll so that I can explain it better in the next two exercises. To accomplish this, I’ll have you break your Sinatra application in various ways and then restructure it so that you know how it’s set up.

What’s Happening Here?

Here’s what’s happening when your browser hits your application:

1. Your browser makes a network connection to your own computer, which is called `localhost` and is a standard way of saying “whatever my own computer is called on the network.” It also uses port 8080.
2. Once it connects, the browser makes an HTTP request to the `bin/app.rb` application and asks for the `/` URL, which is commonly the first URL on any website.

3. Inside `bin/app.rb` is a list of URLs and the classes that they match. The only one we have is the `'/'`, `'index'` mapping. This means that whenever someone goes to `/` with a browser, Sinatra will find the class `index` and load it to handle the request.
4. Now that Sinatra has found `class index`, it calls the `index.GET` method on an instance of that class to actually handle the request. This function runs and simply returns a string for what Sinatra should send to the browser.
5. Finally, Sinatra has handled the request and sends this response to the browser, which is what you are seeing.

Make sure you really understand this. Draw a diagram of how this information flows from your browser, to Sinatra, then to `index.GET`, and back to your browser.

Stopping and Reloading Sinatra

When you make a change to your Sinatra application, you need to stop it and run it again. Simply hit `CTRL-c` and it will stop. Then run the command `ruby bin/app.rb` again to start it. You can also use a tool called `rerun` to automate this process. You install `rerun` using `sudo gem install rerun` and then use it like this:

```
$ rerun 'ruby bin/app.rb'
```

Now make a small change to the `bin/app.rb` file and it should stop and then rerun your Sinatra application. It can take a few seconds for `rerun` to detect the change, but it should work.

Fixing Errors

Go to `http://localhost:8080/hello/` with your browser and watch as Sinatra gives you a standard error page. You're receiving this error because you have not created the `views/index.erb` template yet which line 14 needs to work correctly. We'll fix this in the next section, but for now read through this and familiarize yourself with what an error looks like.

1. Look at the top right of the page for the main error, "Errno::ENOENT at `/hello/`". It is followed by a better description, the file involved, and the line that may contain the error.
2. Look at the BACKTRACE section. It shows you a trace of each line involved in the error that should look familiar to you from other errors you've seen.

3. Look at the GET and POST section. These will say there is no data, but in later exercises you'll be passing data that might show up here as variables from forms.
4. Look at the COOKIES section. This is data your browser stores for each request, but we won't be using it in this book just yet.

Create Basic Templates

You can break your Sinatra application, but did you notice that "Hello World" isn't a very good HTML page? This is a web application, and as such it needs a proper HTML response. We'll create one in the second handler for `/hello/` that appears in lines 12–15. A "handler" is simply a chunk of code that runs when your browser goes to a new place in the application. I'll explain a lot more in the next two exercises, but for now put this in the file `views/index.erb`:

```
index.erb
```

```
1  <html>
2    <head>
3      <title>Gothons Of Planet Percal #25</title>
4    </head>
5  <body>
6
7  <p>
8    I just wanted to say <em style="color: green; font-size: 2em;"><%= greeting %></em>.
9  </p>
10 </body>
11 </html>
```

If you know what HTML is, then this should look fairly familiar. If not, research HTML and try writing a few web pages by hand so you know how it works. This HTML file is a *template*, which means that Sinatra will fill in "holes" in the text depending on variables you pass in to the template. For example, every place you see `$greeting` will be a variable you'll pass to the template that alters its contents.

Once you have that in place, visit `http://localhost:8080/hello/?greeting=Hi` in your browser. You should see a different message in green. You should also be able to do a View Source operation on the page in your browser to see that it is valid HTML. You can also try `http://localhost:8080/hello/` alone and see the default message.

This may have flown by you very fast, so let me explain how a template works:

1. The handler get `'/hello/'` do specifies what happens when your browser goes to `/hello/`.
2. In the handler we are getting the `?greeting=Hi` part by using the `params` hashmap. Sinatra builds `params` from the request, and we want to make the `greeting` from it. Notice the `|| "Hi There"` at the end? This is a way of saying, "Either use what's in `params` or use 'Hi There'."

3. Next the `/hello/` handler uses the `erb` function (remember Ruby lets you not use parentheses on function calls) to "render" the `:index` view. The `:locals = {'greeting' => greeting}` part says, "Also give this greet view the local variables `greeting` with this setting."
4. When I say `:greet` view, I mean the file `views/index.erb` you created earlier. Sinatra knows this too and loads it for you, passing in the `greeting` variable as you requested.
5. The `views/index.erb` file is called a template, and it has holes in it that are dynamically created. The part that reads `<%= greeting %>` does that, so Sinatra runs this code, takes the resulting HTML, and sends it to your browser.

To get deeper into this process, change the `greeting` variable and the HTML to see what effect it has. We will explore how this works in the last two exercises, but play with it some more now and see what you can do.

Study Drills

1. Read the documentation at <http://www.sinatrarb.com/documentation.html>.
2. Experiment with everything you can find there, including the example code.
3. Read about HTML5 and CSS. It's a large topic but just try to get familiar with it.
4. Put some content in `static/howdy.html` and go to <http://localhost:8080/howdy.html>. If it doesn't work, make sure your `app.rb` file has the correct `:public_folder` setting.

Common Student Questions

I can't seem to connect to `http://localhost:8080/`.

Try going to <http://127.0.0.1:8080/> instead.

I can't use port 8080 on my computer.

You probably have an anti-virus program installed that is using that port. Try a different port by changing the set `:port, 8080` line to a different number above 1024.

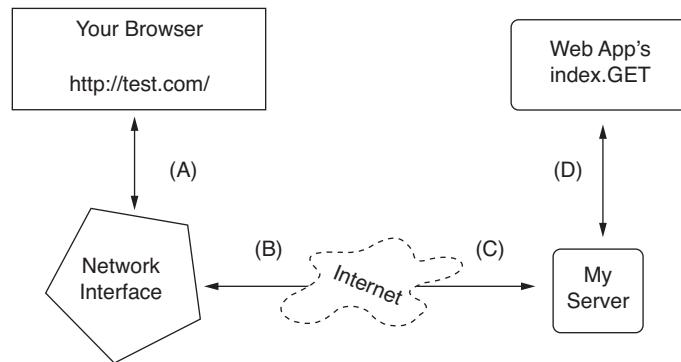
Getting Input from a Browser

While it's exciting to see the browser display "Hello World," it's even more exciting to let the user submit text to your application from a form. In this exercise we'll improve our starter web application by using forms and storing information about users into their "sessions."

How the Web Works

Time for some boring stuff. You need to understand a bit more about how the web works before you can make a form. This description isn't complete, but it's accurate and will help you figure out what might be going wrong with your application. Also, creating forms will be easier if you know what they do.

I'll start with a simple diagram that shows you the different parts of a web request and how the information flows:



I've labeled the lines with letters so I can walk you through a regular request process:

1. You type the url `http://test.com/` into your browser and it sends the request on line (A) to your computer's network interface.
2. Your request goes out over the Internet on line (B) and then to the remote computer on line (C) where my server accepts the request.
3. Once my computer accepts it, my web application gets it on line (D), and my Ruby code runs the `index.GET` handler.
4. The response comes out of my Ruby server when I return it, and it goes back to your browser over line (D).

5. The server running this site takes the response off line (D) and sends it back over the Internet on line (C).
6. The response from the server then comes off the Internet on line (B), and your computer's network interface hands it to your browser on line (A).
7. Finally, your browser displays the response.

In this description there are a few terms you should know so that you have a common vocabulary to work with when talking about your web application:

Browser The software that you're probably using every day. Most people don't know what a browser really does. They just call browsers "the Internet." Its job is to take addresses (like `http://test.com/`) you type into the URL bar, then use that information to make requests to the server at that address.

Address This is normally a URL (Uniform Resource Locator) like `http://test.com//` and indicates where a browser should go. The first part, `http`, indicates the protocol you want to use—in this case, "Hyper-Text Transport Protocol." You can also try `ftp://ibiblio.org/` to see how "File Transport Protocol" works. The `http://test.com/` part is the "hostname," or a human readable address you can remember and which maps to a number called an IP address, similar to a telephone number for a computer on the Internet. Finally, URLs can have a trailing path like the `/book/` part of `http://test.com//book/`, which indicates a file or some resource on the server to retrieve with a request. There are many other parts, but those are the main ones.

Connection Once a browser knows which protocol you want to use (`http`), which server you want to talk to (`http://test.com/`), and which resource on that server to get, it must make a connection. The browser simply asks your operating system (OS) to open a "port" to the computer, usually port 80. When it works, the OS hands back to your program something that works like a file, but is actually sending and receiving bytes over the network wires between your computer and the other computer at `http://test.com/`. This is the same thing that happens with `http://localhost:8080/`, but in this case you're telling the browser to connect to your own computer (`localhost`) and use port 8080 rather than the default of 80. You could also specify `http://test.com:80/` and get the same result, except you're explicitly saying to use port 80 instead of letting it be that by default.

Request Your browser is connected using the address you gave. Now it needs to ask for the resource it wants (or you want) on the remote server. If you gave `/book/` at the end of the URL, then you want the file (resource) at `/book/`, and most servers will use the real file `/book/index.html` but pretend it doesn't exist. What the browser does to get this resource is send a *request* to the server. I won't get into exactly how it does this, but just understand that it has to send something to query the server for the request. The

interesting thing is that these “resources” don’t have to be files. For instance, when the browser in your application asks for something, the server returns something your Ruby code generated.

Server The server is the computer at the end of a browser’s connection that knows how to answer your browser’s requests for files/resources. Most web servers just send files, and that accounts for the majority of traffic. But you’re actually building a server in Ruby that knows how to take requests for resources, and then return strings that you craft using Ruby. When you do this crafting, *you* are pretending to be a file to the browser, but really it’s just code. As you can see from Exercise 50, it doesn’t take much code to create a response.

Response This is the HTML (CSS, JavaScript, or images) your server wants to send back to the browser as the answer to the browser’s request. In the case of files, it just reads them off the disk and sends them to the browser, but it wraps the contents of the disk in a special “header” so the browser knows what it’s getting. In the case of your application, you’re still sending the same thing, including the header, but you generate that data on the fly with your Ruby code.

That is the fastest crash course in how a web browser accesses information on servers on the Internet. It should suffice for you to understand this exercise, but if not, read about it as much as you can until you get it. A really good way to do that is to take the diagram and break different parts of the web application you created in Exercise 50. If you can break your web application in predictable ways using the diagram, you’ll start to understand how it works.

How Forms Work

The easiest way to see how forms work is to create one. Change your `bin/app.rb` code to be this now:

app.rb

```
1 require 'sinatra'
2
3 set :port, 8080
4 set :static, true
5 set :public_folder, "static"
6 set :views, "views"
7
8 get '/' do
9   return 'Hello world'
10 end
11
12 get '/hello/' do
13   erb :hello_form
14 end
```

```
15 post '/hello/' do
16   greeting = params[:greeting] || "Hi There"
17   name = params[:name] || "Nobody"
18
19   erb :index, :locals => {'greeting' => greeting, 'name' => name}
20
21 end
```

The changes I've made are:

1. Make the get '/hello/' handler simply return a :hello_form page. We'll make that next.
2. Create a different handler that is post '/hello/' which looks similar to the previous version of this code—except we start with post to indicate we'll be receiving a form, and then we also accept a name parameter. We'll update the index.erb to match this next.

Once you have that, create this views/index.erb file for the post '/hello/' handler to use:
index.erb

```
1 <html>
2   <head>
3     <title>Gothons Of Planet Percal #25</title>
4   </head>
5   <body>
6
7   <p>
8     I just wanted to say <em style="color: green; font-size: 2em;"><%= greeting %>,
9     <%= name %></em>.
10   </p>
11
12   </body>
13 </html>
```

This script simply adds the name variable to the end of our greeting. Make sure you see how this name variable is now in the post '/hello/' handler and is passed to the erb :index call there.

The secret to making this work is the :hello_form (aka, views/hello_form.erb), which we'll cover next.

Creating HTML Forms

The last piece of this form puzzle is to make the form in views/hello_form.erb:

hello_form.erb

```
1 <html>
2   <head>
3     <title>Sample Web Form</title>
4   </head>
```

```
5  <body>
6
7  <h1>Fill Out This Form</h1>
8
9  <form action="/hello/" method="POST">
10   A Greeting: <input type="text" name="greeting">
11   <br/>
12   Your Name: <input type="text" name="name">
13   <br/>
14   <input type="submit">
15 </form>
16
17 </body>
18 </html>
```

A form then consists of the following:

1. The `<form>` tag starts it off and says where to deliver this form. In this case it's to `action="/hello/"` which is our post `'/hello/'` handler, and `method="POST"` which tells the browser to use this mechanism.
2. Text like you might put in another HTML tag, but also...
3. `<input>` tags give the type of input fields we want, and the parameters to use. In this case we have two: one with `name="greeting"` for our `params[:greeting]` parameters, and `name="name"` for our `params[:name]` parameter.
4. These parameters are then mapped in our post `'/hello/'` code to create the `greeting` and `name` variables, which are passed as `:locals` to the `erb :index` call.
5. Finally, the file `views/index.erb` gets these variables and it prints them.

Creating a Layout Template

As programmers, we have to find common patterns and try to automate them away. One common pattern is the HTML found at the beginning and the end of each of our `.erb` files. You shouldn't have to type that HTML every single time you want to create a new view, and you should be able to change that content in one place to change all the pages. The solution is called a "layout template," which we'll create in `views/layout.erb`:

layout.erb

```
1  <html>
2  <head>
3    <title>Gothons From Planet Percal #25</title>
4  </head>
5  <body>
6
```

```
7   <%= yield %>
8
9   </body>
10  </html>
```

This simply takes the common HTML at the top and bottom of every template and puts it into one file. The code `<%= yield %>` is a Ruby thing that says to stop there, run the other view, and then come back. By putting this into `views/layout.erb` we're telling Sinatra to "wrap" all of our templates with this HTML.

We can now write abbreviated HTML for the `views/index.erb` file:

`index_laid_out.erb`

```
1  <p>
2    I just wanted to say <em style="color: green; font-size: 2em;"><%= greeting %>,
3      <%= name %></em>.
4  </p>
```

We can also do the same for the `views/html_form.erb` file:

`hello_form_laid_out.erb`

```
1  <h1>Fill Out This Form</h1>
2
3  <form action="/hello/" method="POST">
4    A Greeting: <input type="text" name="greeting">
5    <br/>
6    Your Name: <input type="text" name="name">
7    <br/>
8    <input type="submit">
9  </form>
```

Change your files to match these by deleting the same content and then reloading them. Use your browser's View Source feature to see that, even though you do not have the contents of `views/layout.erb` in your new views, it still shows up in the resulting pages.

Writing Automated Tests for Forms

You can also automate the testing of your web application using `Rack::Test`. First install the gem for it:

```
$ sudo gem install rack-test
```

Then write a test file in `tests/test_gothonweb.rb`:

`test_gothonweb.rb`

```
1  require './bin/app.rb'
2  require 'test/unit'
```

```

3   require 'rack/test'
4
5   class MyAppTest < Test::Unit::TestCase
6     include Rack::Test::Methods
7
8     def app
9       Sinatra::Application
10    end
11
12    def test_my_default
13      get '/'
14      assert_equal 'Hello world', last_response.body
15    end
16
17    def test_hello_form
18      get '/hello/'
19      assert last_response.ok?
20      assert last_response.body.include?('A Greeting')
21    end
22
23    def test_hello_form_post
24      post '/hello/', params={:name => 'Frank', :greeting => "Hi"}
25      assert last_response.ok?
26      assert last_response.body.include?('I just wanted to say')
27    end
28  end

```

This file is simply pretending to be a web browser, and it looks similar to how the Sinatra handlers are, but written as if you were telling a browser to visit your webapplication with code. To run this test do what you normally do:

```

$ rake test
/usr/local/bin/ruby -I"lib:tests" -I"/usr/local/lib/ruby/2.1.0" "/usr/local/
  lib/ruby/2.1.0/rake/rake_test_loader.rb" "tests/test_gothonweb.rb"
Run options:

# Running tests:

Finished tests in 0.028189s, 106.4245 tests/s, 177.3742 assertions/s.
3 tests, 5 assertions, 0 failures, 0 errors, 0 skips

ruby -v: ruby 2.1.2p95 (2014-05-08 revision 45877) [x86_64-darwin11.0]

```

Study Drills

1. Read even more about HTML, and give the simple form a better layout. It helps to draw what you want to do on paper and *then* implement it with HTML.

2. This one is hard, but try to figure out how you'd do a file upload form so that you can upload an image and save it to disk.
3. This is even more mind-numbing, but go find the HTTP RFC (the document that describes how HTTP works) and read as much of it as you can. It is really boring but the information comes in handy once in a while.
4. This will also be really difficult, but see if you can find someone to help you set up a web server like Apache, Nginx, or httpd. Try to serve a couple of your .html and .css files with it just to see if you can. Don't worry if you can't. Web servers kind of suck.
5. Take a break after this exercise, and just try making as many different web applications as you can. You should *definitely* read about sessions in Sinatra so you can understand how to keep state for a user.

Common Student Question

Why don't my tests run?

Remember, do *not* change to the tests directory. If you do `cd tests` to run your tests, you are doing it wrong.

The Start of Your Web Game

We're coming to the end of the book, and in this exercise I'm going to really challenge you. When you're done, you'll be a reasonably competent Ruby beginner. You'll still need to go through a few more books and write a couple more projects, but you'll have the skills to complete them. The only thing in your way will be time, motivation, and resources.

In this exercise, we won't make a complete game, but instead we'll make an "engine" that can run the game from Exercise 47 in the browser. This will involve refactoring Exercise 43, mixing in the structure from Exercise 47, adding automated tests, and finally creating a web engine that can run the games.

This exercise will be *huge*, and I predict you could spend anywhere from a week to months on it before moving on. It's best to attack it in little chunks and do a bit a night, taking your time to make everything work before moving on.

Refactoring the Exercise 43 Game

You've been altering the gothonweb project for two exercises, and you'll do it one more time in this exercise. The skill you're learning is called "refactoring," or as I like to call it, "fixing stuff." Refactoring is a term programmers use to describe the process of taking old code, and changing it to have new features or just to clean it up. You've been doing this without even knowing it, as it's second nature to building software.

What you'll do in this part is take the ideas from Exercise 47 of a testable "map" of rooms, along with the game from Exercise 43, and combine them together to create a new game structure. It will have the same content, just "refactored" to have a better structure.

The first step is to grab the code from `ex47/lib/ex47/game.rb` and copy it to `lib/gothonweb/map.rb`. Next, copy the `ex47/tests/ex47_tests.rb` file to `tests/map_tests.rb` and run `rake test` again to make sure it keeps working. You'll need to change the `require` command in `test_map.rb` to match the new `gothonweb/map.rb` location.

WARNING! I won't show you the output of a test run. Just assume that you should be doing it and it'll look like mine unless you have an error.

Once you have the code from Exercise 47 copied over, it's time to refactor it to include the Exercise 43 map. I'm going to start by laying down the basic structure, and then you'll have an assignment to make the `map.rb` file and the `map_tests.rb` file complete.

First you'll want to put the original Room class into a module so put this right at the top of map.rb:

map.rb

```
1 module Map
```

Next you need to write up the rooms. Here are the complete room descriptions, but you can also put in dummy text for them and then fill in the full text later when the game is working. These go after the Room class you already have from Exercise 47:

map.rb

```
1 CENTRAL_CORRIDOR = Room.new("Central Corridor",
2 """
3 The Gothons of Planet Percal #25 have invaded your ship and destroyed
4 your entire crew. You are the last surviving member and your last
5 mission is to get the neutron destruct bomb from the Weapons Armory,
6 put it in the bridge, and blow the ship up after getting into an
7 escape pod.
8
9 You're running down the central corridor to the Weapons Armory when
10 a Gothon jumps out, red scaly skin, dark grimy teeth, and evil clown costume
11 flowing around his hate filled body. He's blocking the door to the
12 Armory and about to pull a weapon to blast you.
13 """
14
15
16 LASER_WEAPON_ARMORY = Room.new("Laser Weapon Armory",
17 """
18 Lucky for you they made you learn Gothon insults in the academy.
19 You tell the one Gothon joke you know:
20 Lbhe zbgure vf fb sng, jura fur fvgf nebhaq gur ubhfr, fur fvgf nebhaq gur ubhfr.
21 The Gothon stops, tries not to laugh, then busts out laughing and can't move.
22 While he's laughing you run up and shoot him square in the head
23 putting him down, then jump through the Weapon Armory door.
24
25 You do a dive roll into the Weapon Armory, crouch and scan the room
26 for more Gothons that might be hiding. It's dead quiet, too quiet.
27 You stand up and run to the far side of the room and find the
28 neutron bomb in its container. There's a keypad lock on the box
29 and you need the code to get the bomb out. If you get the code
30 wrong 10 times then the lock closes forever and you can't
31 get the bomb. The code is 3 digits.
32 """
33
34
35 THE_BRIDGE = Room.new("The Bridge",
36 """
37 The container clicks open and the seal breaks, letting gas out.
38 You grab the neutron bomb and run as fast as you can to the
39 bridge where you must place it in the right spot.
40
41 You burst onto the Bridge with the netron destruct bomb
42 under your arm and surprise 5 Gothons who are trying to
43 take control of the ship. Each of them has an even uglier
44 clown costume than the last. They haven't pulled their
45 weapons out yet, as they see the active bomb under your
```

```

46    arm and don't want to set it off.
47    """")
48
49
50    ESCAPE_POD = Room.new("Escape Pod",
51    """
52    You point your blaster at the bomb under your arm
53    and the Gothons put their hands up and start to sweat.
54    You inch backward to the door, open it, and then carefully
55    place the bomb on the floor, pointing your blaster at it.
56    You then jump back through the door, punch the close button
57    and blast the lock so the Gothons can't get out.
58    Now that the bomb is placed you run to the escape pod to
59    get off this tin can.
60
61    You rush through the ship desperately trying to make it to
62    the escape pod before the whole ship explodes. It seems like
63    hardly any Gothons are on the ship, so your run is clear of
64    interference. You get to the chamber with the escape pods, and
65    now need to pick one to take. Some of them could be damaged
66    but you don't have time to look. There's 5 pods, which one
67    do you take?
68    """")
69
70
71    THE_END_WINNER = Room.new("The End",
72    """
73    You jump into pod 2 and hit the eject button.
74    The pod easily slides out into space heading to
75    the planet below. As it flies to the planet, you look
76    back and see your ship implode then explode like a
77    bright star, taking out the Gothon ship at the same
78    time. You won!
79    """")
80
81
82    THE_END_LOSER = Room.new("The End",
83    """
84    You jump into a random pod and hit the eject button.
85    The pod escapes out into the void of space, then
86    implodes as the hull ruptures, crushing your body
87    into jam jelly.
88    """")
89

```

After this you need to connect these rooms to create the map by using `Room.add_paths`:

`map.rb`

```

1  ESCAPE_POD.add_paths({
2      '2' => THE_END_WINNER,
3      '*'  => THE_END_LOSER
4  })
5
6  GENERIC_DEATH = Room.new("death", "You died.")

```

```

7   THE_BRIDGE.add_paths({
8     'throw the bomb' => GENERIC_DEATH,
9     'slowly place the bomb' => ESCAPE_POD
10    })
11  }
12
13 LASER_WEAPON_ARMORY.add_paths({
14   '0132' => THE_BRIDGE,
15   '*' => GENERIC_DEATH
16  })
17
18 CENTRAL_CORRIDOR.add_paths({
19   'shoot!' => GENERIC_DEATH,
20   'dodge!' => GENERIC_DEATH,
21   'tell a joke' => LASER_WEAPON_ARMORY
22  })
23
24 START = CENTRAL_CORRIDOR

```

Lastly, we'll need a way to find rooms by their names, load them, and save them to a "session." I'll explain what a session is in the next section, but enter this code for the last part of `map.rb`:

`map.rb`

```

1      # A whitelist of allowed room names. We use this so that
2      # bad people on the internet can't access random variables
3      # with names. You can use Test::constants and Kernel.const_get
4      # too.
5      ROOM_NAMES = {
6        'CENTRAL_CORRIDOR' => CENTRAL_CORRIDOR,
7        'LASER_WEAPON_ARMORY' => LASER_WEAPON_ARMORY,
8        'THE_BRIDGE' => THE_BRIDGE,
9        'ESCAPE_POD' => ESCAPE_POD,
10       'THE_END_WINNER' => THE_END_WINNER,
11       'THE_END_LOSER' => THE_END_LOSER,
12       'START' => START,
13     }
14
15   def Map::load_room(session)
16     # Given a session this will return the right room or nil
17     return ROOM_NAMES[session[:room]]
18   end
19
20   def Map::save_room(session, room)
21     # Store the room in the session for later, using its name
22     session[:room] = ROOM_NAMES.key(room)
23   end
24
25 end

```

You'll notice that there are a couple of problems with our Room class and this map:

1. There are parts in the original game where we ran code that determined things like the bomb's keypad code, or the right pod. In this game we just pick some defaults and go with them, but later you'll be given Study Drills to make this feature work again.
2. I've made a generic_death ending to handle all of the bad decisions, which you'll have to finish for me. You'll need to go back through and add in all the original endings and make sure they work.
3. I've got a new kind of transition labeled "*" that will be used for a "catch-all" action in the engine.

Once you have that written, here's the new automated test `tests/test_map.rb` that you should run to get yourself started:

`test_map.rb`

```

1  require "gothonweb/map.rb"
2  require "test/unit"
3
4  class TestMap < Test::Unit::TestCase
5
6      def test_room()
7          gold = Map::Room.new("GoldMap::Room",
8              """This room has gold in it you can grab. There's a
9              door to the north."""")
10         assert_equal(gold.name, "GoldMap::Room")
11         assert_equal(gold.paths, {})
12     end
13
14     def test_room_paths()
15         center = Map::Room.new("Center", "Test room in the center.")
16         north = Map::Room.new("North", "Test room in the north.")
17         south = Map::Room.new("South", "Test room in the south.")
18
19         center.add_paths({ 'north' => north, 'south' => south })
20         assert_equal(center.go('north'), north)
21         assert_equal(center.go('south'), south)
22     end
23
24     def test_map()
25         start = Map::Room.new("Start", "You can go west and down a hole.")
26         west = Map::Room.new("Trees", "There are trees here, you can go east.")
27         down = Map::Room.new("Dungeon", "It's dark down here, you can go up.")
28
29         start.add_paths({ 'west' => west, 'down' => down })
30         west.add_paths({ 'east' => start })
31         down.add_paths({ 'up' => start })
32
33         assert_equal(start.go('west'), west)
34         assert_equal(start.go('west').go('east'), start)
35     end

```

```
36         assert_equal(start.go('down').go('up'), start)
37     end
38
39     def test_gothon_game_map()
40         assert_equal(Map::START.go('shoot!'), Map::GENERIC_DEATH)
41         assert_equal(Map::START.go('dodge!'), Map::GENERIC_DEATH)
42
43         room = Map::START.go('tell a joke')
44         assert_equal(room, Map::LASER_WEAPON_ARMORY)
45
46         # complete this test by making it play the game
47     end
48
49     def test_session_loading()
50         session = {}
51
52         room = Map::load_room(session)
53         assert_equal(room, nil)
54
55         Map::save_room(session, Map::START)
56         room = Map::load_room(session)
57         assert_equal(room, Map::START)
58
59         room = room.go('tell a joke')
60         assert_equal(room, Map::LASER_WEAPON_ARMORY)
61
62         Map::save_room(session, room)
63         assert_equal(room, Map::LASER_WEAPON_ARMORY)
64     end
65 end
```

Your task in this part of the exercise is to complete the map and make the automated test completely validate the whole map. This includes fixing all the generic_death objects to be real endings. Make sure this works really well and that your test is as complete as possible, because we'll be changing this map later and you'll use the tests to make sure it keeps working.

Sessions and Tracking Users

At a certain point in your web application, you'll need to keep track of some information and associate it with the user's browser. The web (because of HTTP) is what we like to call "stateless," which means each request you make is independent of any other requests being made. If you request page A, put in some data, and click a link to page B, all the data you sent to page A just disappears.

The solution to this dilemma is to create a little data store (usually in a database or on the disk) that uses a number unique to each browser to keep track of what that browser was doing. This approach, called "session tracking," uses cookies in the browser to maintain the state of the user through the application. In the little Sinatra framework, it's fairly easy to implement by adding this line at the top:

```
enable :sessions
```

Put that line where you put your set calls, and then you can use the session like this:

```
# set a value in the session for later
session[:room] = "central_corridor"

# get the room in another handler with
current_room = session[:room]
```

This is what we were doing in `map.rb` with the `Map::load_room` and `Map::store_room`, which we used to keep track of which room the player was currently in. You could also use this strategy to keep track of choices they've made, if monsters have died, or how much damage the player has sustained.

You can read more about Sinatra's sessions at the project's README at <http://www.sinatrarb.com/intro.html#Using%20Sessions>.

Creating an Engine

You should have your game map working and a good unit test for it. I now want you to make a simple little game engine that will run the rooms, collect input from the player, and keep track of where a player is in the game. We'll be using the sessions you just learned to make a simple game engine that will:

1. Start a new game for a new user.
 2. Present the room to the user.
 3. Take input from the user.
 4. Run user input through the game.
 5. Display the results and keep going until the user dies.
-

WARNING! If you did not install `rerun` in Exercise 50, then you'll want to install it now for doing the rest of this exercise.

To do this, you're going to take the trusty `bin/app.rb` you've been hacking on and create a fully working, session-based game engine. The catch is: I'm going to make a very simple one with *basic HTML files*, and it'll be up to you to complete it. Here's the base engine:

`app.rb`

```
1  require 'sinatra'
2  require './lib/gothonweb/map.rb'
3
4  set :port, 8080
5  set :static, true
```

```
6   set :public_folder, "static"
7   set :views, "views"
8   enable :sessions
9   set :session_secret, 'BADSECRET'
10
11  get '/' do
12    session[:room] = 'START'
13    redirect to('/game')
14  end
15
16  get '/game' do
17    room = Map::load_room(session)
18
19    if room
20      erb :show_room, :locals => { :room => room }
21    else
22      erb :you_died
23    end
24  end
25
26
27  post '/game' do
28    room = Map::load_room(session)
29    action = params[:action]
30
31    if room
32      next_room = room.go(action) || room.go("*")
33
34      if next_room
35        Map::save_room(session, next_room)
36      end
37
38      redirect to('/game')
39    else
40      erb :you_died
41    end
42  end
```

There are even more new things in this script, but amazingly it's an entire web-based game engine in a small file. The biggest "hack" in the script consists of the lines that bring the sessions back, which are needed so that debug mode reloading works. Otherwise, each time you refresh the page the sessions will disappear and the game won't work.

You should next delete `views/hello_form.erb` and `views/index.erb` and create the two views mentioned in the preceding code. Here's a very simple `views/show_room.erb`:

show_room.erb

```
1  <h1> <%= room.name %> </h1>
2
```

```
3  <pre>
4    <%= room.description %>
5  </pre>
6
7  <% if room.name == "death" || room.name == "The End" %>
8    <p><a href="/">Play Again?</a></p>
9  <% else %>
10   <p>
11     <form action="/game" method="POST">
12       - <input type="text" name="action"> <input type="SUBMIT">
13     </form>
14   </p>
15 <% end %>
```

This template shows a room as you travel through the game. Next you need one to tell someone they died in the case they got to the end of the map by accident, which is `views/you_died.html`:

`you_died.erb`

```
1  <h1>You Died!</h1>
2
3  <p>Looks like you bit the dust.</p>
4  <p><a href="/">Play Again</a></p>
```

With those elements in place, you should now be able to do the following:

1. Get the test `tests/test_app.rb` working again so that you are testing the game. You won't be able to do much more than make a few clicks in the game because of sessions, but you should be able to do some basics.
2. Run the rerun '`ruby bin/app.rb`' script and test the game.

You should be able to refresh and fix the game like normal, and work with the game HTML and engine until it does all the things you want it to do.

Your Final Exam

Do you feel like this was a huge amount of information thrown at you all at once? Good—I want you to have something to tinker with while you build your skills. To complete this exercise, I'm going to give you a final set of exercises for you to complete on your own. You'll notice that what you've written so far isn't very well built; it is just a first version of the code. Your task now is to make the game more complete by doing these things:

1. Fix all the bugs I mention in the code, and any that I didn't mention. If you find new bugs, let me know.

2. Improve all of the automated tests so that you test more of the application and get to a point where you use a test rather than your browser to check the application while you work.
3. Make the HTML look better.
4. Research logins and create a signup system for the application, so people can have logins and high scores.
5. Complete the game map, making it as large and feature-complete as possible.
6. Give people a “help” system that lets them ask what they can do at each room in the game.
7. Add any other features you can think of to the game.
8. Create several “maps” and let people choose a game they want to run. Your `bin/app.rb` engine should be able to run any map of rooms you give it, so you can support multiple games.
9. Finally, use what you learned in Exercises 48 and 49 to create a better input processor. You have most of the code necessary; you just need to improve the grammar and hook it up to your input form and the GameEngine.

Good luck!

Next Steps

You're not a programmer quite yet. I like to think of this book as giving you your "programming black belt." You know enough to start another book on programming and handle it just fine. This book should have given you the mental tools and attitude you need to go through most Ruby books and actually learn something. It might even make it easy.

I recommend you check out some of these projects and try to build something with them:

- *The Well-Grounded Rubyist* to continue your Ruby study and become better at it.
- *Eloquent Ruby* for even more Ruby skills.
- *Learn Python the Hard Way*, you will learn even more about programming as you learn more programming languages, so try learning Python, too.
- *Rails Tutorial* to learn Ruby on Rails.
- *RubyMotion* for building native iOS applications.
- *Thor* to build command line tools.
- *Mechanize* for automatic web crawling and scraping.
- *Chef Solo* for automating system administration.
- *Ruby-Processing* for computer art.
- *Padrino* is like Sinatra but with a lot more features; it works more like Ruby on Rails.
- *Learn C the Hard Way* after you're familiar with Ruby, try learning C and algorithms with my other book. Take it slow; C is different but a very good thing to learn.

Pick one of the these resources, and go through any tutorials and documentation they have. As you go through documentation with code in it, *type in all of the code* and make it work. That's how I do it. That's how every programmer does it. Reading programming documentation is not enough to learn it; you have to do it. After you get through the tutorial and any other documentation, make something. Anything will do—even something someone else has already written. Just make something.

Just understand anything you write will probably suck. That's alright though, I suck at every programming language when I first start using it. Nobody writes pure perfect gold when they're a beginner, and anyone who tells you they did is a huge liar.

How to Learn Any Programming Language

I'm going to teach you how to learn most of the programming languages you may want to learn in the future. The organization of this book is based on how I and many other programmers learn new languages. The process that I usually follow is:

1. Get a book or some introductory text about the language.
2. Go through the book and type in all of the code, making all of it run.
3. Read the book as you work on the code, taking notes.
4. Use the language to implement a small set of programs you are familiar with in another language.
5. Read other people's code in the language, and try to copy their patterns.

In this book, I forced you to go through this process very slowly and in small chunks. Other books aren't organized in the same way, which means you have to extrapolate how I've made you do this to how their content is organized. The best way to do this is to skim through the book and make a list of all the major code sections. Turn this list into a set of exercises based on the chapters and then simply do them in order, one at a time.

This process also works for new technologies, assuming they have books you can read. For anything without books, you follow the same process but use online documentation or source code as your initial introduction.

Each new language you learn makes you a better programmer, and as you learn more, they become easier to learn. By your third or fourth language, you should be able to pick up similar languages in a week, with stranger languages taking longer. Now that you know Ruby, you could potentially learn Python and JavaScript fairly quickly by comparison. This is simply because many languages share similar concepts; once you learn the concepts in one language they work in others.

The final thing to remember about learning a new language is: Don't be a stupid tourist. A stupid tourist is someone who goes to another country and then complains that the food isn't like the food at home. "Why can't I get a good burger in this stupid country!?" When you're learning a new language, assume that what it does isn't stupid, it's just different, and embrace it so you can learn it.

After you learn a language, though, don't be a slave to that language's way of doing things. Sometimes the people who use a language actually do some very idiotic things for no other reason than "that's how we've always done it." If you like your style better and you know how everyone else does it, then feel free to break their rules if it improves things.

I really enjoy learning new programming languages. I think of myself as a "programmer anthropologist" and think of these languages as little insights into the group of programmers who use them. I'm learning a language they all use to talk to each other through computers, and I find this fascinating. Then again, I'm kind of a weird guy—so just learn programming languages because you want to.

Enjoy! This is really fun stuff.

This page intentionally left blank

Advice from an Old Programmer

You've finished this book and have decided to continue with programming. Maybe it will be a career for you, or maybe it will be a hobby. You'll need some advice to make sure you continue on the right path and get the most enjoyment out of your newly chosen activity.

I've been programming for a very long time—so long that it's incredibly boring to me. At the time that I wrote this book, I knew about 20 programming languages and could learn new ones in about a day to a week, depending on how weird they were. Eventually, though, this just became boring and couldn't hold my interest anymore. This doesn't mean I think programming *is* boring, or that you will think it's boring, only that I find it uninteresting at this point in my journey.

What I discovered during this journey of learning is that it's not the languages that matter, but what you do with them. Actually, I always knew that, but I'd get distracted by the languages and forget it periodically. Now I never forget it, and neither should you.

Which programming language you learn and use doesn't matter. Do *not* get sucked into the religion surrounding programming languages, as that will only blind you to their true purpose of being your tool for doing interesting things.

Programming as an intellectual activity is the *only* art form that allows you to create interactive art. You can create projects that other people can play with, and you can talk to them indirectly. No other art form is quite this interactive. Movies flow to the audience in one direction. Paintings do not move. Code goes both ways.

Programming as a profession is only moderately interesting. It can be a good job, but you could make about the same money and be happier running a fast-food joint. You're much better off using code as your secret weapon in another profession.

People who can code in the world of technology companies are a dime a dozen and get no respect. People who can code in biology, medicine, government, sociology, physics, history, and mathematics are respected and can do amazing things to advance those disciplines.

Of course, all of this advice is pointless. If you liked learning to write software with this book, you should try to use it to improve your life any way you can. Go out and explore this weird, wonderful, new intellectual pursuit that barely anyone in the last 50 years has been able to explore. Might as well enjoy it while you can.

Finally, I'll say that learning to create software changes you and makes you different. Not better or worse, just different. You may find that people treat you harshly because you can create software, maybe using words like "nerd." Maybe you'll find that because you can dissect their logic that they hate arguing with you. You may even find that simply knowing how a computer works makes you annoying and weird to them.

To this, I have just one piece of advice: they can go to hell. The world needs more weird people who know how things work and who love to figure it all out. When they treat you like this, just remember that this is *your* journey, not theirs. Being different is not a crime, and people who tell you it is are just jealous that you've picked up a skill they never in their wildest dreams could acquire.

You can code. They cannot. That is pretty damn cool.

This page intentionally left blank

Command Line Crash Course

This appendix is a quick, super-fast course in using the command line. It is intended to be done rapidly in about a day or two, and is not meant to teach you advanced shell usage.

Introduction: Shut Up and Shell

This appendix is a crash course in using the command line to make your computer perform tasks. As a crash course, it's not as detailed or extensive as my other books. It is simply designed to get you barely capable enough to start using your computer like a real programmer does. When you're done with this appendix, you will be able to give most of the basic commands that every shell user touches every day. You'll understand the basics of directories and a few other concepts.

The only piece of advice I am going to give you is this:

Shut up and type all of this in.

Sorry to be mean, but that's what you have to do. If you have an irrational fear of the command line, the only way to conquer an irrational fear is to just shut up and fight through it.

You are not going to destroy your computer. You are not going to be thrown into some jail at the bottom of Microsoft's Redmond campus. Your friends won't laugh at you for being a nerd. Simply ignore any stupid weird reasons you have for fearing the command line.

Why? Because if you want to learn to code, then you must learn this. Programming languages are advanced ways to control your computer with language. The command line is the little baby brother of programming languages. Learning the command line teaches you to control the computer using language. Once you get past that, you can then move on to writing code and feeling like you actually own the hunk of metal you just bought.

How to Use This Appendix

The best way to use this appendix is to do the following:

- Get yourself a small paper notebook and a pen.
- Start at the beginning of the appendix and do each exercise exactly as you're told.
- When you read something that doesn't make sense or that you don't understand, *write it down in your notebook*. Leave a little space so you can write an answer.

- After you finish an exercise, go back through your notebook and review the questions you have. Try to answer them by searching online and asking friends who might know the answer. Email me at help@learncodethehardway.org and I'll help you, too.

Just keep going through this process of doing an exercise, writing down questions you have, then going back through and answering the questions you can. By the time you're done, you'll actually know a lot more than you think about using the command line.

You Will Be Memorizing Things

I'm warning you ahead of time that I'm going to make you memorize things right away. This is the quickest way to get you capable at something, but for some people memorization is painful. Just fight through it and do it anyway. Memorization is an important skill in learning things, so you should get over your fear of it.

Here's how you memorize things:

- Tell yourself you *will* do it. Don't try to find tricks or easy ways out of it, just sit down and do it.
- Write what you want to memorize on some index cards. Put one half of what you need to learn on one side, then the other half on the other side.
- Every day for about 15–30 minutes, drill yourself on the index cards, trying to recall each one. Put any cards you don't get right into a different pile, just drill those cards until you get bored, and then try the whole deck and see if you improve.
- Before you go to bed, drill just the cards you got wrong for about 5 minutes, then go to sleep.

There are other techniques, like you can write what you need to learn on a sheet of paper, laminate it, then stick it to the wall of your shower. While you're bathing, drill the knowledge without looking, and when you get stuck glance at it to refresh your memory.

If you do this every day, you should be able to memorize most of the things I tell you to memorize in about a week to a month. Once you do, nearly everything else becomes easier and intuitive, which is the purpose of memorization. It's not to teach you abstract concepts, but rather to ingrain the basics so that they are intuitive and you don't have to think about them. Once you've memorized these basics, they stop being speed bumps preventing you from learning more advanced abstract concepts.

The Setup

In this appendix you will be instructed to do three things:

- Do some things in your shell (command line, Terminal, PowerShell).
- Learn about what you just did.
- Do more on your own.

For this first exercise, you'll be expected to get your Terminal open and working so that you can do the rest of the appendix.

Do This

Get your Terminal, shell, or PowerShell working so you can access it quickly and know that it works.

Mac OS X

For Mac OS X you'll need to do this:

- Hold down the command key and hit the spacebar.
- In the top right corner, the blue "search bar" will pop up.
- Type: terminal
- Click on the Terminal application that looks kind of like a black box.
- This will open Terminal.
- You can now go to your dock and CTRL-click to pull up the menu, then select Options->Keep In dock.

Now you have your Terminal open and it's in your dock so you can get to it.

Linux

I'm assuming that if you have Linux, then you already know how to get at your Terminal. Look through the menu for your window manager for anything named "Shell" or "Terminal."

Windows

On Windows, we're going to use PowerShell. People used to work with a program called cmd.exe, but it's not nearly as usable as PowerShell. If you have Windows 7 or later, do this:

- Click Start.
- In "Search programs and files" type: powershell
- Hit Enter.

If you don't have Windows 7, you should *seriously* consider upgrading. If you still insist on not upgrading then you can try installing it from Microsoft's download center. Search online to find "powershell downloads" for your version of Windows. You are on your own, though, since I don't have Windows XP, but hopefully the PowerShell experience is the same.

You Learned This

You learned how to get your Terminal open, so you can do the rest of this appendix.

WARNING! If you have that really smart friend who already knows Linux, ignore him when he tells you to use something other than bash. I'm teaching you bash. That's it. He will claim that zsh will give you 30 more IQ points and win you millions in the stock market. Ignore him. Your goal is to get capable enough and at this level it doesn't matter which shell you use. The next warning is stay off IRC or other places where "hackers" hang out. They think it's funny to hand you commands that can destroy your computer. The command rm -rf / is a classic that you *must never type*. Just avoid them. If you need help, make sure you get it from someone you trust and not from random idiots on the Internet.

Do More

This exercise has a large "do more" part. The other exercises are not as involved as this one, but I'm having you prime your brain for the rest of the appendix by doing some memorization. Just trust me: this will make things silky smooth later on.

Linux/Mac OS X

Take this list of commands and create index cards with the names on the left on one side, and the definitions on the other side. Drill them every day while continuing with the lessons in this appendix.

pwd print working directory
hostname my computer's network name
mkdir make directory
cd change directory
ls list directory
rmdir remove directory
pushd push directory
popd pop directory
cp copy a file or directory
mv move a file or directory
less page through a file
cat print the whole file
xargs execute arguments
find find files
grep find things inside files
man read a manual page
apropos find which man page is appropriate
env look at your environment
echo print some arguments
export export/set a new environment variable
exit exit the shell
sudo DANGER! become super user root DANGER!

Windows

If you're using Windows, then here's your list of commands:

pwd print working directory
hostname my computer's network name
mkdir make directory

cd change directory
ls list directory
rmdir remove directory
pushd push directory
popd pop directory
cp copy a file or directory
robocopy robust copy
mv move a file or directory
more page through a file
type print the whole file
forfiles run a command on lots of files
dir -r find files
select-string find things inside files
help read a manual page
helpctr find which manual page is appropriate
echo print some arguments
set export/set a new environment variable
exit exit the shell
runas DANGER! become super user root DANGER!

Drill, drill, drill! Drill until you can say these phrases right away when you see that word. Then drill the inverse, so that you read the phrase and know which command will do that. You're building your vocabulary by doing this, but don't spend so much time you go nuts and get bored.

Paths, Folders, and Directories (**pwd**)

In this exercise you learn how to print your working directory with the **pwd** command.

Do This

I'm going to teach you how to read these "sessions" that I show you. You don't have to type everything I list here, just some of the parts:

- You do *not* type in the \$ (UNIX) or > (Windows). That's just me showing you my session so you can see what I got.
- You type in the stuff after \$ or >, then hit Enter. So if I have \$ pwd, you type just pwd and hit Enter.
- You can then see what I have for output followed by another \$ or > prompt. That content is the output and you should see the same output.

Let's do a simple first command so you can get the hang of this:

Linux/OS X

Exercise 2 Session

```
$ pwd  
/Users/zedshaw  
$
```

Windows

Exercise 2 Windows Session

```
PS C:\Users\zed> pwd  
  
Path  
----  
C:\Users\zed  
  
PS C:\Users\zed>
```

WARNING! In this appendix I need to save space so that you can focus on the important details of the commands. To do this, I'm going to strip out the first part of the prompt (the PS C:\Users\zed above) and leave just the little > part. This means your prompt won't look exactly the same, but don't worry about that.

Remember that from now on I'll include only the > to tell you that's the prompt. I'm doing the same thing for the UNIX prompts, but UNIX prompts are so varied that most people get used to \$ meaning "just the prompt."

You Learned This

Your prompt will look different from mine. You may have your user name before the \$ and the name of your computer. On Windows it will probably look different, too. The key is that you see this pattern:

- There's a prompt.
- You type a command there. In this case, it's pwd.
- It printed something.
- Repeat.

You just learned what pwd does, which means "print working directory." What's a directory? It's a folder. Folder and directory are the same thing, and they're used interchangeably. When you open your file browser on your computer to graphically find files, you are walking through folders. Those folders are the exact same things as these "directories" we're going to work with.

Do More

- Type pwd 20 times and each time say "print working directory."
- Write down the path that this command gives you. Find it with your graphical file browser of choice.
- No, seriously, type it 20 times and say it out loud. Sssh. Just do it.

If You Get Lost

As you go through these instructions, you may get lost. You may not know where you are or where a file is and have no idea how to continue. To solve this problem, I am going to teach you the commands to type to stop being lost.

Whenever you get lost, it is most likely because you were typing commands and have no idea where you've ended up. What you should do is type pwd to *print your current directory*. This tells you where you are.

The next thing you need is a way of getting back to where you are safe, your home. To do this, type cd ~ and you are back in your home.

This means if you get lost at any time, you should type:

```
pwd  
cd ~
```

The first command `pwd` tells you where you are. The second command `cd ~` takes you home so you can try again.

Do This

Right now figure out where you are, and then go home using `pwd` and `cd ~`. This will ensure that you are always in the right place.

You Learned This

How to get back to your home if you ever get lost.

Make a Directory (`mkdir`)

In this exercise you learn how to make a new directory (folder) using the `mkdir` command.

Do This

Remember! *You need to go home first!* Do your `pwd` and then `cd ~` before doing this exercise. Before you do *all* exercises in this appendix, always go home first!

Linux/OS X

Exercise 4 Session

```
$ pwd  
$ cd ~  
$ mkdir temp  
$ mkdir temp/stuff  
$ mkdir temp/stuff/things  
$ mkdir -p temp/stuff/things/frank/joe/alex/john  
$
```

Windows**Exercise 4 Windows Session**

```
> pwd  
> cd ~  
> mkdir temp
```

Directory: C:\Users\zed

Mode	LastWriteTime	Length	Name
----	-----	-----	---
d----	12/17/2011 9:02 AM		temp

```
> mkdir temp/stuff
```

Directory: C:\Users\zed\temp

Mode	LastWriteTime	Length	Name
----	-----	-----	---
d----	12/17/2011 9:02 AM		stuff

```
> mkdir temp/stuff/things
```

Directory: C:\Users\zed\temp\stuff

Mode	LastWriteTime	Length	Name
----	-----	-----	---
d----	12/17/2011 9:03 AM		things

```
> mkdir temp/stuff/things/frank/joe/alex/john
```

```
Directory: C:\Users\zed\temp\stuff\things\frank\joe\alex
```

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
d----	12/17/2011 9:03 AM		john

>

This is the only time I'll list the `pwd` and `cd ~` commands. They are expected in the exercises *every time*. Do them all the time.

You Learned This

Now we get into typing more than one command. These are all the different ways you can run `mkdir`. What does `mkdir` do? It make directories. Why are you asking that? You should be doing your index cards and getting your commands memorized. If you don't know that "`mkdir` makes directories," then keep working the index cards.

What does it mean to make a directory? You might call directories "folders." They're the same thing. All you did in this exercise is create directories inside directories inside of more directories. This is called a "path" and it's a way of saying "first temp, then stuff, then things, and that's where I want it." It's a set of directions to the computer of where you want to put something in the tree of folders (directories) that make up your computer's hard disk.

WARNING! In this appendix I'm using the / (slash) character for all paths since it works the same on all computers now. However, Windows users need to know that you can also use the \ (backslash) character and other Windows users will expect that at times.

Do More

- The concept of a "path" might confuse you at this point. Don't worry. We'll do a lot more with them and then you'll get it.
- Make 20 other directories inside the `temp` directory in various levels. Go look at them with a graphical file browser.
- Make a directory with a space in the name by putting quotes around it: `mkdir "I Have Fun"`
- If the `temp` directory already exists, then you'll get an error. Use `cd` to change to a work directory that you can control and try it there. On Windows, Desktop is a good place.

Change Directory (cd)

In this exercise you learn how to change from one directory to another using the cd command.

Do This

I'm going to give you the instructions for these sessions one more time:

- You do *not* type in the \$ (UNIX) or > (Windows).
- You type in the stuff after this, then hit Enter. If I have \$ cd temp you just type cd temp and hit Enter.
- The output comes after you hit Enter, followed by another \$ or > prompt.
- Always go home first! Do pwd and then cd ~ so you go back to your starting point.

Linux/OS X

Exercise 5 Session

```
$ cd temp
$ pwd
~/temp
$ cd stuff
$ pwd
~/temp/stuff
$ cd things
$ pwd
~/temp/stuff/things
$ cd frank/
$ pwd
~/temp/stuff/things/frank
$ cd joe/
$ pwd
~/temp/stuff/things/frank/joe
$ cd alex/
$ pwd
~/temp/stuff/things/frank/joe/alex
$ cd john/
$ pwd
~/temp/stuff/things/frank/joe/alex/john
```

```
$ cd ..  
$ cd ..  
$ pwd  
~/temp/stuff/things/frank/joe  
$ cd ..  
$ cd ..  
$ pwd  
~/temp/stuff/things  
$ cd ../../..  
$ pwd  
~/  
$ cd temp/stuff/things/frank/joe/alex/john  
$ pwd  
~/temp/stuff/things/frank/joe/alex/john  
$ cd ../../../../../../..  
$ pwd  
~/  
$
```

Windows

Exercise 5 Windows Session

```
> cd temp  
> pwd
```

Path

C:\Users\zed\temp

```
> cd stuff  
> pwd
```

Path

C:\Users\zed\temp\stuff


```
> cd things
```

```
> pwd
```

```
Path
```

```
----
```

```
C:\Users\zed\temp\stuff\things
```

```
> cd frank
```

```
> pwd
```

```
Path
```

```
----
```

```
C:\Users\zed\temp\stuff\things\frank
```

```
> cd joe
```

```
> pwd
```

```
Path
```

```
----
```

```
C:\Users\zed\temp\stuff\things\frank\joe
```

```
> cd alex
```

```
> pwd
```

```
Path
```

```
----
```

```
C:\Users\zed\temp\stuff\things\frank\joe\alex
```

```
> cd john
```

```
> pwd
```

```
Path
```

```
----
```

```
C:\Users\zed\temp\stuff\things\frank\joe\alex\john
```

```
> cd ..
```

```
> cd ..  
> cd ..  
> pwd  
  
Path  
----  
C:\Users\zed\temp\stuff\things\frank
```

```
> cd ../../..  
> pwd  
  
Path  
----  
C:\Users\zed\temp\stuff
```

```
> cd ..  
> cd ..  
> cd temp/stuff/things/frank/joe/alex/john  
> cd ../../../../../../  
> pwd  
  
Path  
----  
C:\Users\zed
```

>

You Learned This

You made all these directories in the last exercise, and now you're just moving around inside them with the `cd` command. In my session, I also use `pwd` to check where I am, so remember not to type the output that `pwd` prints. For example, on line 3 you see `~/temp` but that's the output of `pwd` from the prompt above it. *Do not type this in.*

You should also see how I use the `..` to move "up" in the tree and path.

Do More

A very important part of learning to use the command line interface (CLI) on a computer with a graphical user interface (GUI) is figuring out how they work together. When I started using computers, there was no "GUI" and you did everything with the DOS prompt (the CLI). Later, when computers became powerful enough that everyone could have graphics, it was simple for me to match CLI directories with GUI windows and folders.

Most people today, however, have no comprehension of the CLI, paths, and directories. In fact, it's very difficult to teach it to them and the only way to learn about the connection is for you to constantly work with the CLI until one day it clicks that things you do in the GUI will show up in the CLI.

The way you do this is by spending some time finding directories with your GUI file browser, then going to them with your CLI. This is what you'll do next.

- cd to the joe directory with one command.
- cd back to temp with one command, but not further above that.
- Find out how to cd to your "home directory" with one command.
- cd to your Documents directory, then find it with your GUI file browser (e.g., Finder, Windows Explorer, etc.).
- cd to your Downloads directory, then find it with your file browser.
- Find another directory with your file browser, then cd to it.
- Remember when you put quotes around a directory with spaces in it? You can do that with any command. For example, if you have a directory I Have Fun, then you can do: cd "I Have Fun"

List Directory (ls)

In this exercise you learn how to list the contents of a directory with the ls command.

Do This

Before you start, make sure you cd back to the directory above temp. If you have no idea where you are, use pwd to figure it out and then move there.

Linux/OS X**Exercise 6 Session**

```
$ cd temp
$ ls
stuff
$ cd stuff
$ ls
things
$ cd things
$ ls
frank
$ cd frank
$ ls
joe
$ cd joe
$ ls
alex
$ cd alex
$ ls
$ cd john
$ ls
$ cd ..
$ ls
john
$ cd ../../..
$ ls
frank
$ cd ../..
$ ls
stuff
$
```

Windows**Exercise 6 Windows Session**

```
> cd temp
> ls
```

```
Directory: C:\Users\zed\temp
```

Mode	LastWriteTime	Length	Name
----	-----	-----	---
d----	12/17/2011 9:03 AM		stuff

```
> cd stuff  
> ls
```

```
Directory: C:\Users\zed\temp\stuff
```

Mode	LastWriteTime	Length	Name
----	-----	-----	---
d----	12/17/2011 9:03 AM		things

```
> cd things  
> ls
```

```
Directory: C:\Users\zed\temp\stuff\things
```

Mode	LastWriteTime	Length	Name
----	-----	-----	---
d----	12/17/2011 9:03 AM		frank

```
> cd frank  
> ls
```

```
Directory: C:\Users\zed\temp\stuff\things\frank
```

Mode	LastWriteTime	Length	Name
----	-----	-----	---
d----	12/17/2011 9:03 AM		joe

```
> cd joe  
> ls
```

```
Directory: C:\Users\zed\temp\stuff\things\frank\joe
```

Mode	LastWriteTime	Length	Name
----	-----	-----	---
d----	12/17/2011 9:03 AM		alex

```
> cd alex  
> ls
```

```
Directory: C:\Users\zed\temp\stuff\things\frank\joe\alex
```

Mode	LastWriteTime	Length	Name
----	-----	-----	---
d----	12/17/2011 9:03 AM		john

```
> cd john  
> ls  
> cd ..  
> ls
```

```
Directory: C:\Users\zed\temp\stuff\things\frank\joe\alex
```

Mode	LastWriteTime	Length	Name
----	-----	-----	---
d----	12/17/2011 9:03 AM		john

```
> cd ..
> ls
```

```
Directory: C:\Users\zed\temp\stuff\things\frank\joe
```

Mode	LastWriteTime	Length	Name
----	-----	-----	---
d----	12/17/2011 9:03 AM		alex

```
> cd ../../..
> ls
```

```
Directory: C:\Users\zed\temp\stuff
```

Mode	LastWriteTime	Length	Name
----	-----	-----	---
d----	12/17/2011 9:03 AM		things

```
> cd ..
> ls
```

```
Directory: C:\Users\zed\temp
```

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
d----	12/17/2011 9:03 AM		stuff

>

You Learned This

The `ls` command lists out the contents of the directory you are currently in. You can see me use `cd` to change into different directories and then list what's in them so I know which directory to go to next.

There are a lot of options for the `ls` command, but you'll learn how to get help on those later when we cover the `help` command.

Do More

- *Type every one of these commands in!* You have to actually type these to learn them. Just reading them is *not* good enough. I'll stop yelling now.
- On UNIX, try the `ls -lR` command while you're in `temp`.
- On Windows, do the same thing with `dir -R`.
- Use `cd` to get to other directories on your computer and then use `ls` to see what's in them.
- Update your notebook with new questions. I know you probably have some, because I'm not covering everything about this command.
- Remember that if you get lost, then use `ls` and `pwd` to figure out where you are, then go to where you need to be with `cd`.

Remove Directory (`rmdir`)

In this exercise you learn how to remove an empty directory.

Do This

Linux/OS X

Exercise 7 Session

```
$ cd temp
$ ls
stuff
$ cd stuff/things/frank/joe/alex/john/
$ cd ..
$ rmdir john
$ cd ..
$ rmdir alex
$ cd ..
$ ls
joe
$ rmdir joe
$ cd ..
$ ls
frank
$ rmdir frank
$ cd ..
$ ls
things
$ rmdir things
$ cd ..
$ ls
stuff
$ rmdir stuff
$ pwd
~/temp
$
```

WARNING! If you try to do `rmdir` on Mac OS X and it refuses to remove the directory even though you are *positive* it's empty, then there is actually a file in there called `.DS_Store`. In that case, type `rm -rf <dir>` instead (replace `<dir>` with the directory name).

Windows**Exercise 7 Windows Session**

```
> cd temp  
> ls
```

Directory: C:\Users\zed\temp

Mode	LastWriteTime	Length	Name
----	-----	-----	---
d----	12/17/2011 9:03 AM		stuff

```
> cd stuff/things/frank/joe/alex/john/  
> cd ..  
> rmdir john  
> cd ..  
> rmdir alex  
> cd ..  
> rmdir joe  
> cd ..  
> rmdir frank  
> cd ..  
> ls
```

Directory: C:\Users\zed\temp\stuff

Mode	LastWriteTime	Length	Name
----	-----	-----	---
d----	12/17/2011 9:14 AM		things

```
> rmdir things  
> cd ..  
> ls
```

```
Directory: C:\Users\zed\temp
```

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
d----	12/17/2011 9:14 AM		stuff

```
> rmdir stuff  
> pwd
```

```
Path  
----  
C:\Users\zed\temp
```

```
> cd ..  
>
```

You Learned This

I'm now mixing up the commands so make sure you type them exactly and pay attention. Every time you make a mistake, it's because you aren't paying attention. If you find yourself making many mistakes, then take a break or just quit for the day. You've always got tomorrow to try again.

In this example you'll learn how to remove a directory. It's easy. You just go to the directory right above it, then type `rmdir <dir>`, replacing `<dir>` with the name of the directory to remove.

Do More

- Make 20 more directories and remove them all.
- Make a single path of directories that is 10 deep and remove them one at a time just like I did.
- If you try to remove a directory with content you will get an error. I'll show you how to remove these in later exercises.

Moving Around (pushd, popd)

In this exercise you learn how to save your current location and go to a new location with pushd. You then learn how to return to the saved location with popd.

Do This

Linux/OS X

Exercise 8 Session

```
$ cd temp
$ mkdir -p i/like/icecream
$ pushd i/like/icecream
~/temp/i/like/icecream ~/temp
$ popd
~/temp
$ pwd
~/temp
$ pushd i/like
~/temp/i/like ~/temp
$ pwd
~/temp/i/like
$ pushd icecream
~/temp/i/like/icecream ~/temp/i/like ~/temp
$ pwd
~/temp/i/like/icecream
$ popd
~/temp/i/like ~/temp
$ pwd
~/temp/i/like
$ popd
~/temp
$ pushd i/like/icecream
~/temp/i/like/icecream ~/temp
$ pushd
~/temp ~/temp/i/like/icecream
$ pwd
~/temp
$ pushd
```

```
~/temp/i/like/icecream ~/temp
$ pwd
~/temp/i/like/icecream
$
```

Windows

Exercise 8 Windows Session

```
> cd temp
> mkdir -p i/like/icecream
```

```
Directory: C:\Users\zed\temp\i\like
```

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
d----	12/20/2011 11:05 AM		icecream

```
> pushd i/like/icecream
> popd
> pwd
```

```
Path
-----
C:\Users\zed\temp
```

```
> pushd i/like
> pwd
```

```
Path
-----
C:\Users\zed\temp\i\like
```

```
> pushd icecream
> pwd
```

```
Path  
----  
C:\Users\zed\temp\i\like\icecream
```

```
> popd  
> pwd
```

```
Path  
----  
C:\Users\zed\temp\i\like
```

```
> popd  
>
```

You Learned This

You're getting into programmer territory with these commands, but they're so handy I have to teach them to you. These commands let you temporarily go to a different directory and then come back, easily switching between the two.

The `pushd` command takes your current directory and "pushes" it into a list for later, then it *changes* to another directory. It's like saying, "Save where I am, then go here."

The `popd` command takes the last directory you pushed and "pops" it off, taking you back there.

Finally, on UNIX, the command `pushd`, if you run it by itself with no arguments, will switch between your current directory and the last one you pushed. It's an easy way to switch between two directories. *This does not work in PowerShell.*

Do More

- Use these commands to move around directories all over your computer.
- Remove the `i/like/icecream` directories and make your own, then move around in them.
- Explain to yourself the output that `pushd` and `popd` will print out for you. Notice how it works like a stack?

- You already know this, but remember that `mkdir -p` will make an entire path even if all the directories don't exist. That's what I did very first for this exercise.

Making Empty Files (Touch, New-Item)

In this exercise you learn how to make an empty file using the `touch` (`new-item` on Windows) command.

Do This

Linux/OS X

Exercise 9 Session

```
$ cd temp  
$ touch iamcool.txt  
$ ls  
iamcool.txt  
$
```

Windows

Exercise 9 Windows Session

```
> cd temp  
> New-Item iamcool.txt -type file  
> ls
```

Directory: C:\Users\zed\temp

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
-a---	12/17/2011 9:03 AM		iamcool.txt

>

You Learned This

You learned how to make an empty file. On UNIX touch does this, and it also changes the times on the file. I rarely use it for anything other than making empty files. On Windows, you don't have this command, so you learned how to use the New-Item command, which does the same thing but can also make new directories.

Do More

- UNIX: Make a directory, change to it, and then make a file in it. Then move up one level and run the rmdir command in this directory. You *should* get an error. Try to understand why you got this error.
- Windows: Do the same thing, but you won't get an error. You'll get a prompt asking if you really want to remove the directory.

Copy a File (cp)

In this exercise you learn how to copy a file from one location to another with the cp command.

Do This

Linux/OS X

Exercise 10 Session

```
$ cd temp
$ cp iamcool.txt neat.txt
$ ls
iamcool.txt neat.txt
$ cp neat.txt awesome.txt
$ ls
awesome.txt iamcool.txt neat.txt
$ cp awesome.txt thefourthfile.txt
$ ls
awesome.txt iamcool.txt neat.txt thefourthfile.txt
$ mkdir something
$ cp awesome.txt something/
$ ls
awesome.txt iamcool.txt neat.txt something thefourthfile.txt
```

```
$ ls something/
awesome.txt
$ cp -r something newplace
$ ls newplace/
awesome.txt
$
```

Windows

Exercise 10 Windows Session

```
> cd temp
> cp iamcool.txt neat.txt
> ls
```

```
Directory: C:\Users\zed\temp
```

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
-a---	12/22/2011 4:49 PM	0	iamcool.txt
-a---	12/22/2011 4:49 PM	0	neat.txt

```
> cp neat.txt awesome.txt
> ls
```

```
Directory: C:\Users\zed\temp
```

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
-a---	12/22/2011 4:49 PM	0	awesome.txt
-a---	12/22/2011 4:49 PM	0	iamcool.txt
-a---	12/22/2011 4:49 PM	0	neat.txt

```
> cp awesome.txt thefourthfile.txt
> ls
```

```
Directory: C:\Users\zed\temp
```

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
-a---	12/22/2011 4:49 PM	0	awesome.txt
-a---	12/22/2011 4:49 PM	0	iamcool.txt
-a---	12/22/2011 4:49 PM	0	neat.txt
-a---	12/22/2011 4:49 PM	0	thefourthfile.txt

```
> mkdir something
```

```
Directory: C:\Users\zed\temp
```

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
d----	12/22/2011 4:52 PM		something

```
> cp awesome.txt something/  
> ls
```

```
Directory: C:\Users\zed\temp
```

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
d----	12/22/2011 4:52 PM		something
-a---	12/22/2011 4:49 PM	0	awesome.txt
-a---	12/22/2011 4:49 PM	0	iamcool.txt
-a---	12/22/2011 4:49 PM	0	neat.txt
-a---	12/22/2011 4:49 PM	0	thefourthfile.txt

```
> ls something
```

```
Directory: C:\Users\zed\temp\something
```

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
-a---	12/22/2011 4:49 PM	0	awesome.txt

```
> cp -recurse something newplace  
> ls newplace
```

```
Directory: C:\Users\zed\temp\newplace
```

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
-a---	12/22/2011 4:49 PM	0	awesome.txt

```
>
```

You Learned This

Now you can copy files. It's simple to just take a file and copy it to a new one. In this exercise I also make a new directory and copy a file into that directory.

I'm going to tell you a secret about programmers and system administrators now: they are lazy. I'm lazy. My friends are lazy. That's why we use computers. We like to make computers do boring things for us. In the exercises so far you have been typing repetitive boring commands so that you can learn them, but usually it's not like this. Usually if you find yourself doing something boring and repetitive, there's probably a programmer who has figured out how to make it easier. You just don't know about it.

The other thing about programmers is they aren't nearly as clever as you think. If you overthink what to type, then you'll probably get it wrong. Instead, try to imagine what the name of a command is and try it. Chances are that it's a name or some abbreviation similar to what you thought it was. If you still can't figure it out intuitively, then ask around and search online. Hopefully it's not something really stupid like ROBOCOPY.

Do More

- Use the `cp -r` command to copy more directories with files in them.
- Copy a file to your home directory or desktop.
- Find these files in your graphical user interface and open them in a text editor.
- Notice how sometimes I put a / (slash) at the end of a directory? That makes sure the file is really a directory, so if the directory doesn't exist I'll get an error.

Moving a File (`mv`)

In this exercise you learn how to move a file from one location to another using the `mv` command.

Do This

Linux/OS X

Exercise 11 Session

```
$ cd temp  
$ mv awesome.txt uncool.txt  
$ ls  
newplace uncool.txt  
$ mv newplace oldplace  
$ ls  
oldplace uncool.txt  
$ mv oldplace newplace  
$ ls  
newplace uncool.txt  
$
```

Windows

Exercise 11 Windows Session

```
> cd temp  
> mv awesome.txt uncool.txt  
> ls
```

```
Directory: C:\Users\zed\temp
```

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
d----	12/22/2011 4:52 PM		newplace
d----	12/22/2011 4:52 PM		something
-a---	12/22/2011 4:49 PM	0	iamcool.txt
-a---	12/22/2011 4:49 PM	0	neat.txt
-a---	12/22/2011 4:49 PM	0	thefourthfile.txt
-a---	12/22/2011 4:49 PM	0	uncool.txt

```
> mv newplace oldplace  
> ls
```

```
Directory: C:\Users\zed\temp
```

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
d----	12/22/2011 4:52 PM		oldplace
d----	12/22/2011 4:52 PM		something
-a---	12/22/2011 4:49 PM	0	iamcool.txt
-a---	12/22/2011 4:49 PM	0	neat.txt
-a---	12/22/2011 4:49 PM	0	thefourthfile.txt
-a---	12/22/2011 4:49 PM	0	uncool.txt

```
> mv oldplace newplace  
> ls newplace
```

```
Directory: C:\Users\zed\temp\newplace
```

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
-a---	12/22/2011 4:49 PM	0	awesome.txt

```
> ls
```

```
Directory: C:\Users\zed\temp
```

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
d----	12/22/2011 4:52 PM		newplace
d----	12/22/2011 4:52 PM		something
-a---	12/22/2011 4:49 PM	0	iamcool.txt
-a---	12/22/2011 4:49 PM	0	neat.txt
-a---	12/22/2011 4:49 PM	0	thefourthfile.txt
-a---	12/22/2011 4:49 PM	0	uncool.txt

```
>
```

You Learned This

Moving files or, rather, renaming them. It's easy: give the old name and the new name.

Do More

- Move a file in the newplace directory to another directory, then move it back.

View a File (less, MORE)

To do this exercise you're going to do some work using the commands you know so far. You'll also need a text editor that can make plain text (.txt) files. Here's what you do:

- Open your text editor and type some stuff into a new file. On OS X this could be TextWrangler. On Windows this might be Notepad++. On Linux this could be gedit. Any editor will work.

- Save that file to your desktop and name it `test.txt`.
- In your shell use the commands you know to *copy* this file to your `temp` directory that you've been working with.

Once you've done that, complete this exercise.

Do This

Linux/OS X

Exercise 12 Session

```
$ less test.txt  
[displays file here]  
$
```

That's it. To get out of `less`, just type `q` (as in quit).

Windows

Exercise 12 Windows Session

```
> more test.txt  
[displays file here]  
>
```

WARNING! In the output I'm showing [displays file here] to "abbreviate" what that program shows. I'll do this when I mean to say, "Showing you the output of this program is too complex, so just insert what you see on your computer here and pretend I did show it to you." Your screen will not actually show this.

You Learned This

This is one way to look at the contents of a file. It's useful because, if the file has many lines, it will "page" so that only one screenful at a time is visible. In the *Do More* section, you'll play with this some more.

Do More

- Open your text file again and repeatedly copy-paste the text so that it's about 50–100 lines long.
- Copy it to your `temp` directory again so you can look at it.

- Now do the exercise again, but this time page through it. On UNIX you use the spacebar and w (the letter w) to go down and up. Arrow keys also work. On Windows, just hit the spacebar to page through.
- Look at some of the empty files you created.
- The cp command will overwrite files that already exist, so be careful when copying files around.

Stream a File (cat)

You're going to do some more setup for this one so you get used to making files in one program and then accessing them from the command line. With the same text editor from the last exercise, create another file named `test2.txt` but this time save it directly to your `temp` directory.

Do This

Linux/OS X

Exercise 13 Session

```
$ less test2.txt  
[displays file here]  
$ cat test2.txt  
I am a fun guy.  
Don't you know why?  
Because I make poems,  
that make babies cry.  
$ cat test.txt  
Hi there this is cool.  
$
```

Windows

Exercise 13 Windows Session

```
> more test2.txt  
[displays file here]  
> cat test2.txt  
I am a fun guy.  
Don't you know why?
```

```
Because I make poems,  
that make babies cry.  
> cat test.txt  
Hi there this is cool.  
>
```

Remember that when I say [displays file here], I'm abbreviating the output of that command so I don't have to show you exactly everything.

You Learned This

Do you like my poem? Totally going to win a Nobel. Anyway, you already know the first command, and I'm just having you check that your file is there. Then you cat the file to the screen. This command spews the whole file to the screen with no paging or stopping. To demonstrate that, I have you do this to the test.txt, which should just spew a bunch of lines from that exercise.

Do More

- Make a few more text files and work with cat.
- UNIX: Try cat test.txt test2.txt and see what it does.
- Windows: Try cat test.txt,test2.txt and see what it does.

Removing a File (rm)

In this exercise you learn how to remove (delete) a file using the rm command.

Do This

Linux/OS X

Exercise 14 Session

```
$ cd temp  
$ ls  
uncool.txt iamcool.txt neat.txt something thefourthfile.txt  
$ rm uncool.txt  
$ ls
```

```
iamcool.txt neat.txt something thefourthfile.txt
$ rm iamcool.txt neat.txt thefourthfile.txt
$ ls
something
$ cp -r something newplace
$
$ rm something/awesome.txt
$ rmdir something
$ rm -rf newplace
$ ls
$
```

Windows

Exercise 14 Windows Session

```
> cd temp
> ls
```

Directory: C:\Users\zed\temp

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
d----	12/22/2011 4:52 PM		newplace
d----	12/22/2011 4:52 PM		something
-a---	12/22/2011 4:49 PM	0	iamcool.txt
-a---	12/22/2011 4:49 PM	0	neat.txt
-a---	12/22/2011 4:49 PM	0	thefourthfile.txt
-a---	12/22/2011 4:49 PM	0	uncool.txt

```
> rm uncool.txt
> ls
```

```
Directory: C:\Users\zed\temp
```

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
d----	12/22/2011 4:52 PM		newplace
d----	12/22/2011 4:52 PM		something
-a---	12/22/2011 4:49 PM	0	iamcool.txt
-a---	12/22/2011 4:49 PM	0	neat.txt
-a---	12/22/2011 4:49 PM	0	thefourthfile.txt

```
> rm iamcool.txt  
> rm neat.txt  
> rm thefourthfile.txt  
> ls
```

```
Directory: C:\Users\zed\temp
```

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
d----	12/22/2011 4:52 PM		newplace
d----	12/22/2011 4:52 PM		something

```
> cp -r something newplace  
> rm something/awesome.txt  
> rmdir something  
> rm -r newplace  
> ls  
>
```

You Learned This

Here we clean up the files from the last exercise. Remember when I had you try to `rmdir` on a directory with something in it? Well, that failed because you can't remove a directory with files in it. Instead you have to remove the file, or recursively delete all of its contents. That's what you did at the end of this example.

Do More

- Clean up everything in temp from all the exercises so far.
- Write in your notebook to be careful when running recursive remove commands on files.

Exiting Your Terminal (exit)

Do This

Linux/OS X

Exercise 23 Session

```
$ exit
```

Windows

Exercise 23 Windows Session

```
> exit
```

You Learned This

Your final exercise is how to exit a Terminal. Again, this is very easy, but I'm going to have you do more.

Do More

For your last set of exercises, I want you to use the help system to look up a set of commands you should research and learn how to use on your own.

Here's the list for UNIX:

- xargs
- sudo
- chmod
- chown

For Windows, look up these things:

- `forfiles`
- `runas`
- `attrib`
- `icacls`

Find out what these are, play with them, and then add them to your index cards.

Command Line Next Steps

You have completed the crash course. At this point you should be a barely capable shell user. There's a huge list of tricks and key sequences you don't know yet, and I'm going to give you a few final places to go research more.

UNIX Bash References

The shell you've been using is called bash. It's not the greatest shell but it's everywhere and has a lot of features so it's a good start. Here's a short list of links about bash you should visit:

Bash Cheat Sheet http://cli.learncodethehardway.org/bash_cheat_sheet.pdf created by Raphael and CC licensed.

Reference Manual <http://www.gnu.org/software/bash/manual/bashref.html>

PowerShell References

On Windows, there's really only PowerShell. Here's a list of useful links related to PowerShell:

Owner's Manual <http://technet.microsoft.com/en-us/library/ee221100.aspx>

Cheat Sheet <http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=7097>

Master PowerShell <http://powershell.com/cs/blogs/ebook/default.aspx>

Index

Symbols

" (double quotes)
 escaping, 38
 strings and, 28, 33
 variables and, 30
 ' (single quotes)
 escaping, 38
 strings and, 33
 variables and, 30
 " " " (triple quotes), 37–38
 / (forward-slash) characters, 38–40
 \ (backslash) characters, 38–40
 _ (underscore) characters, 24
 || (or) expressions, 91–96
 += (increment by) operators, 71–72, 99
 = (equal) characters
 == vs., 25
 ARGV and, 46–51
 in asking questions of users, 42
 escaping quotes, 38
 format strings and, 34–37
 naming variables with, 24–26
 printing variables with, 28–29
 returning values from functions with, 74–76
 setting variables to numbers, 44–45
 setting variables to strings, 30–33
 == (double equal) characters, 25, 92–96
 ! (not) expressions, 91–96
 != (not equal) characters, 91–96
 # (pound) characters, 18–19
 ## ?? comments, 159–161
 #{ } (format activators), 28, 35
 % (modulus) operators, 22, 35
 %{} (format activators), 35
 && (and) expressions, 91–96
 *args (asterisk args), 63–65
 @ (object scope), 125, 149–151

[(left bracket), opening arrays with, 106–108
] (right bracket), closing arrays with, 106–108
 > (prompts), 50–51

Numbers

2-dimensional (2D) arrays, 108
 8080 port, 219–223

A

Addresses, defined, 225
 Advanced user input
 exceptions in, 206
 game lexicon and, 204–206
 introduction to, 204
 lexicon types in, 205
 numbers in, 206
 questions about, 210
 sentence breaks in, 204–205
 Study Drills on, 209
 testing first, 206–207
 testing procedures, 207–209
 writing scanners in, 205
 Advice for programmers, 246–247
 After inheritance, 180–182
 Algorithms, 140
 Analysis
 of game engines. *See* Game engine analysis
 of games, 188–189, 191
 object-oriented. *See* Object-oriented analysis
 top down versus bottom up, 170–176
 And (&&) expressions, 91–96
 Argument variable (ARGV). *See* ARGV (argument variable)
 ArgumentError, 206
 Arguments
 *args, 63–65
 ARGV. *See* ARGV (argument variable)

- Arguments (*Continued*)
 - arrays and, 128
 - class style and, 189
 - in command line, 46–47
 - in functions, 62–68, 75
- ARGV (argument variable)
 - introduction to, 46–48
 - opening files with, 52–54
 - prompting and passing with, 50–51
- Arrays
 - 2-dimensional, 108
 - accessing elements of, 114–115
 - bucket, 140–142
 - closing, 106–108
 - data structures and, 129–130
 - hashes vs., 132–133, 144
 - levels of, 142–144
 - loops and, 106–112
 - opening, 106–108
 - questions about, 108–109, 131
 - strings and, 128–129
 - Study Drills on, 108, 131
 - when to use, 130–131, 143
- Asking questions of users
 - decision making and, 102–104
 - overview of, 42
 - questions about, 43
 - Study Drills on, 43
- assert commands, 144, 217
- Association, 133–136
- Asterisk args (*args), 63–65
- Automated testing
 - of forms, 229–230
 - guidelines for, 201
 - introduction to, 198
 - questions about, 202
 - results of, 201
 - Study Drills on, 202
 - writing, 198–201
- B**
- Backslash (\) characters, 38–40
- Before inheritance, 180–182
- begin-rescue, 206, 210
- bin/ folders, 192–196
- Bitbucket.com, 80
- Boolean logic
 - overview of, 94–96
 - questions about, 96
 - Study Drills on, 96
 - true/false in, 92
- Branches
 - overview of, 116–119
 - questions about, 119
 - Study Drills on, 118
- Browsers
 - automated testing of forms and, 229–230
 - defined, 225
 - HTML forms and, 226–228
 - input from, generally, 224
 - layout templates and, 228–229
 - questions about, 231
 - Study Drills on, 230–231
 - web workings and, 224–226
 - websites and, 220
- Bucket arrays, 140–142
- C**
- cat (stream file) command, 285–286
- cd (change directory), 260–264
- Chef Solo, 242
- Child classes, 179–186
- chomp, 42–48
- Classes
 - Child, 179–186
 - creating, 148–149
 - in game engine analysis, 167–169
 - hierarchies of, 167–168
 - introduction to, 146–148
 - modules and, 148–149
 - nil and, 160–162
 - in object-oriented programming, generally, 146–151
 - objects vs., 158
 - Parent, 179–186
 - in phrases, 152–153

- style of, 189–190
testing, 168–169
- `close` command, 56
- Closing arrays with right bracket (`[]`), 106–108
- Command line tools
- `ARGV`. See `ARGV` (argument variable)
 - `cat` (stream file), 285–286
 - `cd` (change directory), 260–264
 - `exit` (exit terminal), 289–290
 - `in` Linux. See Linux
 - `ls` (list directory), 264–269
 - `in` Mac OS X. See Mac OS X
 - memorizing, 250
 - `mkdir` (make directory), 257–259
 - `mv` (move file), 281
 - `new-item` (create empty file), 276–277
 - `popd` (return to saved location),
273–276
 - `pushd` (save location, go to new location),
273–276
 - `pwd` command, 254–256
 - references on, 290
 - `rm` (remove file), 286–289
 - `rmdir` (remove directory), 269–272
 - setup for, 251
 - shells, 249
 - `touch` (create empty file), 276–277
 - viewing files with, 283–285
 - for when lost, 256–257
 - in Windows. See Windows
- Comments
- `## ??` 159–161
 - in English, 18, 24, 33
 - in game creation, 190–191
 - overview of, 18
 - questions about, 18–19
 - Study Drills on, 18–19
- Connections, defined, 225
- Copying files
- `cp` for, 277–281
 - overview of, 60–61
 - questions about, 61
 - Study Drills on, 61
- Copy-pasting, 3
- Correcting bad code, 88
- `cp` (copy file) command, 277–281
- D**
- Data structures, 129–130, 136
- Data types, 124
- Debugging games, 120–121
- Decision making
- exercises in, 102–103
 - questions about, 103–104
 - Study Drills on, 103
- `def` (define function), 63, 152
- `delete`, 141
- Designing games, 120–121
- Details, significance of, 2
- Dictionaries, 132
- Differences, significance of, 3
- Directories. See Skeleton directories
- Double equal (==) characters, 25,
92–96
- Double quotes (""). See " (double quotes)
- Drawing problems, 165–166
- E**
- Ease of learning, 2–5
- Elements of arrays, access to, 114–115
- Eloquent Ruby*, 242
- `else` conditions
- `elsif` as. See `elsif`
 - making decisions with, 102–104
 - overview of, 100–101
 - questions about, 101
 - Study Drills on, 101
- `elsif`
- branches and, 116–117
 - definition of, 123
 - introduction to, 101
 - making decisions with, 102–104
 - rules for, 120
- English
- comments in, 18, 24, 33
 - in object-oriented programming, 156

- Equality operators
`!=` (not equal), 91–96
`=` (single equal). See **`=`** (equal) characters
`==` (double equal), 25, 92–96
 in Boolean logic, 95–96
- Errors
 debugging games for, 120–121
 exceptions as. See Exceptions
 in `if-` statements, 100, 120
 load, 17
 name, 25
`ParserError`, 213–216
 in prompt variables, 51
`rake test` and, 201–202
 reading messages about, 16
 spelling, 37
 syntax. See Syntax errors
 tokens, 205
 in unpacking variables, 47
 in website creation, 221–222
- Escape sequences
 introduction to, 38
 overview of, 38–39
 questions about, 40
 string, 124
 Study Drills on, 40
- Evaluation
 of game engines. See Game engine
 analysis of games, 188–189, 191
 object-oriented. See Object-oriented
 analysis
- Exceptions
 numbers and, 206
 in sentences, 213
 symbols for, 122–123
- `exit` commands, 116–119, 289–290
- Explicitly overriding inheritance, 180
- Extracting game concepts, 166. See also
 Game engine analysis
- F
- `f` variable, 70–71
- False
 in Boolean logic, 95–96
- format strings and, 34–35
 math operators and, 20–21
 overview of, 90–92
- `File.exist?(to_file)`, 60–61
- Files
 copying, 60–61, 277–281
 creating empty, 276–277
 functions and, 70–72
 moving, 281
 opening, 52–58
 questions about, 71–72
 reading, 52–58
 reading backward, 24–26
 removing, 286–289
 running, 13, 15
 streaming, 285–286
 Study Drills on, 71
 viewing, 283–285
 writing, 56–58
- First programs
 on Mac OS X, 12, 14
 overview of, 12
 questions about, 17
 Study Drills on, 16
 on Windows, 13, 15
- Floating point numbers, 21
- for-loop
 exercises in, 106–108
 questions about, 108–109
 rules for, 120
 Study Drills on, 108
`while-loop` vs., 112
- Format activators, 28, 35
- Format strings
`=` characters and, 34–37
 overview of, 34
 printing, 34
 questions about, 35
 Study Drills on, 35
- Forms
 automated tests for, 229–230
 HTML, 227–231

- layout templates for, 228
- overview of, 226–227
- Forward-slash (/),** 38–40
- Frequently asked questions**
 - on # (pound) characters, 18–19
 - on advanced user input, 210
 - on arrays, 108–109, 131
 - on asking questions of users, 43
 - on automated testing, 202
 - on Boolean logic, 96
 - on branches, 119
 - on browsers, 231
 - on comments, 18–19
 - on copying files, 61
 - on else conditions, 101
 - on escape sequences, 40
 - on first programs, 17
 - on format strings, 35
 - on functions, generally, 65
 - on functions, values from, 76
 - on functions and files, 71–72
 - on functions and variables, 67–68, 87
 - on has-a phrases, 161–162
 - on hashes, 144
 - on if- statements, 99
 - on inheritance, 186–187
 - on is-a phrases, 161–162
 - on for-loop, 108–109
 - on making decisions, 103–104
 - on math, 22
 - on memorizing logic, 92
 - on names, 25–26
 - on numbers, 22
 - on object-oriented analysis, 177
 - on object-oriented programming, 151, 156
 - on practicing code, 83, 87
 - on printing, 33, 37
 - on prompting and passing, 51
 - on reading and writing files, 54, 58
 - on sentences, 217
 - on skeleton directories, 195–196
 - on strings, 31
 - on symbols, 127
 - on text, 31
 - on variables, 25–26, 47–48
 - on website creation, 223
 - on while-loop, 112
- Functions**
 - branches and, 115–119
 - checklists for, 64, 70
 - code and, 62–65
 - def (define function) for, 152
 - files and, 70–72
 - in game creation, 189
 - importing and running, 84–86
 - names and, 62–65
 - overview of, 62–64
 - questions about, 65, 71–72, 76
 - Study Drills on, 64, 71, 75–76
 - style of, 189
 - values and, 74–76
 - variables and, 62–68, 84–87
- G**
- Game engine analysis.** *See also Games*
 - class hierarchies in, 167–168
 - coding classes in, 168
 - extracting/researching concepts in, 166
 - object maps in, 167–168
 - refining code in, 169–170
 - testing classes in, 168–169
 - top down versus bottom up approach to, 170–176
 - writing/drawing problems in, 165–166
- Game engines**
 - analysis of. *See Game engine analysis*
 - creation of, 238–240
 - refactoring of, 232–237
 - session-based, 238–241
- Game lexicon**
 - exceptions in, 206
 - introduction to, 204
 - lexicon types in, 205
 - numbers in, 206
 - sentence breaks in, 204–205
 - writing scanners and, 205

- G**
- Games
 - analysis of engines for. *See Game engine analysis*
 - arrays in, 129–131
 - branches in, 116–119
 - class style in, 189–190
 - code style in, 190
 - comments in, 190–191
 - debugging, 120–121
 - designing, 120–121
 - engines for. *See Game engines*
 - evaluating, 188–189, 191
 - function style in, 189
 - functions in, generally, 116–119
 - introduction to, 188
 - lexicon in. *See Game lexicon*
 - user input in. *See Advanced user input on the web. See Web games*
 - gedit text editor
 - setup and, 8–9, 11
 - viewing files in, 283
 - gem, 218–219
 - get commands, 141
 - gets.chomp
 - asking questions of users with, 42–48
 - opening files with, 52–54
 - prompting and passing with, 50–51
 - Github.com, 80
 - Gitorious.org, 80
 - Global variables, 68
 - Google, 10
 - "Gothons from Planet Percal #25," 170–176
 - Grammar, 213
- H**
- Handlers, 222–223, 227
 - Hard coding, 52
 - Hard way overview
 - copy-pasting vs., 3
 - details, significance of, 2
 - differences, significance of, 3
 - ease of learning in, 2
 - instructional videos, 3
- I**
- idiomatic Ruby, 11
 - if- statements
 - arrays and, 106
 - else conditions and, 100–101
 - exercises in, 98–99
 - making decisions with, 102–104
 - questions about, 99
 - rules for, 120
 - Study Drills on, 99
 - Implicit inheritance, 179
 - Increment by (+=) operators, 71–72, 99
 - Index.GET, 221–224
 - Inheritance
 - altering before/after, 180–182
 - combining types of, 182–183

- composition vs., 183–186
 definition of, 178–179
 implicit, 179
 introduction to, 178
 overriding explicitly, 180
 questions about, 186–187
 Study Drills on, 186
`super()` with `initialize`, 183
 when to use, 185–186
- i**
initialize, 183
Input methods, 42–47
Instantiating classes, 148–149
Instructional videos, 3
Integer() functions, 206
Intended readers, 4
International programming, 16
Internet
 browsers and. *See* *Browsers*
 games on. *See* *Web games*
 searching, 10
- Irb** interpreter, 53–54, 85–86
is-a phrases
 exercises in, 159–161
 introduction to, 158–159
 questions about, 161–162
 Study Drills on, 161
- K**
Keywords, 122–123
- L**
Launchpad.net, 80
Layout templates, 228–229
Learn C the Hard Way, 242
Learn Python the Hard Way, 4, 242
Learn Ruby the Hard Way, 3–4
Learning programming languages, 243–244
Learning Ruby, overview. *See* *Hard way*
 overview
Left bracket ([]), opening arrays with, 106–108
Lexicon. *See* *Game lexicon*
- Linux**
`cat` (stream file) in, 285
`cd` (change directory) in, 260–261
 command line tools in, 252–253
`cp` (copy file) in, 277–278
`exit` (exit terminal) in, 289
`less` command in, 284
`ls` (list directory) in, 265
`mkdir` (make directory) in, 257
`mv` (move file) in, 281
`popd` (return to saved location) in, 273–274
`pushd` (save location, go to new location) in, 273–274
`pwd` command in, 255
`rm` (remove file) in, 286–287
`rmdir` (remove directory) in, 270
`setup` on, 8–10
 Terminal setup in, 251
`touch` (create empty file) in, 276–277
- Lisp**, 5
list, 141
List directory (ls), 264–269
localhost, 220–225
Loops
 arrays and, 106–112
`for-loop`, 106–108, 112, 120
`while-loop`, 110–112, 120, 169
ls (list directory), 264–269
- M**
Mac OS X
`cat` (stream file) in, 285
`cd` (change directory) in, 260–261
 command line tools in, generally, 252–253
`cp` (copy file) in, 277–278
`exit` (exit terminal) in, 289
 first programs on, 12, 14
`less` command in, 284
`ls` (list directory) in, 265
`mkdir` (make directory) in, 257
`mv` (move file) in, 281
`popd` (return to saved location) in, 273–274

- Mac OS X (Continued)**
- pushd (save location, go to new location) in, 273–274
 - pwd command in, 255
 - rmdir (remove directory) in, 270
 - running files on, 14
 - setup on, 6–7
 - Terminal setup in, 251
 - touch (create empty file) in, 276–277
- Make directory (`mkdir`)**, 257–259
- Mapping**
- of hashes, 133–136
 - of objects, 167–168
 - in refactoring, 232–237
- Match and peek**, 212–213
- Math. See also Numbers**
- algorithms for data structures, 140
 - overview of, 20–21
 - questions about, 22
 - Study Drills on, 21
- Mechanize**, 242
- Memorization**
- of characters, 78
 - of command line tools, 250
 - of logic, 90–92
 - of truth tables, 91–92
 - of truth terms, 90–91
- mkdir (make directory)**, 257–259
- Modules**
- classes and, 148
 - hashes and, 136–139, 146–148
 - in object-oriented programming, generally, 146–151
- Modulus (%) operators**, 22, 35
- mv (move file) command**, 281
- N**
- Names**
- functions and, 62–65
 - overview of, 24–25
 - questions about, 25–26
 - Study Drills on, 25
- new (create initializer) command**, 140
- new-item (create empty file) command**, 276–277
- nil**
- classes and, 160–162
 - as data type, 124
 - hashes and, 132–139
- Not (!) expressions**, 91–96
- Not equal (!=) characters**, 91–96
- Notepad++**
- first programs in, 13
 - setup and, 7
 - viewing files in, 283
- Numbers. See also Math**
- in game lexicon, 206
 - overview of, 20–21
 - questions about, 22
 - Study Drills on, 21
- O**
- Object maps**, 167–168
- Object scope (@)**, 125
- Object-oriented analysis**
- introduction to, 164–165
 - questions about, 177
 - of simple game engines. *See Game engine analysis*
 - Study Drills on, 176–177
 - top down versus bottom up approach in, 170–176
- Object-oriented programming (OOP)**
- analysis in. *See Object-oriented analysis*
 - classes in, 148–149
 - English option in, 156
 - examples of, 150–151
 - exercises in, 152
 - getting things from things in, 150
 - hashes in, 146–148
 - introduction to, 148–149
 - modules in, 146–149
 - objects in, 148–149
 - phrase drills for, 152–153
 - questions about, 151, 156
 - reading code in, 153–156

- require in, 148–149
 Study Drills on, 151
 word drills for, 152
- Objects**
 @ for, 125
 classes vs., 158–161
 instantiating, 148–149
 maps of, 167–168
 in object-oriented programming, generally, 146–151
 require and, 148–149
- Octothorpe**, 18–19
- OED (*Oxford English Dictionary*)**, 136
- OOP (object-oriented programming)**. See Object-oriented programming (OOP)
- open(filename)**, 52–58
- Opening arrays with left bracket ([])**, 106–108
- Operators**, defined, 125. *See also specific operators*
- Or (| |) expressions**, 91–96
- Overriding inheritance**, 180
- Oxford English Dictionary (OED)**, 136
- P**
- Padrino**, 242
- Parameters**, 46–47
- Parent classes**, 179–186
- Parentheses Exponents Multiplication Division Addition Subtraction (PEMDAS)**, 22
- Parsers**, 213–216
- PEMDAS (Parentheses Exponents Multiplication Division Addition Subtraction)**, 22
- Persistence**, 3–4
- Phrase drills**, 152–153
- popd (return to saved location)**, 273–276
- Port 8080**, 219–223
- Pound (#) characters**, *See # (pound) characters*
- PowerShell**
 first programs in, 15–16
 references on, 290
 running files in, 15
 setup and, 6–8
- Practicing code**
 exercises in, 82–86
 importance of, 3
 questions about, 83, 87
 Study Drills on, 83, 87
- Printing**
 format strings, 34–35
 practice exercises in, 32–33, 36–37
 print for, 42
 print_two for, 62
 pwd (print working directory) for, 254–256
 questions about, 33, 37
 Study Drills on, 33, 37
 variables, 28–29
- Programmers**
 advice for, 246–247
 bad versus good, 86
 details, significance of, 2
 differences, significance of, 3
 warnings for, 4–5
- Programming languages**, 243–244
- Project skeleton directories**. *See Skeleton directories*
- Prompting**
 > for, 50–51
 for numbers, 44
 passing and, 50–51
 questions about, 51
 Study Drills on, 44, 51
- pushd (save location, go to new location)**, 273–276
- puts command**
 comments and, 18–19
 format strings and, 34–36
 introduction to, 12–13
 math symbols and, 20–21
 strings and, 30–32
- pwd (print working directory)**, 254–256
- Q**
- Questions asked by students**. *See Frequently asked questions*

- Questions asked of users. *See Asking questions of users*
- R**
- rake test
 - automated testing with, 199–202
 - setting up, 194
 - syntax errors in, 202
 - websites and, 230, 232
 - Rakefile, 193–194, 199–202
 - .rb suffix, 12
 - Reading code
 - for files. *See Reading files*
 - in object-oriented programming, 153–156
 - resources for, 80–81
 - in Ruby, generally, 2
 - symbols in, 126
 - Reading files
 - backward, 24–26
 - exercises in, 52–53, 56–57
 - questions about, 54, 58
 - read command for, 56
 - readline command for, 56
 - Study Drills on, 53, 57–58
 - Refactoring game engines, 232–237
 - References, 242, 290
 - Refining code, 169–170
 - Remove directory (rmdir), 269–272
 - Remove file (rm), 286–289
 - Requests, defined, 225–226
 - require, 148–149
 - Researching game concepts, 166. *See also Game engine analysis*
 - Responses, defined, 226
 - Return to saved location (popd), 273–276
 - Right bracket ([]), closing arrays with, 106–108
 - rm (remove file), 286–289
 - rmdir (remove directory), 269–272
 - Room class, 232–237
 - Ruby, introduction to
 - first programs in, 12–17
 - idiomatic Ruby, 11
 - learning, generally. *See Hard way overview*
 - setting up, 6–11
 - Ruby on Rails, 242
 - RubyMotion, 242
 - Ruby-Processing, 242
 - Rules
 - for if-statements, 120
 - for for-loop, 120
 - for while-loop, 120
 - Running files, 13, 15
- S**
- Save location, go to new location (pushd), 273–276
 - Scanners, 205
 - Searching Internet, 10
 - Sentences
 - breaks in, 204–205
 - creating, generally, 212
 - exceptions in, 213
 - grammar in, 213
 - match and peek in, 212–213
 - parsers in, 213–216
 - questions about, 217
 - Study Drills on, 217
 - testing of, 217
 - Servers, defined, 226
 - Session-based game engines, 238–241
 - Sessions, 237–238
 - set, 141–142
 - Setup
 - for command line tools, 251
 - generally, 6
 - Internet searches for, 10
 - on Linux, 8–10
 - on Mac OS X, 6–7
 - warnings about, 10–11
 - on Windows, 7–8
 - Shells, 249
 - Sinatra
 - browser interactions with, 220
 - errors in, 221–222
 - “Hello World” project in, 219–220

- installing, 218–219
 - stopping/reloading, 221
 - templates in, 222–223
 - Single equal (=) characters. *See also* = (equal) characters
 - Skeleton directories
 - creating, 192–193
 - final structure of, 193–194
 - introduction to, 192
 - questions about, 195–196
 - quiz on, 195
 - testing setup of, 194
 - using, 195
 - Sourceforge.net, 80
 - Stateless sessions, 237
 - Stream file (cat) command, 285–286
 - String escape sequences, 124. *See also* Escape sequences
 - Strings
 - arrays and, 128–129
 - escape sequences and, 38
 - format, 34–35
 - questions about, 31
 - Study Drills on, 31
 - text and, 30–31
 - variables and, 28–29
 - Student questions. *See* Frequently asked questions
 - Study Drills
 - on # characters, 18–19
 - on accessing elements of arrays, 115
 - on advanced user input, 209
 - on arrays, 108, 131
 - on asking questions of users, 43
 - on automated testing, 202
 - on Boolean logic, 96
 - on branches, 118
 - on browsers, 230–231
 - on comments, 18–19
 - on copying files, 61
 - on else conditions, 101
 - on escape sequences, 40
 - on first programs, 16
 - on functions, generally, 64
 - on functions and files, 71
 - on functions and values, 75–76
 - on functions and variables, 67, 87
 - on has-a phrases, 161
 - on hashes, 143–144
 - on if- statements, 99
 - on inheritance, 186
 - on is-a phrases, 161
 - on for-loop, 108
 - on making decisions, 103
 - on math, 21
 - on names, 25
 - on numbers, 21
 - on object-oriented analysis, 176–177
 - on object-oriented programming, 151
 - on practicing code, 83, 87
 - on printing, 29, 33, 37
 - on prompting and passing, 51
 - on prompting for numbers, 44
 - on reading files, 53, 57–58
 - on sentences, 217
 - on strings, 31
 - on symbols, 126–127
 - on text, 31
 - on variables, 25, 29, 47
 - on website creation, 223
 - on while-loop, 112
 - on writing files, 57–58
- Style, 189–190
- sudo gem install, 218–221, 229
- super() with initialize, 183
- super(name), 160–162
- Symbols. *See also* specific symbols
 - data types, 124
 - introduction to, 122
 - keywords, 122–123
 - operators, 125
 - questions about, 127
 - reading code and, 126
 - string escape sequences, 124
- Study Drills on, 126–127

- Syntax errors
 branches and, 118
 defined, 16
 in if- statements, 100
`rake test` and, 202
- T**
 Templates, 222–223, 228–229
 Terminal
 Boolean logic in, 94
 first programs in, 13–16
 running files in, 14
 setup and, 6–11
 Testing
 automated. *See* Automated testing
 classes, in game engine analysis, 168–169
 correcting bad code and, 88
 procedures for, 207–209
 rake for. *See* rake test
 in sentence creation, 217
 of skeleton directory setup, 194
 user input and, 206–209
- Text
 questions about, 31
 strings and, 30–31
 Study Drills on, 31
- Text editors
 creating, 56–58
 gedit, 8–11, 283
 Notepad++ as. *See* Notepad++
 running files on, 12–17
 setting up, 6–11
 Terminal as. *See* Terminal
 TextWrangler as. *See* TextWrangler
- TextWrangler
 first programs in, 12
 setup and, 6, 11
 viewing files in, 283
- Thor, 242
- Top down versus bottom up analysis, 170–176.
See also Object-oriented analysis
- touch (create empty file) command, 276–277
- Tracking sessions/users, 237–238
- Triple quotes (" " "), 37–38
- True
 format strings and, 34–35
 math operators and, 20–21
 strings and, 31
- truncate command, 56–58
- Truth tables, 91–92
- Truth terms, 90–91
- `txt = open(filename)`, 52–54
- Typing, importance of, 2
- U**
 Underscore (_) characters, 24
 UNIX Bash, 290
 Unpacking arguments/variables, 44, 46–48
- Upgrades, 240–241
- URLs, 220–225
- Users
 asking questions of, 42–43, 102–104
 input of. *See* Advanced user input
 tracking, 237–238
- V**
 Values, 74–76, 121
- Variables
 ARGV. *See* ARGV (argument variable)
 f, 70–71
 functions and, 62–68, 84–87
 global, 68
 naming, 24–26
 overview of, 24–25
 passing to scripts, 46–47
 printing, 28–29
 questions about, 25–26, 47–48, 67–68
 setting to numbers, 44–45
 setting to strings, 30–33
 strings and, 28–33
 Study Drills on, 25, 47, 67
- Videos, 3
- Viewing files, 283–285

W

'w, ' 56–61
Warnings, 4–5, 10–11

Web games
 engine creation for, 238–240
 introduction to, 232
 refactoring engines for, 232–237
 sessions in, 237–238
 tracking users in, 237–238
 upgrades to, 240–241

Websites
 browsers and. *See* Browsers
 creating, generally, 218
 error repair in, 221–222
 “Hello World” project, 219–220
 questions about, 223
 Sinatra for, installing, 218–219
 Sinatra for, stopping/reloading, 221
 Study Drills on, 223
 template creation in, 222–223

The Well-Grounded Rubyist, 242
while true, 116–119
while-loop

 exercises in, 110–112
 in game creation, 169
 for-loop vs., 112
 questions about, 112
 rules for, 120
 Study Drills on, 112

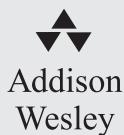
Windows
 cat (stream file) in, 285–286
 cd (change directory) in, 261–263

command line tools in, 253–254
cp (copy file) in, 278–280
exit (exit terminal) in, 285–286
first programs on, 13, 15
ls (list directory) in, 265–269
mkdir (make directory) in, 258–259
more command in, 284
mv (move file) in, 281–283
new-item (create empty file) in,
 276–277
popd (return to saved location), 274–275
PowerShell setup in, 252
pushd (save location, go to new location),
 274–275
pwd command in, 255
rm (remove file) in, 287–288
rmdir (remove directory) in, 271–272
running files on, 15
setup on, 7–8
Word drills, 152
Writing files
 exercises in, 56–57
 questions about, 58
 Study Drills on, 57–58
 write('stuff') command for, 56
Writing problems, 165–166
Writing scanners, 205
Writing skills, importance of, 2

Z

Zen koans, 158

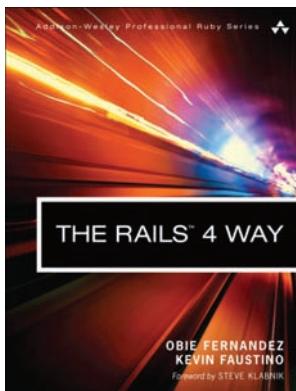
This page intentionally left blank



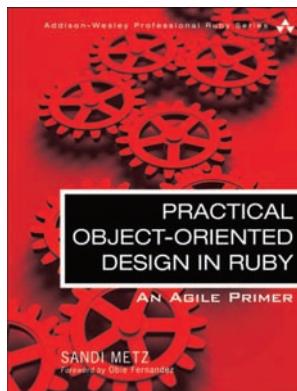
The Addison-Wesley Professional Ruby Series

informit.com/ruby

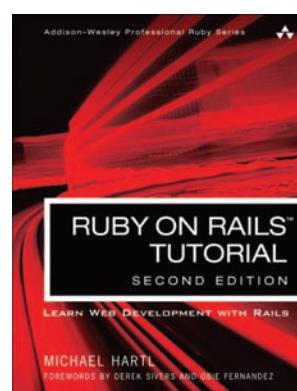
**Expert reference books, eBooks, and videos
from experienced practitioners**



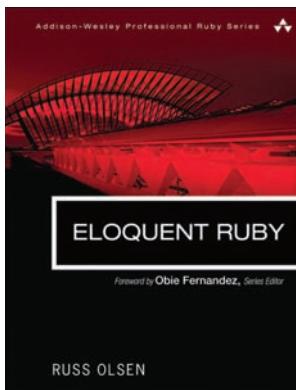
ISBN-13: 978-0-321-94427-6



ISBN-13: 978-0-321-72133-4



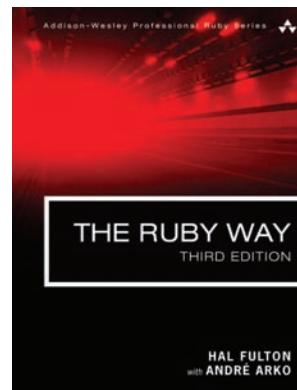
ISBN-13: 978-0-321-83205-4



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



ISBN-13: 978-0-13-303999-3



ISBN-13: 978-0-132-80826-2



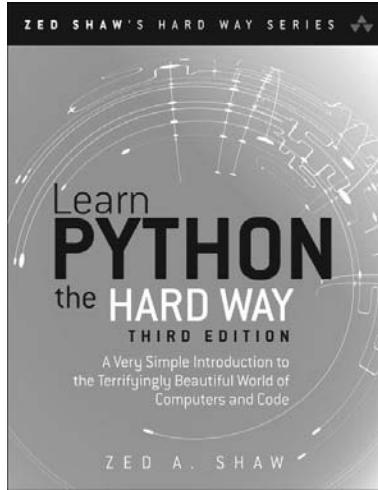
Look for these titles and more on informit.com/ruby.

Available in print and eBook formats



You Will Learn Python!

informit.com/hardway



Zed Shaw has perfected the world's best system for learning Python. Follow it and you will succeed—just like the hundreds of thousands of beginners Zed has taught to date! You bring the discipline, commitment, and persistence; the author supplies everything else.

In *Learn Python the Hard Way, Third Edition*, you'll learn Python by working through 52 brilliantly

crafted exercises. Read them. Type their code precisely. (No copying and pasting!) Fix your mistakes. Watch the programs run. As you do, you'll learn how software works; what good programs look like; how to read, write, and think about code; and how to find and fix your mistakes using tricks professional programmers use.

This tutorial will reward you for every minute you put into it. Soon, you'll know one of the world's most powerful, popular programming languages. You'll be a Python programmer.

Watch Zed, too! The accompanying DVD contains 5+ hours of passionate, powerful teaching: a complete Python video course!



For more information and sample content visit informit.com/hardway.



eBook and print formats available

PEARSON

informIT is a brand of Pearson and the online presence for the world's leading technology publishers. It's your source for reliable and qualified content and knowledge, providing access to the leading brands, authors, and contributors from the tech community.

▼ Addison-Wesley Cisco Press IBM Press Microsoft Press

PEARSON
IT CERTIFICATION

PRENTICE HALL

QUE

SAMS

VMWARE PRESS

LearnIT at InformIT

Looking for a book, eBook, or training video on a new technology? Seeking timely and relevant information and tutorials. Looking for expert opinions, advice, and tips? **InformIT has a solution.**

- Learn about new releases and special promotions by subscribing to a wide variety of monthly newsletters. Visit informit.com/newsletters.
- FREE Podcasts from experts at informit.com/podcasts.
- Read the latest author articles and sample chapters at informit.com/articles.
- Access thousands of books and videos in the Safari Books Online digital library. safari.informit.com.
- Get Advice and tips from expert blogs at informit.com/blogs.

Visit informit.com to find out all the ways you can access the hottest technology content.

Are you part of the **IT** crowd?

Connect with Pearson authors and editors via RSS feeds, Facebook, Twitter, YouTube and more! Visit informit.com/socialconnect.





Addison
Wesley

REGISTER



THIS PRODUCT

informit.com/register

Register the Addison-Wesley, Exam Cram, Prentice Hall, Que, and Sams products you own to unlock great benefits.

To begin the registration process, simply go to informit.com/register to sign in or create an account.

You will then be prompted to enter the 10- or 13-digit ISBN that appears on the back cover of your product.

Registering your products can unlock the following benefits:

- Access to supplemental content, including bonus chapters, source code, or project files.
- A coupon to be used on your next purchase.

Registration benefits vary by product. Benefits will be listed on your Account page under Registered Products.

About InformIT — THE TRUSTED TECHNOLOGY LEARNING SOURCE

INFORMIT IS HOME TO THE LEADING TECHNOLOGY PUBLISHING IMPRINTS Addison-Wesley Professional, Cisco Press, Exam Cram, IBM Press, Prentice Hall Professional, Que, and Sams. Here you will gain access to quality and trusted content and resources from the authors, creators, innovators, and leaders of technology. Whether you're looking for a book on a new technology, a helpful article, timely newsletters, or access to the Safari Books Online digital library, InformIT has a solution for you.

informIT.com

THE TRUSTED TECHNOLOGY LEARNING SOURCE

Addison-Wesley | Cisco Press | Exam Cram
IBM Press | Que | Prentice Hall | Sams

SAFARI BOOKS ONLINE

Where Are the Companion Content Files?

Register this digital version of *Learn Ruby the Hard Way, Third Edition*, to access important downloads.



Register this digital version to unlock the companion files that are included on the disc that accompanies the print edition. Follow the steps below.

1. Go to <http://www.informit.com/register> and log in or create a new account.
2. Enter this ISBN: **9780321884992**
NOTE: This is the ISBN of the print book which must be used to register the eBook edition.
3. Answer the challenge question as proof of purchase.
4. Click on the “Access Bonus Content” link in the “Registered Products” section of your account page, which will take you to the page where your downloadable content is available.

The Professional and Personal Technology Brands of Pearson



Cisco Press



informIT

PEARSON IT Certification



QUE

SAMS

VMWARE PRESS



JOIN THE **INFORMIT** AFFILIATE TEAM!

You love our titles and you love to share them with your colleagues and friends...why not earn some \$\$ doing it!

If you have a website, blog, or even a Facebook page, you can start earning money by putting InformIT links on your page.

Whenever a visitor clicks on these links and makes a purchase on informit.com, you earn commissions* on all sales!

Every sale you bring to our site will earn you a commission. All you have to do is post the links to the titles you want, as many as you want, and we'll take care of the rest.

APPLY AND GET STARTED!

It's quick and easy to apply.

To learn more go to:

<http://www.informit.com/affiliates/>

*Valid for all books, eBooks and video sales at www.informit.com



SAMS

informIT