The Pragmatic Programmers

Updated for Ruby 3

Learn to Program

Third Edition

Chris Pine

edited by Tammy Coron

The Facets of Ruby Series

Early Praise for *Learn to Program, Third Edition*

I would absolutely recommend this book. It's the book I wish I'd had when I started.

➤ **Erin Dees**
Principal Engineer and Author

Chris Pine has an uncanny ability to empathize with beginning programmers. His understanding of software, curriculum design and the science of attention continues to make *Learn to Program, Third Edition* the best resource I've ever found for those that are just getting started on this journey. I'm presently building a team of developer educators and I intend to buy a copy for every one of them as an example of how to design software curriculum.

➤ **Jonan Scheffler**
Director of Developer Relations and Hacker/Maker

This is an excellent book for someone who has never tried programming before, or someone who has tried but found it frustrating or confusing.

➤ **Justin Foote**
Software Engineer and Architect

I really love this book because Chris has so much compassion for his readers. He really brings you on a wonderful and enjoyable journey of code.

➤ **Rey Abolofia**
Senior Software Engineer and Educator

We've left this page blank to make the page numbers the same in the electronic and paper books.

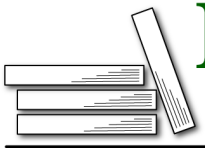
We tried just leaving it out, but then people wrote us to ask about the missing pages.

Anyway, Eddy the Gerbil wanted to say “hello.”

Learn to Program, Third Edition

Chris Pine

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Tammy Coron
Copy Editor: Corina Lebegioara
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-817-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—June 2021

Contents

Acknowledgments

The book you are reading is better than the book I originally wrote.

First and foremost, I want to thank my dear wife, Katy. Through all the incarnations of this book over the last 18 years, you were there supporting me from the very beginning. You were the original guinea pig for this experiment. This book wouldn't exist without you.

I also want to thank Tammy Coron, my amazing editor. I'm so glad no one has to read the book I originally wrote! You make me appear to be a better author than I actually am.

Thanks to my technical reviewers: Erin Dees, Jason Clark, Jonan Scheffler, Justin Foote, Rachael Berecka, Rachel Klein, and Rey Abolofia! I really look like I know what I'm talking about because of you.

A special thanks to my dear daughter, Ruby, for her artwork appearing at the end of the third chapter. It's a magical thing to see your child develop skills that far surpass your own.

Thanks to all of my readers and technical reviewers of earlier editions. Your feedback was invaluable in making this book what it is.

It really does take a village.

Preface to the Third Edition

I just realized that I've been working on this book, in one form or another, for 18 years. When I started, it was before smartphones, before Twitter and YouTube, and before Gmail and Google Maps. You couldn't visit MySpace in Firefox, because neither one existed yet. RSS feeds were the hot new thing.

The world has changed so much since then, and the changes keep happening faster and faster. This third edition was written under the shadow of COVID-19. As unemployment in the United States reaches heights not seen since the Great Depression, I still see lots of open programmer jobs out there. And programming is a job you can easily do from home.

Programming was a fun and interesting thing to learn in 2002. At the time not nearly as many programming jobs existed, but there were enough. And anyway, many of us did it for fun, for the joy of creating new things.

Today, surrounded by computers, programming increasingly feels like a critical skill to learn. It's still as fun and rewarding as it ever was. But as more industries become digital, and more companies become software companies with "an app for that," the importance of programming is greater than it has ever been.

I'm excited to embark on this journey with you.

What's New in This Edition

For the third edition of this book, I've updated all the examples for Ruby 3, the latest version of the language. Since the second edition came out around when Ruby 2 did, it seemed like the right thing to do and the right time to do it.

It also gave me a chance to update the examples for the world we live in. Some of the jokes and references stopped being funny/cute around 2012 or so.

Best of all, though, I added a new chapter! It's about APIs and how to use them. I figured that since it's so *easy* now to write programs that talk across

the Internet to programs on the other side of the world, I just had to show you how it's done.

AND... I fixed a typo that no one had found in 18 years, which I kind of can't believe, but there it is.

Chris Pine

Portland, Oregon, USA, December 2020

Introduction

I vividly remember writing my first program. (My memory is pretty horrible; I don't vividly remember many things, only things like waking up after oral surgery, watching the birth of my children, or writing my first program…you know, things like that.)

I suppose, looking back, that it was a fairly ambitious program for a newbie, maybe twenty or thirty lines of code. But I was a math major, after all, and we're supposed to be good at things like "logical thinking." So, I went down to the Reed College computer lab, armed only with a book on programming and my ego, sat down at one of the Unix terminals there, and started programming. Well, maybe "started" isn't the right word. Or "programming." I mostly just sat there, feeling hopelessly stupid. Then ashamed. Then angry. Then small. Eight grueling hours later, the program was finished. It worked, but I didn't much care at that point. It wasn't a triumphant moment for me.

It has been more than two decades, but I can still feel the stress and humiliation in my stomach when I think about it.

Clearly, this was *not* the way to learn programming.

Why was it so hard? I mean, there I was, this reasonably bright guy with some fairly rigorous mathematical training—you'd think I'd be able to get this! And I did go on to make a living programming and even to write a book about it, so it's not like I "didn't have what it took" or anything like that (which is kind of how I felt). No, in fact, I find programming to be pretty easy these days, for the most part.

So, why was it so hard to tell a computer to do something only mildly complex? Well, it wasn't the "mildly complex" part that was giving me problems; it was the "tell a computer" part.

In any communication with humans, you can leave out all sorts of steps or concepts and let them fill in the gaps. In fact, you have to do this, otherwise we'd never be able to get anything done. A favorite example is making a peanut

butter and jelly sandwich. Normally, if you wanted someone to make you a peanut butter and jelly sandwich, you might simply say, "Hey, could you make me a peanut butter and jelly sandwich?" But if you were talking to someone who had never made a sandwich before, you'd have to tell them how to do it:

1. Get out two slices of bread (and put the rest back).
2. Get out the peanut butter, the jelly, and a butter knife.
3. Spread the peanut butter on one slice of bread and the jelly on the other one.
4. Put the peanut butter and jelly away and take care of the knife.
5. Put the slices together, put the sandwich on a plate, and bring it to me. Thanks!

I imagine those would be sufficient instructions for a small child. Small children are needlessly, recklessly clever, though. What would you have to say to a computer? Well, let's look at that first step. To get out the two slices of bread, the computer would have to do the following:

1. a. Locate bread.
   b. Pick up bread.
   c. Move to empty counter.
   d. Set down bread on counter.
   e. Open bag of bread.
   f. (And so on…)

But no, this isn't nearly good enough. For starters, how does it "locate bread"? You'll have to set up some sort of database associating items with locations. The database will also need locations for peanut butter, jelly, knife, sink, plate, counter….

Oh, and what if the bread is in a bread box? You'll need to open it first. Or in a cabinet? Or in your fridge? Perhaps behind something else? Or what if it's *already on the counter?* You didn't think of that one, did you? So, now we have this:

- Initialize item-to-location database.
- If bread is in bread box:
  - Open bread box.
  - Pick up bread.
  - Remove hands from bread box. (Important!)
  - Close bread box.

- If bread is in cabinet:
  - Open cabinet door.
  - Pick up bread.
  - Remove hands from cabinet.
  - Close cabinet door.
- (And so on…)

And on and on it goes. What if no clean knife is available? What if no empty counter space is available? And you'd better hope—with everything you've got—that there's no twist-tie on that bread!

Even steps such as "open bread box" need to be explained…and this is why we don't have robots making sandwiches for us (yet). It's not that we can't build the robots; it's that we can't program them to make sandwiches. It's because making sandwiches is *hard* to describe (but easy to do for smart creatures like us humans), and computers are good only for things that are (relatively) *easy* to describe (but hard to do for slow creatures like us humans).

And that's why I had such a hard time writing that first program. Computers are way dumber than I was prepared for.

When you teach someone how to make a sandwich, your job is made much easier because they already know what a sandwich is. It's this common, informal understanding of "sandwichness" that allows them to fill in the gaps in your explanation. Step 3 tells you to spread the peanut butter on one slice of bread. It doesn't say to spread it on only one side of the bread or to use the knife to do the spreading (as opposed to, say, your forehead). You assume they simply know these things.

To make this clearer, I think it'll help to talk a bit about what programming is so that you have a sort of informal understanding of it.

What Is Programming?

Programming is telling your computer how to do something. Large tasks must be broken down into smaller tasks, which must be broken down into still smaller tasks, continuing down to the simplest tasks that you don't have to describe—the tasks your computer already knows how to do. (These are *really* basic things such as arithmetic or displaying some text on your screen.)

My biggest problem when I was learning to program was that I was trying to learn it backwards. I knew what I wanted the computer to do and tried working backward from that, breaking it down until I got to something the computer knew how to do. Bad idea. I didn't know what the computer *could*

do, so I didn't know what to break the problem down to. (Mind you, now that I do know, this is exactly how I program. But it doesn't work to start that way.)

That's why you're going to learn it differently. You'll first learn about some of the things your computer can do and then break some simple tasks down into steps your computer can handle. *Your* first program will be so easy, it won't even take you a minute.

One reason your first program will be so easy is that you'll be writing it in one of the more elegant programming languages, Ruby. It's a good choice for a first programming language.

Programming Languages

To tell your computer how to do something, you must use a programming language. A programming language is similar to a human language in that it's made up of basic elements (such as nouns and verbs) and ways to combine these elements to create meaning (sentences, paragraphs, and novels).

There are so many programming languages out there, each with their own strengths and weaknesses: Java, C++, Ruby, Lisp, Go, Erlang, and a few hundred more. As someone learning to program, how are you supposed to know which one to learn first? You let someone else pick for you. And in this case, Ruby is the language you'll be using.

Ruby is one of the easiest programming languages to learn. In my opinion, it's every bit as easy as some of the "beginner" languages out there. And since programming is hard enough as it is, let's make this as easy as we can.

Even though Ruby is as easy to learn as a language specifically geared toward beginners, it's a full, professional-strength programming language. By the end of this book, you won't need to go out and learn a "real" language. People have jobs writing Ruby code. Quite a few readers of this book's first and second editions have reached out to let me know that they're supporting their families writing Ruby code. Many websites are powered by Ruby. It's the real deal.

Ruby is also a fun programming language. In fact, the reason that Ruby's creator, Yukihiro Matsumoto, made Ruby in the first place was to create a programming language that would make programmers happy. This spirit of the joy of programming pervades the Ruby community. I know maybe a dozen programming languages at this point, and Ruby is still the first one I turn to.

But perhaps the best reason for using Ruby is that Ruby programs tend to be *short*. For example, here's a small program in Java:

```
public class HelloWorld {
  public static void main(String []args) {
    System.out.println("Hello world");
  }
}
```

And here's the same program in Ruby:

```
puts "Hello world"
```

This program, as you might guess from the Ruby version, writes `Hello world` to your screen. That's not nearly as obvious from looking at the Java version.

How about another comparison? Let's write a program to do *nothing*. Literally nothing at all. In Ruby, you don't need to *write* anything at all; a completely blank program is a valid Ruby program that does nothing. Which makes sense, right?

In Java, though, you need all of this:

```
public class DoNothing {
  public static void main(String[] args) {
  }
}
```

You need all of that code simply to do nothing beyond saying, "Hey, I am a Java program, and I don't do anything."

So, these are the reasons why you'll use Ruby in this book. (My first program was *not* in Ruby, which is another reason why it was so painful.) Now let's get you all set up for writing Ruby programs.

Installation and Setup

You'll be using three main tools when you program: a text editor (to write your programs), the Ruby interpreter (to run your programs), and your command line (which is how you tell your computer which programs you want to run).

Although there's pretty much only one Ruby interpreter and one command line, there are many text editors to choose from—and some are much better for programming than others. A good text editor can help catch many of those "silly mistakes" that beginning programmers make. Oh, all right, that *all* programmers make. It makes your code much easier for yourself and others to read in a number of ways: by helping with indentation and formatting, by letting you set markers in your code (so you can easily return to something you're working on), by helping you match up your parentheses, and most importantly by *syntax coloring* (coloring different parts of your code with

different colors according to their meanings in the program). You'll see syntax coloring in this book's examples.

With so many good editors (and so many bad ones), it can be hard to know which one to choose. I'll tell you which one I use in the setup appendix for your OS; that'll have to be good enough for now. ☺ But whatever you choose as your text editor, do *not* use something like Word, Pages, or Google Docs. Aside from being made for an entirely different purpose, they usually don't produce plain text (that is, text with no extra information about font, text size, or bold/italics/strikethrough), and your code must be in plain text for your programs to run.

Since setting up your environment differs somewhat from platform to platform (which text editors are available, how to install Ruby, how your command line works, and so on), I've placed the setup instructions in this book's appendices. Find the right one for your computer: Windows, macOS, or Linux.

Then come back here, because there's one last thing I want to talk to you about before we get started: the art of programming.

The Art of Programming

An important part of programming is, of course, making a program that does what it's supposed to do. In other words, it should have no bugs. You already know this. But focusing on correctness and bug-free programs misses a lot of what programming is all about. Programming isn't only about the end product; it's about the process that gets you there. (Anyway, an ugly process will result in buggy code. This happens every time.)

Programs aren't built in one go. They are talked about, sketched out, prototyped, played with, refactored, tuned, tested, tweaked, deleted, rewritten....

A program isn't built; it's grown.

Because a program is always growing and changing, it must be written with change in mind. I know it's not clear yet what this means in practical terms, but I'll be discussing it throughout the book.

Probably the first, most fundamental rule of good programming is to avoid duplication of code at all costs. This is sometimes called the DRY rule: Don't Repeat Yourself.

I usually think of it in another way: a good programmer cultivates the virtue of laziness. But not just any laziness. You must be aggressively, proactively lazy. Save yourself from doing unnecessary work whenever possible. If making

a few changes now means you'll be able to save yourself more work later, do it. Make your program a place where you can do the absolute minimum amount of work to get the job done. Not only is programming this way much more interesting (as it's boring to do the same thing over and over), but it produces less buggy code, and it also produces code faster. It's a win-win-win situation.

Either way you look at it (DRY or laziness), the idea is the same: make your programs flexible. When change comes (and it *always* does), you'll have a much easier time changing with it.

Well, that about wraps up the introduction to this book. Looking at other technical books I own, they always seem to have a section here about "Who should read this book," "How to read this book," or something like that. Well, I think *you* should read this book, and front-to-back always works for me. (I mean, I did put the chapters in this order for a reason, you know.) Anyway, I never read that stuff, so let's program!

CHAPTER 1

Numbers

Are you ready to start programming? Is Ruby installed and do you have a text editor picked out? If not, check out the appropriate appendix to get things set up on your computer.

Open your text editor, and type the following:

```
puts 1+2
```

Save your program (yep, that's a complete program) as calc.rb. Run it by typing ruby calc.rb into your command line. You'll see that it puts a 3 on your screen.

What, you didn't see it? Maybe you saw a window flash up and then disappear? Whatever the case may be, if you don't see a 3 on your screen, the problem could be that you didn't run the program from the command line.

Don't just click your program's icon.

Don't just press F5 in your text editor.

You need to run your program by typing ruby calc.rb into your command line.

So, what's going on here? How does your program work and what does it do? You're about to find out.

Printing to the Screen

Looking at the program, you can guess that the 1+2 handles the math part. In fact, you can simply write this:

```
puts 3
```

And you'll end up with the same result: a 3 on your screen.

You may have guessed by now that the puts method writes onto the screen whatever comes after it; for example, the result of 1+2.

Now let's talk about the two different kinds of numbers that we can use in our programs.

Integers and Floats

In most programming languages (and Ruby is no exception), numbers without a decimal point are called *integers*, and numbers with a decimal point are usually called *floating-point numbers* or, more simply, *floats*.

Here are some integers:

```
5
-205
99999999999999999999999999
0
```

And here are some floats:

```
54.321
0.001
-205.3884
0.0
```

In practice, most programs use integers more often than they use floats. After all, no one wants to look at 7.4 emails, browse 1.8 web pages, or listen to 5.24 of their favorite songs. Many programs even use integers for money, separately keeping track of the number of pennies.

Floats, on the other hand, are used more for academic purposes, like physics experiments and such, and for audio and video programs (including 3D applications).

Now let's look at how to use simple arithmetic to solve problems that we might use a calculator for.

Simple Arithmetic

We can write programs to solve arithmetic problems like a calculator would. For addition and subtraction, we use `+` and `-`, as we saw. For multiplication, we use `*`, and for division we use `/`.

Let's expand the calc.rb program a little. Try coding this program:

```
Line 1 puts 1.0 + 2.0
     2 puts 2.0 * 3.0
     3 puts 5.0 - 8.0
     4 puts 9.0 / 2.0
```

This is what the program returns:

```
3.0
6.0
-3.0
4.5
```

(The spaces in the program aren't important; they simply make the code easier to read.) Well, that result wasn't too surprising. Now let's try it with integers:

```
puts 1+2
puts 2*3
puts 5-8
puts 9/2
```

This is mostly the same, right?

```
3
6
-3
4
```

Um...except for that last one. When you do arithmetic with integers, you'll get integer answers. When Ruby can't get the "correct" answer, it always rounds down. (Of course, `4` *is* the correct answer in integer arithmetic for `9/2`. But it might not be the answer you were expecting.)

Perhaps you're wondering why you'd use integer division. What's it good for?

Well, let's say you're going to the movies but you have only $9. Back in Ye Olde Portland, Oregon, you could see a movie at the Bagdad theater for two bucks. Believe it or not, it was cheaper for two people to go to the Bagdad and get a pitcher of beer—*good* beer—than it is now to go see a movie at your typical theater. (And, at the Bagdad, the seats all had tables in front of them. For your beer! It was heavenly.) Anyway, nostalgia aside, how many movies could you see at the Bagdad for nine bucks (`9/2`)? Here's a hint: the answer isn't four and a half.

You see, in this scenario, 4.5 is definitely *not* the correct answer since the folks that operate the Bagdad theater won't let you watch half of a movie or let half of you see a whole movie...some things just aren't divisible.

Before you experiment with some programs of your own, I have a couple of tips for you. First, if you're typing a large number, like a million, don't use commas. If you type `1,000,000`, you'll confuse Ruby.

Second, if you want to write more complex expressions, you can use parentheses. Everything inside the parentheses will be computed before things outside them, so you can use parentheses to control the order of operations as in the following example:

```
Line 1 puts 5 * (12-8) + -15
     2 puts 98 + (59872 / (13*8)) * -51

  ❮ 5
    -29227
```

A Few Things to Try

Write a program that tells you the following:

- *Hours in a year.* How many hours are in a year?
- *Minutes in a decade.* How many minutes are in a decade?
- *Your age in seconds.* How many seconds old are you? I'm not going to check your answer, so be as accurate—or not—as you want.

Here's a tougher question:

- *Our dear author's age.* If I'm 1,390 million seconds old (which I am, though I was somewhere in the 800 millions when I started the first edition of this book), how old am I?

Progress Checkpoint

In this chapter you learned about numbers. (I really nailed the chapter title, didn't I?) More specifically, you learned about the two kinds of numbers that computers use most often: integers and floats. You also learned how to do basic arithmetic on those numbers, including the slightly weird case of integer division.

Your next stop is learning how to use text in your programs. Because it's pretty hard to write interesting programs without printing some sort of text.

CHAPTER 2

Letters

In the previous chapter, you learned about numbers. But what about letters? Words? Text?

In a program, groups of letters are known as *strings*. (You can picture little beads with letters on them, tied together with a string.) Here are a few strings to get you started:

```
"Hello!"
"99 Red Balloons"
"Snoopy says #%^?&*@! when he stubs his toe."
"     "
""
```

Notice that strings can have more than letters—they can include punctuation, digits, symbols, spaces, or nothing at all. The last string in that example is known as an *empty string* because it contains nothing.

Since you're already familiar with using `puts` to print numbers, let's use `puts` to print strings:

```
Line 1 puts "Hello, world!"
   2 puts ""
   3 puts "Good-bye."
```

```
❮ Hello, world!

  Good-bye.
```

Hopefully that's the output you were expecting. Now let's look at some more things you can do with strings.

String Arithmetic

Just as you can do arithmetic on numbers, you can also do arithmetic on strings. Well, sort of. You can add strings together.

Let's try to add two strings and see how puts handles that:

```
Line 1 puts "I like" + "apple pie."
```

```
❮ I likeapple pie.
```

Whoops! There's no space between "I like" and "apple pie.". Spaces don't usually matter much in your code, but inside a string they matter a lot. (You know what they say: computers don't do what you *want* them to do, only what you *tell* them to do.)

Okay, let's try this again:

```
Line 1 puts "I like " + "apple pie."
   2 puts "I like" + " apple pie."
```

```
❮ I like apple pie.
  I like apple pie.
```

Ah, much better. For the record, you can add the space to either string.

Adding strings is cool, but you can do more than that: you can also multiply them. Watch this:

```
Line 1 puts "blink " * 4
```

And you get this:

```
❮ batting my eyes
```

Just kidding. Not even Ruby is that clever. What you actually get is this:

```
❮ blink blink blink blink
```

If you think about it, this result makes perfect sense. After all, 7*3 actually means 7+7+7, so "moo"*3 would mean "moo"+"moo"+"moo".

Before we go any further, let's make sure you understand the difference between *numbers* and *digits*.

Numbers vs. Digits

12 is a number, but "12" is a string of two digits. To smart human brains, these are similar concepts and we hardly ever talk about them. But to Ruby, these are very different things.

Let's play around with this for a bit in code to get a feel for how numbers behave differently from strings of digits. Try this:

```
Line 1 puts  12  +  12
   2 puts "12" + "12"
   3 puts "12  +  12"
```

```
❮ 24
  1212
  12  +  12
```

And now this:

```
Line 1 puts  2  *  5
   2 puts "2" *  5
   3 puts "2  *  5"
```

```
❮ 10
  22222
  2  *  5
```

These examples are pretty clear. But you need to be careful with how you mix your strings and numbers, or you might run into problems.

When Bad Things Happen to Good Programs

You've seen what happens when you mix strings with numbers and things go right. Now let's see what happens when things go wrong. Try this:

```
Line 1 puts "12" + 12
   2 puts "2" * "5"
```

```
❮ example.rb:1:in `+': no implicit conversion of Integer into String (TypeError)
        from example.rb:1:in `<main>'
```

Well, that's not good. You get an error. The problem here is that you can't add a number to a string or multiply a string by another string. So, for example, you can't do something like this:

```
Line 1 puts "Marceline" + 12
   2 puts "Finn" * "Jake"
```

When you think about it, adding "Marceline" and 12 makes no sense. Neither does multiplying "Finn" and "Jake".

Here's something else to be aware of: you can write "Apollo"*11 in a program, since it means eleven sets of the string "Apollo" all added together. But you *can't* write 11*"Apollo", since that means "Apollo" sets of the number 11, which is certainly creative but will never work.

Here's another thing that can go wrong: what if you want a program to print out They said, "Yes!" when it runs? You could try this:

```
Line 1 puts "They said, "Yes!""
```

The trouble is, *that* won't work; you can tell that from the syntax coloring. The problem with this code is that Ruby can't tell the difference between a double quote character that's supposed to be inside the string and a double quote character intended to end the string. The confusion is reasonable here. They are, after all, the same character. To get around this problem, you need a way to tell Ruby "I want a double quote here, inside this string." How do you let Ruby know you want to stay in the string? You have to *escape* the double quote, like this:

```
Line 1 puts "They said, \"Yes!\""

❮ They said, "Yes!"
```

As you can see, the backslash is the escape character. When you have a backslash in front of another character, it almost always means "just put this next character in the string." (A few special *escape sequences*, like "\n", are converted into a different character. But we'll get back to that later.)

Let's see a few examples of escaping in strings:

```
Line 1 puts "They said, \"Yes!\""
   2 puts "up\\down"
   3 puts "backslash at the end of a string: \\"
   4 puts "Dip!"
   5 puts "\D\i\p\!"

❮ They said, "Yes!"
up\down
backslash at the end of a string: \
Dip!
Dip!
```

It's important to note that those last two strings are identical. Obviously, they don't look the same in the code, but when your program is running, those are two different ways of describing identical strings. In both cases, the string has exactly four characters. When you write "Dip!", you use six characters to describe this four-character string (I'm counting the quotes), and when you write "\D\i\p\!", you use ten characters to describe the same four-character string.

As for the special escape sequences, you'll almost never use them at first, aside from "\n", which generates a newline character, and maybe "\t", which generates a tab character:

```
Line 1  puts "Duck...\nDuck..."
    2  puts "\tGoose! (or grey duck for some of you)"

    ❮  Duck...
       Duck...
               Goose! (or grey duck for some of you)
```

That's all you need for the examples in this book.

Progress Checkpoint

In the last chapter we looked at numbers and arithmetic, and in this chapter you learned about strings and how to add and multiply them. We also covered the difference between numbers and digits, and how to use the backslash to create strings with quotes and backslashes in them. Next, you'll learn how to store these strings and numbers in variables (and why you'd want to).

CHAPTER 3

Variables and Assignment

So far, we've been writing some fairly simple programs with numbers and strings. As our programs get more and more complex, it's going to be a lot easier if we can assign names to those numbers and strings. We call these names *variables*. In this chapter, we'll learn how to use variables to make our programs easier to write and read.

Let's say you wanted to print a string twice. With what you've learned so far, the only way to do it would be like this:

```
Line 1 puts "...you can say that again..."
     2 puts "...you can say that again..."

❮    ...you can say that again...
     ...you can say that again...
```

That works, but I don't want to type that string twice. I'm *lazy.* Plus, I might type it wrong the second time. Sure, we could copy and paste the string, but that's not maximally lazy. What if we want to change the string later? We'd have to change it *twice.* Copy and paste is the opposite of the Don't Repeat Yourself rule.

To store the string in your computer's memory for use later in your program, you need to give the string a name. Programmers often refer to this process as *assignment*, and we call the names variables. A variable name can usually be almost any sequence of letters and numbers, but in Ruby, the first character of this name needs to be a lowercase letter. Let's try that last program again, but this time, let's give the string the name `my_string` (though we could have also named it `str` or `my_own_little_string` or `henry_the_8th`):

```
Line 1 my_string = "...you can say that again..."
   2 puts my_string
   3 puts my_string
```

```
...you can say that again...
...you can say that again...
```

So, is this program prettier than the first example? Yes. This is longer but prettier. We'll make it prettier still, and even shorter than the original, on page 51.

In the previous example, whenever you tried to do something to my_string, the program did it to "...you can say that again..." instead. You can think of the variable my_string as "pointing to" the string "...you can say that again...". Here's a slightly more motivated example:

```
Line 1 name = "Anya Christina Emmanuella Jenkins Harris"
   2 puts "My name is " + name + "."
   3 puts "Wow! " + name + "\nis a really long name."
```

```
My name is Anya Christina Emmanuella Jenkins Harris.
Wow! Anya Christina Emmanuella Jenkins Harris
is a really long name.
```

This is cool because you don't want to type that name out twice if you can avoid it. But what I don't love about this example is adding a bunch of strings together. If you only have two strings, adding them together is fine. But if you have more than that, you end up with all these quotes, plusses, and spaces all over the place, and it's harder to read.

The more common thing is to use *string interpolation*, like this:

```
Line 1 name = "Anya Christina Emmanuella Jenkins Harris"
   2 puts "My name is #{name}."
   3 puts "Wow! #{name}\nis a really long name."
```

```
My name is Anya Christina Emmanuella Jenkins Harris.
Wow! Anya Christina Emmanuella Jenkins Harris
is a really long name.
```

Ruby takes whatever you put in the "#{...}", and inserts it right there in the string. Much better.

Here's more cool stuff about variables: just as you can *assign* an object to a variable, you can *reassign* a different object to that variable. (This is why they're called *variables*—what they point to can vary.) Let's try it:

```
composer = "Mozart"
puts "#{composer} was all the rage in his day."

composer = "Beethoven"
puts "But I prefer #{composer}, personally."
```

```
Mozart was all the rage in his day.
But I prefer Beethoven, personally.
```

Of course, variables can point to any kind of object, not only strings:

```
my_own_var = "just another " + "string"
puts my_own_var

my_own_var = 5 * (1+2)
puts my_own_var
```

```
just another string
15
```

Now you know how to create variables and assign values to them. The idea is simple enough, but it can sometimes seem complicated when you have multiple variables. Let's look at this more closely to get a clearer understanding of how it works.

Variables Point to Values

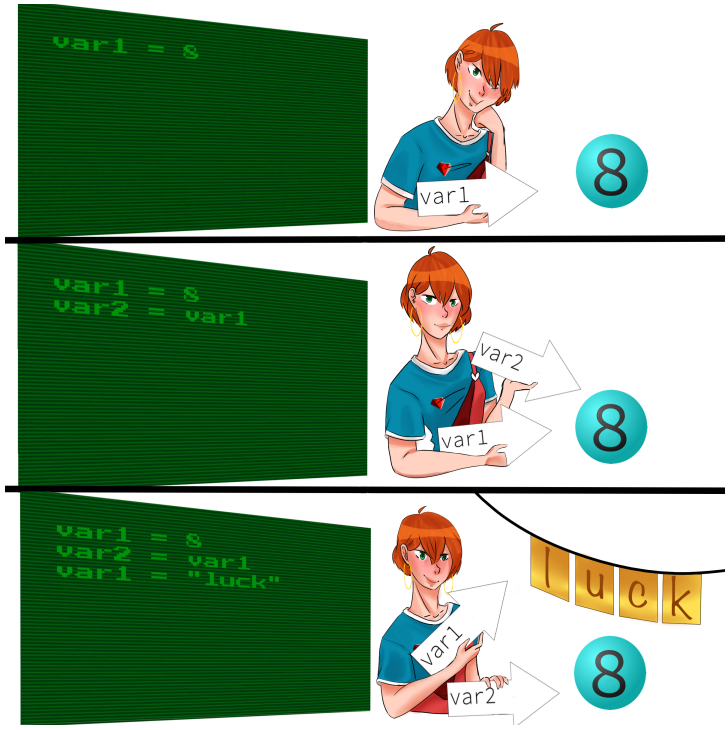
As you've seen, a variable can point to any kind of object. What happens if you try to point a variable to another variable? Let's see:

```
var1 = 8
var2 = var1
puts var1
puts var2

puts ""

var1 = "luck"
puts var1
puts var2
```

```
8
8

luck
8
```

Notice that on line 2, `var2` points to `var1`, which is pointing to `8`. Then on line 8, we reassign `var1` so that it points to `"luck"`. So why is `puts` printing `8` for `var2` instead of `"luck"`? Good question.

The issue here is that var2 was never *really* pointing to var1. Variables can't point to other variables, so var2 was actually pointing to 8 the whole time.



Progress Checkpoint

Now you know how to assign names to the numbers and strings in your programs. This enables you to write cleaner, lazier code. It'll also make it a lot easier to keep things straight as you move into the next chapter, where you'll learn how to work with both strings and numbers together.

CHAPTER 4

Conversions and Input

So far, you've seen objects like integers, floats, and strings, and you've set up variables to point to these objects. Now it's time for you to get them all to play nicely together.

Let's suppose you have a program that you want to print 25. Using the following code won't work because you can't add numbers and strings together:

```
Line 1 var1 =  2
     2 var2 = "5"
     3 puts var1 + var2
```

Part of the problem is that Ruby doesn't know if you were trying to print 7 (2 + 5) or 25 ("2" + "5"), so it's your job to specify which. But before you can do that, there's something else you need to do first.

To add the values together in some meaningful way, you first need to convert the objects to be the same type. In other words, you need to get the string version of the integer—in this case var1—or the integer version of the string—in this case var2.

Let's look at how you do that.

Numbers to Strings and Back Again

To get the string version of an object, you simply write .to_s after it like this:

```
Line 1 var1 =  2
     2 var2 = "5"
     3 puts var1.to_s + var2

  ❮ 25
```

Similarly, appending .to_i gives the integer version of an object, and .to_f gives the float version. Let's look at what these methods do (and *don't* do) a little more closely:

```
Line 1  var1 =  2
    2  var2 = "5"
    3  puts var1.to_s + var2
    4  puts var1 + var2.to_i
```

```
❮ 25
  7
```

Notice that, even after you got the string version of var1 by calling to_s, var1 was always pointing at 2 and never at "2". Unless you explicitly reassign var1 (which requires an = sign), it'll continue to point at 2 for the life of the program.

Let's try some more interesting (and a few just plain weird) conversions:

```
Line 1  puts "15".to_f
    2  puts "99.999".to_f
    3  puts "99.999".to_i
    4  puts ""
    5  puts "5 is my favorite number!".to_i
    6  puts "Who asked you about 5 or whatever?".to_i
    7  puts "Your mama did.".to_f
    8  puts ""
    9  puts "stringy".to_s
   10  puts 3.to_i
```

```
❮ 15.0
  99.999
  99

  5
  0
  0.0

  stringy
  3
```

So, this program probably gave you some surprises. The conversion on line 1 is pretty standard, giving 15.0. After that, you converted the string "99.999" to a float and to an integer. The float did what was expected; the integer was, as always, rounded down.

Next, you had examples of some *unusual* strings being converted into numbers. On line 5, to_i ignores the first thing it doesn't understand (and the rest of the string from that point on). So the string on line 5 was converted to 5; but on lines 6 and 7, since those strings started with letters, they were ignored completely. When this happens, Ruby picks zero.

Finally, you saw that the last two conversions did nothing at all, as you'd expect.

Let Me Tell You a Secret

There's something strange about the puts method. Have a look at this:

```
Line 1  puts  20
    2  puts  20.to_s
    3  puts "20"

 ❮  20
    20
    20
```

Why do these three all print the same thing? Well, the last two should, since 20.to_s is "20". But what about the first example, the integer 20? For that matter, what does it even mean to write the *integer* 20? When you write a *2* and then a *0* on a piece of paper, you're writing a string, not an integer. The integer 20 is the number of fingers and toes I have; it isn't a *2* followed by a *0*.

Well, here's the big secret behind the puts method: before puts tries to write out an object, it uses to_s to get the string version of that object. In fact, the *s* in puts stands for *string*; puts actually means *put string*.

And here's a *bonus secret*: string interpolation does the same thing. Check it out:

```
Line 1  puts "my favorite number really is #{2+3}"

 ❮  my favorite number really is 5
```

This actually ends up being convenient, as it's one less conversion that you have to think about.

Now let's check out how you can get strings directly from the user, which will allow you to write all sorts of fun programs.

Getting Strings from the User

To get strings from the user, you use the gets method. If puts means *put string*, you can probably guess what gets means. And just as puts always spits out strings, gets retrieves only strings. So, where do these strings come from?

From you—well, from your keyboard, anyway! Of course, since your keyboard makes only strings, that works out beautifully. What actually happens is that gets simply sits there, reading what you type until you press Enter. Try it out:

```
Line 1 puts gets

⇛ Is there an echo in here?
❮ Is there an echo in here?
```

As expected, whatever you type will get repeated back to you. Run it a few times and try typing different things.

Did It Work?

Maybe you didn't need any help installing Ruby, so you skipped Chapter 1. Hey, no problem.

Maybe you've done a little programming before, so you skipped Chapter 2. That's totally fine.

The only thing is that you missed some stuff there that you need to know now. If you haven't been running your programs from the command line, then you'll almost certainly have problems with gets, and you're going to be using it a lot from now on. So, if you saved your program as example.rb, you should run your program by typing ruby example.rb into your command line. If you're having trouble getting around on your command line, refer to the OS-specific appendix at the back of this book.

Now that you know how to use gets, you can make interactive programs. These will be *way more interesting* than the programs you've written so far. Still, one issue remains with gets that'll bite you, so let's look at what it is and how to avoid it.

Cleaning Up User Input

Let's make a program that greets you by name. Write the following program:

```
Line 1 puts "Hello there, and what's your name?"
  2 name = gets
  3 puts "Your name is #{name}? What a lovely name!"
  4 puts "Pleased to meet you, #{name}. :)"
```

I ran it, and this is what happened:

```
❮ Hello there, and what's your name?
⇛ Chris
❮ Your name is Chris
  ? What a lovely name!
  Pleased to meet you, Chris
  . :)
```

Yikes! This program first prompts for a name, and then it prints a message. But something's wrong. When I typed the letters C, h, r, i, and s, and then pressed Enter, the gets method got all of the letters in my name *and* the Enter. Well, that's not good. Fortunately, there's a method that deals with exactly this sort of thing: chomp. The chomp method takes off any newline characters hanging out at the end of your string. Let's try that program again, but this time using chomp to help tidy up the response:

```
Line 1 puts "Hello there, and what's your name?"
  2 name = gets.chomp
  3 puts "Your name is #{name}? What a lovely name!"
  4 puts "Pleased to meet you, #{name}. :)"

❮ Hello there, and what's your name?
⇛ Chris
❮ Your name is Chris? What a lovely name!
  Pleased to meet you, Chris. :)
```

Much better. Notice that since name is pointing to gets.chomp, you don't ever have to say name.chomp; name was already chomped. (Of course, if you did chomp it again, it wouldn't do anything; it has no more Enter characters to chomp off. You could chomp on that string all day, and it wouldn't change it. Like week-old bubble gum.)

Now you're ready to write your own truly interactive programs! Try it out with these exercises. (Or ignore these and try it out with your own ideas. That works, too.)

A Few Things to Try

- *Full name greeting.* Write a program that asks for a person's first name, then middle name, and then last name. Finally, have the program greet the person using their full name.
- *Bigger, better favorite number.* Write a program that asks for a person's favorite number. Have your program add 1 to the number, and then suggest the result as a bigger and better favorite number. (Please be tactful about it, though.)

Good Variable Names

When writing a program, always try to have a good feel for what each variable is pointing to: a number, a string, or whatever. As in the favorite number program, at some point you'll have the person's favorite number as a string, and at another point you'll have it as an integer. It's important to keep track of which is which, and you can do this by keeping them in different variables.

Name the variables so that it's easy to tell what they are at a glance. For example, if you had a variable for someone's name, you might call it `name` and assume it was a string. If you had someone's age in a variable, you might call it `age` and assume it was an integer. So, if you needed to have a string version of someone's age, try to make that obvious by calling it something like `age_string` or `age_as_string`.

Most of all, try not to use *bad* variable names: no vague, meaningless names like `stuff` and `things`. In nearly all cases, avoid variables with a single letter. (The few exceptions are in abstract, math-heavy code, or in graphics programming, where `x` and `y` have concrete meanings.) And while you might often see example code in tutorials using `foo` and `bar`, never use these in your code. These are almost like meta-variables, meaning "replace these in your code with variables that actually make sense."

I'm not sure you know, but this book started out as an online tutorial. (It was much shorter back then.) I've received hundreds of emails from people getting stuck. In most of those cases, the problem was conversion, and it usually was someone trying to add an integer and a string together. Let's look at that error a bit more closely:

```
Line 1  my_birth_month = "August"
    2  my_birth_day   = 3
    3
    4  puts my_birth_month + my_birth_day

❮   example.rb:4:in `+': no implicit conversion of Integer into String (TypeError)
            from example.rb:4:in `<main>'
```

Here, Ruby is saying that it can't convert an integer into a string. Well, you know it *can* convert an integer into a string, but it doesn't want to without your explicit instructions. (It's only a computer, after all, and computers aren't exactly known for their independent thinking and initiative.) Honestly, it's probably a good thing, because maybe you don't want to convert the integer into a string, you know? Maybe you want to convert the string into an integer. Essentially, you're back to the whole "2 plus 5 adding up to 7 or 25" problem that was discussed on page 15.

It's easy to get frustrated when your program has errors. But instead of thinking about these things in terms of errors, think of Ruby as a small child asking for help. If only Ruby were a bit older and could ask for things more eloquently, your computer might say something more like, "Excuse me, but I'm unclear as to one small point…did you want me to convert the integer to a string here, or vice versa? Although it's probably obvious to any human

what you're trying to do, I'm just a computer." Maybe someday computers will be able to do that, but in the meantime, try to be patient with it.

Progress Checkpoint

In this chapter you learned how to convert between numbers and strings. This was important because, as you saw, puts and gets only operate on strings. This means that you now know how to write truly interactive programs. And finally, we talked about the importance of good names in programming, because this stuff can quickly get confusing.

In the next chapter, you'll learn more about how methods actually work, review the methods you already know, and also learn several new ones. This will enable you to write more interesting and powerful programs.

CHAPTER 5

Methods

So far, you've seen a number of different methods—puts, gets, and so on. (Pop quiz: List all of the methods you've seen so far. There are ten of them. The answer is coming in a moment.) But we haven't actually talked about what methods are and how they work. In this chapter we'll do exactly that. You'll also learn some new methods, so you can do more in your programs.

The "technical" definition of *methods* is that they are *things that do stuff.* If objects—such as strings, integers, and floats—are the nouns in the Ruby language, then methods are the verbs; and you can't have a verb without a noun to *do* the verb. For example, ticking isn't something that happens on its own; a clock has to do it. In the English language, you'd say, "The clock ticks." In Ruby, you'd say clock.tick (assuming that clock was a Ruby object, of course, and one that could tick). Programmers might say that they were "calling clock's tick method" or that they "called tick on clock."

So, did you take the pop quiz? Good. Well, I'm sure you remembered the methods puts, gets, and chomp, since we recently covered those. You probably also got the conversion methods, to_i, to_f, and to_s. But did you get the other four? Yeah? No? Why, it's none other than our old arithmetic buddies: +, -, *, and /. (Arithmetic buddies? Seriously, Chris? I don't know why my editor even allows stuff like this.)

As I was saying, just as every verb needs a noun, every method needs an object. It's usually easy to tell which object is performing the method. The object is what comes right before the dot, like in the clock.tick example or in 101.to_s. Sometimes, it's not as obvious, as with the arithmetic methods. As it turns out, 5 + 5 is only a shortcut for writing 5.+(5). Here's an example (using different numbers):

```
Line 1 puts("hello ".+("world"))
   2 puts((10.*(9)).+(9))
```

```
hello world
99
```

It isn't pretty, so please don't write it like that. But it's important to understand what's *really* happening.

This also gives you a deeper understanding of why `"Apollo"*11` works, but `11*"Apollo"` doesn't (as discussed in Chapter 2). `"Apollo"*11` is telling `"Apollo"` to do the multiplying, but `11*"Apollo"` is telling `11` to do the multiplying. `"Apollo"` knows how to make `11` copies of itself and add them all together; but `11` will have a much more difficult time of making `"Apollo"` copies of *itself* and adding them together.

That leaves the `puts` and `gets` methods to explain. Where are their objects? In English, you can sometimes leave out the noun; for example, if a villain yells "Die!" the implicit noun is whomever they're yelling at. In Ruby, if you say `puts "to be or not to be"`, the implicit object is whatever object you happen to be *in*. But hang on, you don't even know *how* to be in an object yet. To see what object you're in, you can use the special variable `self`. Check this out:

```
Line 1 puts self
```

```
main
```

See that? You may not have realized it, but in all of your programs you've been inside a special object that Ruby created for you. That object represents the entire program, and is otherwise known as *main*.

If you didn't entirely follow all of that, that's okay. The important thing to get from all of this is that every method is being done by some object, even if it doesn't have a dot in front of it. If you understand that, then you're all set.

Now that you've learned what methods are, it's time to expand your method vocabulary. We'll start with some string methods.

Fancy String Methods

Let's learn a few fun string methods. You don't have to memorize them all; you can look up this page again if you forget them. I want to show you a *small* part of what strings can do. In fact, I can't remember even half of the string methods myself—but that's fine, because the Internet has great references with all of the string methods listed and explained. (I'll show you where to find them in Chapter 15, Beyond This Book, on page 133.) Actually, I don't even *want* to know all the string methods. It's kind of like knowing every word

in the dictionary: I can speak English just fine without knowing every word in the dictionary.

The first string method is reverse, which returns a reversed version of the string:

```
Line 1  var1 = "stop"
   2  var2 = "deliver repaid desserts"
   3  var3 = "....TCELES B HSUP - A magic spell?"
   4
   5  puts var1.reverse
   6  puts var2.reverse
   7  puts var3.reverse
   8  puts var1
   9  puts var2
  10  puts var3
```

```
❮ pots
  stressed diaper reviled
  ?lleps cigam A - PUSH B SELECT....
  stop
  deliver repaid desserts
  ....TCELES B HSUP - A magic spell?
```

That third string is a spoiler from *Final Fantasy 1* (or, as we called it at the time, *Final Fantasy*, back when the word "final" used to mean something). There's a statute of limitations on spoilers, right?

Anyway, as you can see, reverse doesn't change the original string; it simply makes a new backward version of it. That's why var1 is still "stop" even after you called reverse on it.

Another string method is length, which tells you the number of characters (including spaces) in the string:

```
Line 1  puts "What is your full name?"
   2  name = gets.chomp
   3  puts "Did you know there are " + name.length + " characters"
   4  puts "in your name, " + name + "?"
```

```
❮ What is your full name?
⇛ Chris Pine
❮ example.rb:3:in `+': no implicit conversion of Integer into String (TypeError)
          from example.rb:3:in `<main>'
```

Uh-oh! See? There it is. It's an easy mistake to make. (You probably think that I made that mistake on purpose, since I'm obviously such a fabulous programmer, writing books and all. You were thinking that, right?) Anyway, if you didn't know to be on the lookout for this error, you can still figure that the problem must have happened sometime after the line name = gets.chomp,

since “Chris Pine” was entered correctly. See if you can figure out where the bug is.

Did you find the bug? The problem is with length: it gives you an integer, but you want a string. That’s easy enough; you can throw in a .to_s to convert the length to a string. But you should use string interpolation to begin with:

```
Line 1  puts "What is your full name?"
    2  name = gets.chomp
    3  puts "Did you know there are #{name.length} characters"
    4  puts "in your name, #{name}?"

    ❮  What is your full name?
   ⇛  Chris Pine
    ❮  Did you know there are 10 characters
       in your name, Chris Pine?
```

Note that 10 is the number of *characters* in “Chris Pine”, not the number of *letters*. I guess you could write a program that asks for your first and last names individually and then adds those lengths together—hey, why don’t you do that? Go ahead, I’ll wait.

Did you do it? Awesome!

Some string methods also change the case (uppercase and lowercase) of a string. upcase changes every lowercase letter to uppercase, and downcase changes every uppercase letter to lowercase. swapcase switches the case of every letter in the string, and finally, capitalize is like downcase, except it switches the first character to uppercase (if it’s a letter). Let’s check them out:

```
Line 1  letters = "aAbBcCdDeE"
    2  puts letters.upcase
    3  puts letters.downcase
    4  puts letters.swapcase
    5  puts letters.capitalize
    6  puts " a".capitalize
    7  puts letters

    ❮  AABBCCDDEE
       aabbccddee
       AaBbCcDdEe
       Aabbccddee
        a
       aAbBcCdDeE
```

As you can see from line 6, the capitalize method capitalizes only the first *character*, not the first *letter*. And as you’ve seen before, throughout all of these method calls, letters remains unchanged. I don’t mean to belabor the

point, but it's important to understand. Some methods *do* change the associated object, but you haven't seen any yet, and you won't for some time.

The last of the fancy string methods you'll learn are for visual formatting. The first, center, adds spaces to the beginning and end of the string to make it centered. But just as you have to tell the puts method what you want it to print, and you have to tell the + method what you want it to add, you have to tell the center method how wide you want your centered string to be.

So, if you wanted to center the lines of a poem, you could do it like this:

```
Line 1 line_width = 50
  2 puts(                "Old Mother Hubbard".center(line_width))
  3 puts(               "Sat in her cupboard".center(line_width))
  4 puts(        "Eating her curds and whey,".center(line_width))
  5 puts(          "When along came a spider".center(line_width))
  6 puts(           "Who sat down beside her".center(line_width))
  7 puts("And scared her poor shoe dog away.".center(line_width))
```

```
         Old Mother Hubbard
        Sat in her cupboard
    Eating her curds and whey,
     When along came a spider
     Who sat down beside her
And scared her poor shoe dog away.
```

This code might look a bit weird. I wanted to line up the .center(line_width) part, so I added those extra spaces before the strings. This is because I think it's prettier that way. Programmers often have strong feelings about code aesthetics, and they often disagree with each other about them. The more you program, the more you'll come into your own style.

Anyway, I don't think that's how the nursery rhyme goes, but I'm too lazy to look it up. Speaking of laziness, do you see how the width of the poem is stored in the variable line_width? This lets you go back later and make the poem wider by changing only the first line of the program, instead of having to change every line that does centering. With a long poem, this could save you a lot of time. That's the kind of laziness you want in your programs.

You may have noticed that the centering isn't as beautiful as Google Docs would have done it. If you truly want perfect centering (and maybe a nicer font), then you should use something like Google Docs. Ruby is a wonderful tool, but no tool is the right tool for *every* job.

The other two string-formatting methods you'll learn in this chapter are ljust and rjust, which stand for *left justify* and *right justify*. They are similar to center,

except that they pad the string with spaces on the right and left sides, respectively. Let's look at all three in action:

```
line_width = 40
str = "==> text <=="
puts(str.ljust(line_width))
puts(str.center(line_width))
puts(str.rjust(line_width))
puts(str.ljust(line_width/2) + str.rjust(line_width/2))
```

```
==> text <==
              ==> text <==
                            ==> text <==
==> text <==                ==> text <==
```

And there you have it! You've learned nine new string methods: reverse, length, upcase, downcase, swapcase, capitalize, center, ljust, and rjust. Before moving on to learn some more mathy methods, cement your knowledge by writing some programs that use these string methods.

A Few Things to Try

- *Angry boss.* Write an angry boss program that rudely asks what you want. Whatever you answer, the angry boss should yell it back to you and then fire you. For example, if you type in I want a raise, the angry boss should yell back:

```
WHADDAYA MEAN "I WANT A RAISE"?!? YOU'RE FIRED!!
```

- *Table of contents.* Here's a way for you to play around more with center, ljust, and rjust: write a program that displays a table of contents so that it looks like this:

```
                Table of Contents

Chapter 1: Numbers                        page 1
Chapter 2: Letters                        page 5
Chapter 3: Variables                      page 9
```

Higher Math

The rest of this chapter is optional. Some of it assumes a fair degree of mathematical knowledge. If you aren't interested, you can go straight to Chapter 6, Flow Control, on page 33, without any problems. But a quick scan of this chapter might come in handy, especially the section on random numbers.

There aren't nearly as many number methods as there are string methods (though I still don't know them all off the top of my head).

Higher Math

Here, you'll look at the rest of the arithmetic methods, a random number generator (which is pretty handy to know), and the `Math` object, with its trigonometric and transcendental methods (which is less handy for most folks).

More Arithmetic

You've already learned four arithmetic methods: `+`, `-`, `*`, and `/`. The other two arithmetic methods are `**` (exponentiation) and `%` (modulus). So, if you want to say "five squared" in Ruby, you'd write it as `5**2`. You can also use floats for your exponent; so, if you want the square root of 5, you could write `5**0.5`.

The modulus method gives you the remainder after integer division. This might seem a bit bizarre, like integer division did at first, but integer division and modulus are particularly handy when used together. For example, suppose you have $7, and a bag of chips costs $3. Integer division tells you how many bags you can buy (`7/3`, which is 2), and modulus tells you how much money you'll have left afterward (`7%3`, which is 1). Let's see this working in a program:

```
Line 1  puts 5**2
    2  puts 5**0.5
    3  puts 7/3
    4  puts 7%3
    5  puts 365%7
```

```
❮ 25
  2.23606797749979
  2
  1
  1
```

From that last line, you learn that a (non-leap) year has some number of weeks (how many? `365/7`), plus one day. So, if your birthday was on a Tuesday this year, it'll be on a Wednesday next year. You can also use floats with the modulus method. It works the only sensible way it could…but I'll let you play around with that. (In twenty years of programming in Ruby, I don't think I've *ever* used modulus with floats.)

I have one last method to mention before you check out the random number generator: `abs`. This method simply returns the absolute value of the number:

```
Line 1  puts (5-2).abs
    2  puts (2-5).abs
```

```
❮ 3
  3
```

Now that you have a feel for these new arithmetic methods, let's explore how to get Ruby to give you random numbers. (Ruby has been a bit too predictable up to this point, don't you think?)

Random Numbers

Ruby comes with a pretty nice random number generator. The method to get a randomly chosen number is `rand`. If you call `rand` simply like that, you will get a float greater than or equal to `0.0` and less than `1.0`. If you give it an integer (by calling `rand(5)`, for example), it'll give you an integer greater than or equal to `0` and less than `5` (so five possible numbers, from `0` to `4`).

Let's see `rand` in action. In this example, I use a lot of parentheses. Why? Well, when I have several levels of things going on, all on a single line of code, I like to add parentheses to make sure Ruby and I agree on what's supposed to happen. Anyway, let's run it and see what happens:

```
Line 1  puts rand
      -  puts rand
      -  puts rand
      -  puts(rand(100))
      5  puts(rand(100))
      -  puts(rand(100))
      -  puts(rand(1))
      -  puts(rand(1))
      -  puts(rand(1))
     10  puts(rand(99999999999999999999999999999999999999))
      -  puts("My weather app said there is a")
      -  puts("#{rand(101)}% chance of rain,")
      -  puts("but it's literally raining right now!")

   ❮  0.623180804567097
      0.7584826837571998
      0.5071988058744626
      4
      24
      25
      0
      0
      0
      50006848653972728227347008242260298 00
      My weather app said there is a
      39% chance of rain,
      but it's literally raining right now!
```

Note that for the weather example `rand(101)` was used to get numbers from `0` to `100` and that `rand(1)` always returns `0`. Not understanding the range of possible return values is the biggest mistake many people make with `rand`,

even professional programmers and even with finished consumer products. I once had a CD player that, when set to shuffle, would play every song but the last one. (I wonder what would have happened if I'd put in a CD with only one song on it?)

Sometimes you might want rand to return the *same* random numbers in the same sequence on two different runs of your program. (For example, when I worked on the game *Civilization III*, I used randomly generated numbers to generate the game worlds. If I found a world that I really liked, I'd save it, run tests on it, and so on.) To do this, you need to set the *seed*, which you can do with srand:

```
Line 1  srand 1976
      - puts(rand(100))
      - puts(rand(100))
      - puts(rand(100))
      5 puts(rand(100))
      - puts ""
      - srand 1976
      - puts(rand(100))
      - puts(rand(100))
     10 puts(rand(100))
      - puts(rand(100))

      ❮ 50
        21
        80
        15

        50
        21
        80
        15
```

This program will do the same thing every time you seed it with the same number. If you want to get different numbers again (like what happens if you don't call srand at all), then simply call srand, passing in no value at all. This seeds it with a weird number, using (among other things) the current time on your computer, down to the millisecond. After that, you'll start getting new random numbers again.

Now that you know how to use the methods rand and srand to get random numbers from Ruby, we'll finish up with a look at a special object that holds a few other math methods and values you might be interested it.

The Math Object

The Math object holds several mathy methods, like sine and cosine, and also some important mathematical constants, like π. They say a code example is worth a thousand words:

```
Line 1  puts(Math::PI)
    2  puts(Math::E)
    3  puts(Math.cos(Math::PI/3))
    4  puts(Math.tan(Math::PI/4))
    5  puts(Math.log(Math::E**2))
    6  puts((1 + Math.sqrt(5))/2) # golden ratio
```

```
❮ 3.141592653589793
  2.718281828459045
  0.5000000000000001
  0.9999999999999999
  2.0
  1.618033988749895
```

The first thing you noticed was probably the :: notation. Explaining the *scope operator* (which is what that is) is beyond the…uh…scope of this book. No pun intended. I swear. Suffice it to say, you can use Math::PI like any other variable, except that you can't assign a new value to it. (And why would you want to?) This is because Math::PI is actually *not* a variable; it's a constant. It doesn't vary. Remember how I said that variables in Ruby have to start with a lowercase letter? Constants start with an uppercase letter. The main difference between them is that Ruby complains if you try to reassign a constant.

As you can see, Math has all of the features you'd expect a decent scientific calculator to have. And, as always, the floats are *really close* to being the right answers but not exact; don't trust them further than you can calculate them.

Progress Checkpoint

In this chapter, you learned nine new string methods, two new arithmetic methods, two methods for working with random numbers, and the Math object, with the methods and constants that it contains. Your Ruby vocabulary has increased a lot!

Next, you'll learn how to change the structure, the plumbing, the *flow* of your programs, so that they can be far more dynamic and interactive than what you've been able to write so far.

CHAPTER 6

Flow Control

Up to this point, the program examples in this book have been somewhat static and predictable. In other words, you get the same experience each time the program runs. Sure, when the program asks for my name, instead of responding with "Chris," I could answer with "Fromaggio the Odorous," but that's hardly a new experience, right? And just barely interactive.

After this chapter, though, you'll be able to write truly interactive programs. To do this, you need to learn four things: comparison methods, branching, looping, and logic operators. Let's take them in that order.

In the past, you made programs that *said* different things depending on your keyboard input, but after this chapter they'll actually *do* different things. But how will you determine when to do one thing instead of another? You'll use comparison methods.

Comparison Methods

Comparison methods, like > and <, allow you to ask Ruby the question, "Is this greater than (or less than) that?" In the next section you'll learn how we use the answer to these questions to make your program do different things. But before we worry about the answers, let's see how to ask the questions:

```
Line 1 puts 1 > 2
     2 puts 1 < 2

❮    false
     true
```

This looks fairly straightforward, yes?

Likewise, you can find out whether an object is greater than or equal to (or less than or equal to) another with the methods >= and <=:

```
Line 1  puts 5 >= 5
   2  puts 5 <= 4

   ❮  true
      false
```

And finally, you can see whether two objects are equal using == (which asks Ruby, "Are these equal?") and != (which asks, "Are these *not* equal?"):

```
Line 1  puts 1 == 1
   2  puts 2 != 1

   ❮  true
      true
```

It's important not to confuse = with ==. A single = is for telling a variable to point at an object (assignment), and == is for asking the question "Are these two objects equal?" (Having said that, we've all made this mistake before.)

Hey, guess what? You can also compare strings. When strings get compared, Ruby uses their *lexicographical ordering*, which simply means the order they appear in a dictionary. For example, `cat` comes before `dog` in the dictionary, so you have this:

```
Line 1  puts "cat" < "dog"

   ❮  true
```

There's a catch, though. Computers usually order capital letters before lowercase letters. (That's how they store the letters in fonts: all of the capital letters first and then the lowercase ones.) This means the computer will think `"Watermelon Steven"` comes before `"a giant woman"`. So if you want to figure out which word would come first in a real dictionary, make sure to use `downcase` (or `upcase` or `capitalize`) on both words before you try to compare them:

```
Line 1  puts "a giant woman"           < "Watermelon Steven"
   2  puts "a giant woman".downcase < "Watermelon Steven".downcase

   ❮  false
      true
```

Similarly surprising is this:

```
Line 1  puts  2  <  10
   2  puts "2" < "10"

   ❮  true
      false
```

Okay, 2 is less than 10, so no problem. But that last one? Well, the "1" character comes before the "2" character—remember, in a string those are simply characters. The "0" character after the "1" doesn't make the "1" any larger.

One last note before moving on: the comparison methods aren't giving you the strings "true" and "false"; they're giving you the special objects true and false that represent…well, truth and falsity. (Of course, true.to_s gives us the string "true", which is why puts printed true.) true and false are used all of the time in a language construct called *branching*.

Branching

Branching is a simple concept, but it's powerful. It's how we tell Ruby to sometimes do this thing, and other times do that thing. Here's what it looks like in code:

```
Line 1  puts "Hello, what's your name?"
   2  name = gets.chomp
   3  puts "Hello, #{name}."
   4
   5  if name == "Chris"
   6    puts "What a lovely name!"
   7  end
```

```
❮  Hello, what's your name?
⇛  Chris
❮  Hello, Chris.
   What a lovely name!
```

But if we put in a different name…

```
❮  Hello, what's your name?
⇛  Chewbacca
❮  Hello, Chewbacca.
```

And that's branching. If what comes after the if keyword is true, you run the code between the if and the end. If what comes after the if keyword is false, you don't. Plain and simple.

You may have noticed that the code is indented between the if and the end keywords. This makes it easier to keep track of the branching that way. Nearly all programmers do this, regardless of the language they're programming in. It may not seem that helpful in this simple example, but when programs get more complex, it makes a big difference. Often, when people send me programs that don't work but they can't figure out why, the issue is something that is both:

- easier to see what the problem is if the indentation is nice, and
- impossible to see what the problem is otherwise.

So, try to keep your indentation nice and consistent. Have your `if` and `end` keywords line up vertically, and have everything between them indented. Most Ruby programmers use an indentation of two spaces, so you should, too.

Often, you'd like a program to do one thing if an expression is `true` and another if it's `false`. That's what the `else` keyword is for:

```
Line 1  puts "I am a fortune-teller. Tell me your name:"
    2  name = gets.chomp
    3
    4  if name == "Chris"
    5    puts "I see great things in your future."
    6  else
    7    puts "Your future is...oh my! Look at the time!"
    8    puts "I really have to go. Sorry!"
    9  end
```

```
❮ I am a fortune-teller. Tell me your name:
⇛ Chris
❮ I see great things in your future.
```

Now let's try a different name:

```
❮ I am a fortune-teller. Tell me your name:
⇛ Boromir
❮ Your future is...oh my! Look at the time!
  I really have to go. Sorry!
```

Branching is kind of like coming to a fork in the code: do you take the path for people whose `name == "Chris"`, or `else` do you take the other, less egocentric, path?

Like the branches of a tree, code can have branches that themselves have branches:

```
Line 1  puts "Hello, and welcome to seventh grade English."
    -  puts "My name is Mrs. Gabbard. And your name is...?"
    -  name = gets.chomp
    -
    5  if name == name.capitalize
    -    puts "Please take a seat, #{name}."
    -  else
    -    puts "#{name}? You mean #{name.capitalize}, right?"
    -    puts "Don't you even know how to spell your name??"
   10    reply = gets.chomp
    -
    -    if reply.downcase == "yes"
    -      puts "Hmmph! Well, sit down!"
```

```
-   else
15    puts "GET OUT!!"
-   end
-  end
```

```
❮ Hello, and welcome to seventh grade English.
  My name is Mrs. Gabbard. And your name is...?
⇛ chris
❮ chris? You mean Chris, right?
  Don't you even know how to spell your name??
⇛ yes
❮ Hmmph! Well, sit down!
```

Fine, I'll capitalize my name:

```
❮ Hello, and welcome to seventh grade English.
  My name is Mrs. Gabbard. And your name is...?
⇛ Chris
❮ Please take a seat, Chris.
```

Me and Mrs. Gabbard didn't get along terribly well. (Hah! See that, Mrs. Gabbard, using "Me and Mrs. Gabbard" instead of "Mrs. Gabbard and I"? And I'm a published author now!)

Sometimes it might get confusing trying to figure out where all the `if`s, `else`s, and `end`s go. To keep it all straight, try to write the `end` *at the same time* you write the `if`. So, if you were writing the previous program, this is how it would look first:

```
Line 1 puts "Hello, and welcome to seventh grade English."
  2    puts "My name is Mrs. Gabbard. And your name is...?"
  3    name = gets.chomp
  4
  5    if name == name.capitalize
  6    else
  7    end
```

Then you fill it in with *comments*, stuff in the code to help *you* out, but that Ruby will ignore:

```
Line 1 puts "Hello, and welcome to seventh grade English."
  2    puts "My name is Mrs. Gabbard. And your name is...?"
  3    name = gets.chomp
  4
  5    if name == name.capitalize
  6      # She's civil.
  7    else
  8      # She gets mad.
  9    end
```

Anything after a # is considered a comment (unless, of course, the # is in a string). After that, you can replace the comments with working code. Some people like to leave the comments in; personally, I think well-written code often speaks for itself. (The trick, of course, is in writing well-written code.) I used to use more comments, but the more "fluent" in Ruby I become, the less I use them. I actually find them distracting sometimes. Now I usually use comments to explain *why* I choose to solve a problem in this or that way, rather than to explain *what* the code is doing. It's a personal choice; you'll find your own style (which will probably evolve over time, like mine has).

Anyway, the next step in writing this program would probably look something like this:

```
Line 1  puts "Hello, and welcome to seventh grade English."
-       puts "My name is Mrs. Gabbard. And your name is...?"
-       name = gets.chomp
-
5       if name == name.capitalize
-         puts "Please take a seat, #{name}."
-       else
-         puts "#{name}? You mean #{name.capitalize}, right?"
-         puts "Don't you even know how to spell your name??"
10        reply = gets.chomp
-
-         if reply.downcase == "yes"
-         else
-         end
15      end
```

Again, you want to write the if, else, and end all at the same time. It'll help you keep track of "where you are" in the code. It also makes the job seem easier because you can focus on one small part, such as filling in the code between the if and the else lines. The other benefit of doing it this way is that Ruby can understand this program at any stage. Every one of the unfinished versions of the program you've seen here would actually run. They weren't finished, but they were working programs. That way you could test them as you wrote them, which would help you see how your program was coming along and where it still needed work. (And if there's a bug in your program, this would allow you to see it right after you wrote it, which makes it much easier to fix.)

I *strongly* suggest you approach your programs in this way. These tips will help you write programs with branching, but they also help with the other main type of flow control: looping.

Looping

Often, you'll want Ruby to do the same thing over and over again. After all, that's what computers are supposed to be good at doing.

When you tell Ruby to keep repeating something, you also need to say when to stop. Computers never get bored, so if you don't tell them when to stop, they won't.

You make sure this doesn't happen by telling Ruby to repeat certain parts of a program `while` a certain condition is true. It works in much the same way that `if` works:

```
Line 1  input = ""
    2  while input != "bye"
    3    puts input
    4    input = gets.chomp
    5  end
    6  puts "Come again soon!"
```

```
❮
⇛ Hello?
❮ Hello?
⇛ Hi!
❮ Hi!
⇛ Very nice to meet you.
❮ Very nice to meet you.
⇛ Oh, how sweet!
❮ Oh, how sweet!
⇛ bye
❮ Come again soon!
```

It's not a fabulous program, though. For one thing, `while` tests your condition at the top of the loop. This means the program uses `puts` to insert a blank line before the first `gets`. In my mind, it *feels* like the `gets` comes first and the echoing `puts` comes later. It'd be nice if you could say something like this:

```
Line 1  # THIS IS NOT A REAL PROGRAM!
    2  while just_like_go_forever
    3    input = gets.chomp
    4    puts input
    5    if input == "bye"
    6      stop_looping
    7    end
    8  end
    9
   10  puts "Come again soon!"
```

That's not valid Ruby code, but it's surprisingly close. To get it to loop forever, you need to give while a condition that's always true. And Ruby does have a way to break out of a loop:

```
Line 1  # THIS IS TOTALLY A REAL PROGRAM!
  2  while "Spike" > "Angel"
  3    input = gets.chomp
  4    puts input
  5    if input == "bye"
  6      break
  7    end
  8  end
  9
 10  puts "Come again soon!"

 ⇛  Hi, and your name is...
 ❮  Hi, and your name is...
 ⇛  Cute. And original.
 ❮  Cute. And original.
 ⇛  bye
 ❮  bye
    Come again soon!
```

Now, isn't that better? Okay, I'll admit, the "Spike" > "Angel" thing is a little silly. When I get bored coming up with jokes and Buffy references for these examples, I'll use the actual true object:

```
Line 1  while true
  2    input = gets.chomp
  3    puts input
  4    if input == "bye"
  5      break
  6    end
  7  end
  8
  9  puts "Come again soon!"

 ⇛  Hey.
 ❮  Hey.
 ⇛  You again?!
 ❮  You again?!
 ⇛  bye
 ❮  bye
    Come again soon!
```

And that's a loop. It's a bit trickier than a branch, so take a minute to look it over and let it sink in.

Loops are lovely things. But like weapons-grade plutonium or bubble gum, they can cause big problems if handled improperly. Here's a big one: what if

Ruby gets trapped in an infinite loop? If you think this may have happened, go to your command line, hold down the ctrl key, and press c. (You're running these from the command line, right?)

Before you start playing around with loops, though, let's go over a few things to make your job easier.

A Little Bit of Logic

You know how to tell Ruby to do something if a certain condition is true, or to keep doing it while a certain condition is true. That's what you learned in the previous two sections. But you don't know how to tell Ruby to do something if this *or* that is true. In this section, you'll learn how to say "or", "and", and "not" in a way that Ruby will understand.

Let's take another look at the first branching program, on page 35. What if my wife came home from work, saw that program, tried it, and it didn't tell her what a lovely name *she* had? Help me rewrite it so I don't look like a jerk:

```
Line 1  puts "Hello, what's your name?"
  -     name = gets.chomp
  -     puts "Hello, #{name}."
  -
  5     if name == "Chris"
  -       puts "What a lovely name!"
  -     else
  -       if name == "Katy"
  -         puts "What a lovely name!"
 10       end
  -     end

  ❮     Hello, what's your name?
  ⇛     Katy
  ❮     Hello, Katy.
        What a lovely name!
```

Well, it works, but it isn't a pretty program. Why not? It doesn't feel right to me that the whole "Katy" chunk of code isn't lined up with the "Chris" chunk of code. These are supposed to be totally equal and symmetrical options, yet one feels distinctly subordinate to the other. This is profoundly not the kind of marriage we have. You want your code to match, as much as possible, the way you think.

Fortunately, another Ruby construct can help: elsif. This code means the same thing as the last program but feels so much lovelier:

```
Line 1 puts "Hello, what's your name?"
   2 name = gets.chomp
   3 puts "Hello, #{name}."
   4
   5 if name == "Chris"
   6   puts "What a lovely name!"
   7 elsif name == "Katy"
   8   puts "What a lovely name!"
   9 end

   ❮ Hello, what's your name?
  ⇛ Katy
   ❮ Hello, Katy.
     What a lovely name!
```

This is a definite improvement, but something is still wrong. If we want the program to do the same thing when it gets Chris or Katy, then it should really *do the same thing*, as in execute the same code. Here, two different lines of code are doing the same thing. That's still not right.

From a pragmatic point of view, it's a bad idea to duplicate code anywhere. Remember the DRY rule? Don't. Repeat. Yourself. For pragmatic reasons, for aesthetic reasons, or just because you're lazy, don't *ever* repeat yourself. Weed out duplication in code (or even design) whenever you see it. In this case, we repeated the line puts "What a lovely name!". What we're trying to say is, "If the name is Chris or Katy, do this." So let's *code* it that way:

```
Line 1 puts "Hello, what's your name?"
   2 name = gets.chomp
   3 puts "Hello, #{name}."
   4
   5 if name == "Chris" || name == "Katy"
   6   puts "What a lovely name!"
   7 end

   ❮ Hello, what's your name?
  ⇛ Katy
   ❮ Hello, Katy.
     What a lovely name!
```

Nice. Much, much better—and it's even shorter. I don't know about you, but I'm excited. It's almost the same as the original program, too. Bliss, I tell you…sparkly programming bliss.

To make it work, we used ||, which is how you say "or" in most programming languages.

At this point, you might be wondering why you couldn't say this:

```
Line 1  ...
    2
    3  if name == ("Chris" || "Katy")
    4    puts "What a lovely name!"
    5  end
```

It makes sense in English, but you have to remember how staggeringly brilliant humans are compared to computers. The reason this makes sense in English is that humans are just fabulous at dealing with context. In this context, it's clear to a human that "if your name is Chris or Katy" means "if your name is Chris or if it is Katy." (I even used the word "it"—another triumph of human context handling.) But when Ruby sees `("Chris" || "Katy")`, it's not even looking at the `name ==` code; before it gets there, it tries to figure out whether one of `"Chris"` or `"Katy"` is true…because that's what `||` does. But that doesn't really make sense, so you have to be explicit and write the whole thing.

Anyway, that's "or." The other *logical operators* are `&&` ("and") and `!` ("not"). Let's see how they work:

```
Line 1  i_am_chris  = true
    -  i_am_purple = false
    -  i_like_beer = true
    -  i_eat_rocks = false
    5
    -  puts i_am_chris  && i_like_beer
    -  puts i_like_beer && i_eat_rocks
    -  puts i_am_purple && i_like_beer
    -  puts i_am_purple && i_eat_rocks
   10  puts
    -  puts i_am_chris  || i_like_beer
    -  puts i_like_beer || i_eat_rocks
    -  puts i_am_purple || i_like_beer
    -  puts i_am_purple || i_eat_rocks
   15  puts
    -  puts !i_am_purple
    -  puts !i_am_chris

    ❮  true
       false
       false
       false

       true
       true
       true
       false

       true
       false
```

The only one of these that might trick you is ||. In English, we often use "or" to mean "one or the other, but not both." For example, I might say to my kids, "For dessert, you can have pie or cake." I did *not* mean they could have them both. A computer, on the other hand, uses || to mean "one or the other, or both." (Another way of saying it is "at least one of these is true.") This explains why my kids think computers are more fun than I am.

To make sure everything is well cemented for you, let's look at one more example before you go it alone. This will be a simulation of talking to my son, C, back when he was two years old. (C is 17 now, and is unlikely to read his papa's boring old book. Hey, C! I'll give you $50 if you show me this page of the book!) For background, when he talked about Ruby, he was referring to his baby sister, Ruby. He managed to bring everyone he loved into every conversation. (And yes, we did name our children after programming languages. What nerds.) So, this is pretty much what happened whenever you asked little C to do something:

```
Line 1  while true
-         puts "What would you like to ask C to do?"
-         request = gets.chomp
-
5         puts "You say: C, please #{request}."
-
-         puts "C responds:"
-         puts "  C #{request}."
-         puts "  Papa #{request}, too."
10        puts "  Mama #{request}, too."
-         puts "  Ruby #{request}, too."
-         puts
-
-         if request == "stop"
15          break
-         end
-       end

    What would you like to ask C to do?
⇒   eat
    You say: C, please eat.
    C responds:
      C eat.
      Papa eat, too.
      Mama eat, too.
      Ruby eat, too.

    What would you like to ask C to do?
⇒   go potty
    You say: C, please go potty.
    C responds:
      C go potty.
```

```
  Papa go potty, too.
  Mama go potty, too.
  Ruby go potty, too.
What would you like to ask C to do?
⇛ stop
❮ You say: C, please stop.
C responds:
  C stop.
  Papa stop, too.
  Mama stop, too.
  Ruby stop, too.
```

Yeah, that's about what it was like. You couldn't sneeze without hearing about mama or Ruby sneezing, too. ☺

And that's how you write a loop. Now it's time to flex those new branching and looping muscles to really cement that learning.

A Few Things to Try

- *"99 Bottles of Beer on the Wall."* Write a program that prints out the lyrics to this beloved classic song.
- *Deaf grandma.* Whatever you say to Grandma (whatever you type in), she should respond with this:

```
❮ HUH?! SPEAK UP, SONNY!
```

  unless you shout it (type in all capitals). If you shout, she can hear you (or at least she thinks so) and yells back:

```
❮ NO, NOT SINCE 1938!
```

  To make your program *really* believable, have Grandma shout a different year each time, maybe any year at random between 1930 and 1950. (This part is optional and would be much easier if you read the section on Ruby's random number generator on page 30.) You can't stop talking to Grandma until you shout `BYE`.

  Hint 1: Don't forget about `chomp`. `"BYE"` with an Enter at the end isn't the same as `"BYE"` without one.

  Hint 2: Try to think about what parts of your program should happen over and over again. All of those should be in your `while` loop.

  Hint 3: People often ask me, "How can I make `rand` give me a number in a range not starting at zero?" But you don't need it to. Is there something you could do to the number `rand` returns to you?

- *Deaf grandma extended.* What if Grandma doesn't want you to leave? When you shout `BYE`, she could pretend not to hear you. Change your previous program so that you have to shout `BYE` three times *in a row*. Make sure to test your program: if you shout `BYE` three times but not in a row, you should still be talking to Grandma.
- *Leap years.* Write a program that asks for a starting year and an ending year and then prints all the leap years between them (and including them, if they are also leap years). Leap years are years divisible by 4 (like 1984 and 2004). But years divisible by 100 are *not* leap years (such as 1800 and 1900) unless they are also divisible by 400 (such as 1600 and 2000, which were in fact leap years). What a mess.

When you finish those, take a break. That was a *lot* of programming.

Progress Checkpoint

In this chapter you learned branching with `if`, looping with `while`, breaking out of a loop with `break`, and the logical operators for creating more complex conditions for when to branch and loop. Congratulations, you're well on your way.

Relax, have a nice cold (possibly root) beer, and let's pick this back up in the next chapter. You'll be learning how to create and use collections of objects. Using collections of objects allows you to do things that otherwise would simply be impossible.

CHAPTER 7

Arrays and Iterators

Welcome back. You've come a long way to get to this point. The flow control you learned in the last chapter, all that branching and looping, brings you to a whole other level now. With the power to write programs that are richer and more complex, you're going to find that you want more powerful and interesting *data structures*. In this chapter, you'll learn about one such data structure, the *array*.

Let's say you wanted to write a program that asks the user to type in as many words as they want—one word per line until they press Enter on an empty line. Then the program repeats the words back in alphabetical order. Take a moment to think about how you could write such a program.

Well, er... I don't think you could.

The problem is that you need to keep track of an unknown number of words. So far, you've used variables to keep track of your objects, but that won't work here because you might need to keep track of ten words, or a thousand. You won't know until the program is running.

What you need is a way to store a collection of words, and you do that with arrays.

An array is like an electronic list for your computer. Every slot in the list acts like a variable: you can see what object a particular slot points to, and you can make it point to a different object. Let's look at some arrays:

```
Line 1 []
   2 [5]
   3 ["Hello", "Goodbye"]
   4
   5 flavor = "vanilla"             # Not an array, of course...
   6 [89.9, flavor, [true, false]]  # ...but this is.
```

First, there's an empty array, then an array that holds a single number, followed by an array that holds two strings. After that, there's a simple assignment, and finally an array that holds three objects, the last of which is another array: [true, false]. Remember, variables aren't objects, so the last line is simply an array that points to a float, a *string*, and another array. (Even if you were to set flavor to point to something else later in the program, that wouldn't change this array.)

To help you find a particular object in an array, each slot is given an index number. Most programmers (and mathematicians) like to start counting from zero, so the first slot in the array is slot zero. Here's how you'd reference the objects in an array:

```
Line 1 gems = ["Pearl", "Garnet", "Amethyst"]
   2
   3 puts gems
   4 puts         # just a blank line
   5 puts gems[0]
   6 puts gems[1]
   7 puts gems[2]
   8 puts gems[3] # This is out of range.
```

```
Pearl
Garnet
Amethyst

Pearl
Garnet
Amethyst
```

Notice that puts gems prints each string in the gems array, whereas puts gems[0] prints the *first* gem in the array, and puts gems[1] prints the second gem in the array. I'm sure this seems confusing, but you'll eventually get used to it. Adjust your thinking so that counting begins at zero, and stop using words such as *first* and *second*. If you have a five-course meal, don't talk about the first course; talk about course zero (and in your head, be thinking course[0]). You have five fingers on your right hand, and their numbers are 0, 1, 2, 3, and 4. My wife and I are jugglers. When we juggle six clubs, we're juggling clubs 0–5. In the next few months, we hope to be able to juggle club 6 (and thus be juggling seven clubs between us). You'll know you have it when you start using the word *zeroth*. ☺

Finally, the code example shows puts gems[3], which is out of range. Were you expecting an error? As you've seen in the past, sometimes when you ask Ruby a question, it doesn't make any sense (at least not to Ruby); that's when you get an error. Sometimes, however, you can ask a question, and the answer

is *nothing*. What's in slot three? Nothing. What is gems[3]? *nil*: Ruby's way of saying "nothing." nil is a special object that means "not any other object." And when you use puts nil, it prints out nothing. (Actually, a new line.)

Now, I said the slots in your arrays act like variables. This means you can assign to them as well. If you had to guess what that code looked like, you'd probably guess something like this:

```
Line 1  other_gems = []
   2    other_gems[3] = "Ruby"
   3    other_gems[0] = "Pink Diamond"
   4    other_gems[2] = "Sapphire"
   5    other_gems[0] = "Rose Quartz"
   6    puts other_gems
```

```
Rose Quartz
Sapphire
Ruby
```

As you can see, you don't have to assign to the slots in any particular order, and any you leave empty are filled with nil by default.

If this funny numbering of array slots is getting to you, fear not. Often, you can avoid them completely by using various array methods, such as each.

My First Iterator

The method each allows you to do something (whatever you want) to *each* object the array points to. (It looks weird, though, and this can throw people off, so brace yourself.) For example, if you want to say something nice about each language in the following array, you could do something like this:

```
Line 1  languages = ["English", "Norwegian", "Ruby"]
   2
   3    languages.each do |lang|
   4      puts "I love #{lang}! Don't you?"
   5    end
   6
   7    puts "And let's hear it for Fortran!"
   8    puts "<crickets chirp in the distance>"
```

```
I love English! Don't you?
I love Norwegian! Don't you?
I love Ruby! Don't you?
And let's hear it for Fortran!
<crickets chirp in the distance>
```

What happened here? (Aside from the gratuitous dissing of Fortran.) Well, you were able to go through every object in the array without using any

index numbers, no "counting starts at zero," so that's definitely nice. You see those weird vertical-bar-thingies around lang; I'll get to that. But first, to make sure you understand what this code means (if not necessarily *why* it means it), let's translate it into English: for each object in languages, point the new variable lang to the object, and then do everything I tell you to, until you come to the end.

So the first time through, lang is pointing to languages[0], which is the string "English". The second time, lang is pointing to languages[1], which is "Norwegian", and so on.

You use do and end to specify a block of code. In this case, you're sending that block to the each method, saying, "This is what I want you to do with each of the objects in the array." Blocks are great, but a bit advanced, which is why we're not *really* going to talk about them until Chapter 13, Blocks and Procs, on page 113. Until then, you can still use them, but we just won't talk about them. Much.

Except we'll talk about the vertical-bar-thingies, like in |lang|. It looks weird, but the idea is simple: lang is the variable that each will use to point to the objects in the array. How would you otherwise refer to the string "English"? (Well, maybe using languages[0], but the entire point here was to avoid messing with the slot numbers.) The vertical bars don't *do* anything to lang; they simply let each know which variable to use to feed in the objects from the array.

You might be thinking to yourself, "This is a lot like the loops from earlier." Yep, it's similar. One important difference is that the method each is simply that: a method. while and end (much like do, if, else, and all the other keywords) are not methods. They are a fundamental part of the Ruby language, like = and parentheses; they are kind of like punctuation marks in English.

But this isn't true with each; each is simply another method. (In this case, it's a method that all arrays have. We'll see more array methods in the next section.) Methods like each that "act like" loops are often called *iterators* because they iterate over each element in some collection.

One thing to notice about iterators is that they're always followed by a block —that is, by some code wrapped inside do…end. On the other hand, while and if never have a do near them.

Here's another cute little iterator, but this one isn't an array method:

```
Line 1 # integer method FTW
  2 3.times do
  3   puts "Hip-Hip-Hooray!"
  4 end
```

```
Hip-Hip-Hooray!
Hip-Hip-Hooray!
Hip-Hip-Hooray!
```

It's an integer method. Now you cannot *tell* me that ain't the cutest code you've ever seen. And, as promised on page 12, here's that pretty program again:

```
2.times do
  puts "...you can say that again..."
end
```

```
...you can say that again...
...you can say that again...
```

Aside from each, a few other important array methods exist that you'll find useful. Let's check them out.

More Array Methods

You've learned about each, but many other array methods exist, almost as many as string methods. In fact, some of them (such as length, reverse, +, and *) work the same way they do for strings, except they operate on the slots of the array rather than on the letters of the string. Others, such as last and join, are specific to arrays. Still others, such as push and pop, actually change the array. As with the string methods, you don't have to remember all of these, as long as you can remember where to find out about them (and that would be right here).

Let's look at to_s and join first. to_s gives you a nice readable string description, in this case making it clear that this is an array of strings. join calls to_s on the individual elements of the array (which does nothing in this case, since they are already strings), and it adds the string you provide in between those, like this:

```
foods = ["artichoke", "brioche", "caramel"]

puts foods
puts
puts foods.to_s
puts
puts foods.join(", ")
puts
puts foods.join("  :)  ") + "  8)"

200.times do
  puts []
end
```

Two hundred times? No more sugar for me!

```
artichoke
brioche
caramel

["artichoke", "brioche", "caramel"]

artichoke, brioche, caramel

artichoke  :)  brioche  :)  caramel  8)
```

Whew! It's a good thing that puts treats arrays differently from other objects; that would have been a boring couple of pages if puts had written 200 blank lines. With arrays, puts calls puts on each of the objects in the array. That's why calling puts on an empty array 200 times doesn't do anything; the array doesn't contain anything, so there's nothing to call puts on. Doing nothing 200 times is still doing nothing (unless you're grinding in *Final Fantasy*, in which case you just leveled). Try calling puts on an array containing other arrays. What do you think will happen? For real, try it! Did it do what you expected?

Now, let's take a look at push, pop, and last. The methods push and pop are sort of opposites, like + and -. push adds an object to the end of your array, and pop removes the last object from the array (and tells you what it was). last is similar to pop in that it tells you what's at the end of the array, except that it doesn't change the array at all. Again, push and pop *actually change the array*:

```
favorites = []
favorites.push("raindrops on roses")
favorites.push("whiskey on kittens")

puts favorites[0]
puts favorites.last
puts favorites.length

puts favorites.pop
puts favorites
puts favorites.length
```

```
raindrops on roses
whiskey on kittens
2
whiskey on kittens
raindrops on roses
1
```

Make sense? You won't know for sure until you use it in a program, so here are some projects to try out.

A Few Things to Try

- *Building and sorting an array.* Write the program you saw at the beginning of this chapter, the one that asks you to type as many words as you want (one word per line, continuing until you press `Enter` on an empty line) and then repeats the words back in alphabetical order. Make sure to test your program thoroughly; for example, does hitting `Enter` on an empty line *always* exit your program? Even on the first line? And the second? *Hint:* There's a lovely array method that'll give you a sorted version of an array: `sort`. Use it!

- *Table of contents, revisited.* Rewrite your table of contents program on page 28. Start the program with an array holding all of the information for your table of contents (chapter names, page numbers, and so on). Then print out the information from the array in a beautifully formatted table of contents.

Progress Checkpoint

Excellent! You now know what arrays are, and you've learned several array methods, including the iterator `each`. Using arrays, you can write richer, more interesting programs than you could before. At this point, I've lost track of the number of methods you've learned, but it's a lot. The next step on your journey is to learn how to create brand new methods of your own. This will enable you to write even more powerful programs, with less code and less repetition, and to be the laziest programmer you can be.

CHAPTER 8

Custom Methods

As you've seen, loops and iterators allow you to do the same thing (run the same code) over and over again. But sometimes you want to do the same thing a number of times but not back-to-back in a loop. For example, let's say you're writing a program for a flavor tournament. (I mean, I guess it's a regular tournament program, but instead of people or teams, I'm putting in ice-cream flavors.)

It might look something like this:

```
Line 1  match_1 = ["vanilla", "chocolate"]
  -     match_2 = ["rhubarb", "pistachio"]
  -     match_3 = []  # this will hold the winners from 1 & 2
  -     winner  = nil # this will hold the final winner
  5
  -     puts "Welcome to ULTIMATE FLAVOR TOURNAMENT!"
  -     puts
  -     puts "MATCH 1: Which flavor is best?"
  -     puts "0. "+match_1[0]
 10     puts "1. "+match_1[1]
  -     while true
  -       answer = gets.chomp.downcase
  -       if (answer == "0" || answer == "1")
  -         match_3[0] = match_1[answer.to_i]
 15         break
  -       else
  -         puts "Please answer '0' or '1'."
  -       end
  -     end
 20
  -     puts
  -     puts "MATCH 2: Which flavor is best?"
  -     puts "0. "+match_2[0]
  -     puts "1. "+match_2[1]
 25     while true
  -       answer = gets.chomp.downcase
```

```
  if (answer == "0" || answer == "1")
    match_3[1] = match_2[answer.to_i]
    break
  else
    puts "Please answer '0' or '1'."
  end
end

puts
puts "CHAMPIONSHIP MATCH!"
puts "Which flavor is best?"
puts "0. "+match_3[0]
puts "1. "+match_3[1]
while true
  answer = gets.chomp.downcase
  if (answer == "0" || answer == "1")
    winner = match_3[answer.to_i]
    break
  else
    puts "Please answer '0' or '1'."
  end
end

puts
puts "And the Ultimate Flavor Champion is:"
puts winner.upcase+"!!"
```

Alright, let's see it in action:

```
Welcome to ULTIMATE FLAVOR TOURNAMENT!

MATCH 1: Which flavor is best?
0. vanilla
1. chocolate
vanilla
Please answer '0' or '1'.
0

MATCH 2: Which flavor is best?
0. rhubarb
1. pistachio
1

CHAMPIONSHIP MATCH!
Which flavor is best?
0. vanilla
1. pistachio
1

And the Ultimate Flavor Champion is:
PISTACHIO!!
```

Note that I tried typing out "vanilla" first, instead of answering with 0 or 1. This is because you *always* want to test every path in your code. What if there was a bug in the code that asks the user to type a 0 or 1? You wouldn't know until you showed this program to someone else and they found the bug. Or worse, until it's immortalized in a book and someone reads it, tries out the code, and it crashes. *This is the stuff of my literal nightmares.*

Anyway, apologies to those pistachio ice cream haters out there. Honestly, I prefer almost any kind of potato chips to almost any flavor of ice cream, but we made rhubarb ice cream last night, so it's on my mind. (It's pretty good, made with fresh rhubarb from our garden. But it's no pistachio.)

Back to that program, it's kind of ugly, with lots of ugly repetition. As you saw before, repetition is a Very Bad Thing. I guess you could *maybe* put it in a loop? It would reduce the repetition, but the code would be awkward and weird. Like how you store the answers in three different places for the three different rounds; how would that even look?

In situations like these, it's best to write a method of your own. Let's start with something small first, and you'll return to the flavor tournament later.

Let's write a method that prints "moo":

```
Line 1 def say_moo
     2   puts "mooooooo..."
     3 end
```

```
❮
```

Um…the program didn't `say_moo`. Why not? Because you didn't tell it to. You told it *how* to `say_moo`, but you never actually said to *do* it. Let's give it another shot:

```
Line 1 def say_moo
     2   puts "mooooooo..."
     3 end
     4
     5 say_moo # once...
     6 say_moo # twice...
     7 say_moo # three times a bovine
```

```
❮ mooooooo...
  mooooooo...
  mooooooo...
```

Ahhh, much better.

So, you `def`(ined) the method `say_moo`. (Method names, like variable names, almost always start with a lowercase letter. A few exceptions exist, though,

such as + or ==.) But don't methods always have to be associated with objects? Well, yes, they do, and in this case (as with puts and gets), the method is associated with the object representing the whole program. In Chapter 11, Custom Classes and Class Extensions, on page 89, you'll see how to add methods to other objects.

But first, you should learn how to make your methods more flexible by giving them *arguments*.

Method Arguments: What Goes In

You may have noticed that some methods (such as gets, reverse, to_s, and so on) can just be called on an object. But other methods (such as +, -, puts…) take additional values to tell the object how to do the method. For example, you wouldn't just say 5+, right? You're telling 5 to add, but you aren't telling it *what* to add.

To change say_moo to accept an argument (how about the number of moos?), you'd do the following:

```
Line 1  def say_moo(number_of_moos)
  2       puts "mooooooo..."*number_of_moos
  3     end
  4
  5     say_moo(3)
  6     puts "oink-oink"
  7
  8     # This last line should give an error
  9     # because the argument is missing...
 10     say_moo
```

```
❮ moooooooo...moooooooo...moooooooo...
  oink-oink
  example.rb:1:in `say_moo': wrong number of arguments (given 0, expected 1)
          (ArgumentError)
          from example.rb:10:in `<main>'
```

The variable number_of_moos points to the argument passed in. I'll say that again, because it's a little confusing: number_of_moos is a variable that points to the argument passed in. So, if you type say_moo(3), the variable number_of_moos points to the value 3 when the method is being run.

As you can see, the argument is now *required*. After all, what is say_moo supposed to multiply "mooooooo..." by if you don't give it an argument? Poor Ruby has no idea.

If objects in Ruby are like nouns in English and methods are like verbs, then you can think of arguments as adverbs (like with say_moo, where the argument

told you *how* to say_moo) or sometimes as direct objects (like with puts, where the argument is *what* gets putsed).

Sometimes these special variables that methods have, like number_of_moos, are called *parameters*: parameters are the variables that point to arguments. But they aren't the only kind of variables your methods can have. To help you write clean, encapsulated code, you can also create *local variables*.

Local Variables: What's Inside

The following program has two variables:

```
Line 1  def double_this(num)
  2       num_times_2 = num*2
  3       puts "#{num} doubled is #{num_times_2}"
  4     end
  5
  6     double_this(44)

❮ 44 doubled is 88
```

The variables are num (which is a parameter) and num_times_2. They both sit inside the method double_this. These (and all the variables you've seen so far) are *local variables*. This means they live inside the method, and they cannot leave. If you try to access them outside of the method, you'll get an error:

```
Line 1  def double_this(num)
  2       num_times_2 = num*2
  3       puts "#{num} doubled is #{num_times_2}"
  4     end
  5     double_this(44)
  6
  7     # this will not work!
  8     puts num_times_2.to_s

❮ 44 doubled is 88
example.rb:8:in `<main>': undefined local variable or method `num_times_2'
        for main:Object (NameError)
```

Undefined local variable.... In fact, you *did* define that local variable, but it isn't local to where you tried to use it; it's local to the method, which means it's local to double_this: it only exists while double_this is executing.

This might seem inconvenient, but it's actually nice. Although it does mean you have no access to variables inside methods, it also means they have no access to *your* variables and thus can't screw them up, as the following example shows:

```
Line 1  tough_var = "You can't even touch my variable!"
   2
   3  def little_pest(tough_var)
   4    tough_var = nil
   5    puts "HAHA! I ruined your variable!"
   6  end
   7
   8  little_pest(tough_var)
   9  puts tough_var
```

```
HAHA! I ruined your variable!
You can't even touch my variable!
```

In fact, *two* variables in that little program are named tough_var: one is the parameter of little_pest, and the other is outside of the method. They don't communicate. They aren't related. They aren't even friends. When you called little_pest(tough_var), you passed the string from one tough_var to the other (via the method call, the only way they can communicate at all) so that both were pointing to the same string. Then little_pest pointed its own *local* tough_var to nil, but that did nothing to the tough_var variable outside the method.

Okay, local variables, neat trick…but maybe you're wondering, "What's the point?" Legitimate question, but it's kind of hard to see the point without considering what it would be like *without* local variables. So let's do a thought experiment about what programming would be like without them.

Experiment: Stuby

When I do the dishes, I put the glasses where the glasses go and the bowls where the bowls go. So, when I need a glass, I don't have to go sifting through a bunch of bowls (and plates, and spatulas, and novelty mugs, and corn-cob holder things, and more spatulas, little tiny spatulas that I don't think we ever even use, and…). The utensils are in a lovely utensil tray, so when we have an urgent ice cream emergency, I don't even have to search through soup spoons or serving spoons to find the dessert spoons.

The Perfect Utensil Tray

This aside isn't a metaphor: I'm really excited about my utensil tray and wanted to brag about it. It's the exact dimensions of my utensil drawer, because I made it myself, using Ruby!

I'm not what you'd call a handyperson; I barely know which end of a hammer is the handle. But I do own a laser cutter, and I'm not a bad programmer. So I wrote a little Ruby program to define the shapes of all of the pieces I'd want for the perfect utensil tray,

The Perfect Utensil Tray

and output those shapes in a text-based format called *SVG*. The laser cutter can read SVG and cut out those shapes, and I'm now the envy of utensil tray aficionados everywhere.

Anyway, back to the metaphor: without a way to keep unrelated components separate and organized, your kitchen would be a disaster. The same is true of your code, but it's a little harder to see the ways in which local variables help accomplish this. To get a feel for life without local variables, let's imagine a fake language called Stuby (for *Stupid Ruby*).

Stuby is like regular Ruby, except that all variables live in the same scope (that is, they have the same visibility). There's no concept of local or global variable, no local or global scope.

Let's say you wanted to make a method to print the square of a number:

```
Line 1 def square(x)
     2   puts(x * x)
     3 end
```

You've used the variable `x` here, but you're only using it as a placeholder. You could as well have used `y` or `my_fabulous_number`; it should make no difference, right? The whole point is to say that the square of *something* is "something times something." That's the abstraction that methods provide. In this particular case, no matter what variable you're squaring, "the square of something" simply means "something times something." That's what you're trying to say when you define this method.

Now, let's say, in Stuby, you wrote this:

```
Line 1 x = 5
     2 square(x)
```

At this point, assuming you defined `square` like you did earlier, then `x` is pointing to `5`, before and after you called `square`. No problem.

And what about this program (again, in Stuby)?

```
Line 1 my_number = 5
     2 square(my_number)
```

It's the same program but with a different variable name. Now `my_number` is pointing to `5`. But what about `x`? What's it pointing to?

I guess it would also have to be `5`. To use the `square` method, the value passed into it (`5`, pointed to by `my_number`) needs to be assigned to `x` (that is, have `x` point to it) before you can run the `x * x` code. So, `x` is `5`. So far, so good.

Now consider this code:

```
Line 1  x = 10
    2  my_number = x / 2
    3  square(my_number)
    4  puts x # prints "5"
```

In this case, my_number is half of x (so it must be 5), but that means x must also have been set to 5 when you called square, even though you had just set it to 10. And this is the real problem: calling the square method displays the squared value, but it *also* has the nasty side effect of resetting x to be whatever was passed in. This is Just Plain Wrong. I mean, x used to be 10! Now it's 5! *No, thank you.*

What if you don't *want* any of your variables to be changed? What if you only want to display the square of whatever number you pass in? Can you get around the problem of the unintended side effects? Maybe.

You could start defining all of your methods like this:

```
Line 1  def square(liauwechygfajtuewhalf)
    2    puts(liauwechygfajtuewhalf * liauwechygfajtuewhalf)
    3  end
```

And you could maybe hope that you don't use liauwechygfajtuewhalf anywhere else in the program, but that hardly seems ideal. ☺ I'm not going to write code like that, and neither should you. Can't we do better than this?

What if the x used in the square method was a *different* x, a totally private x that you use only for the square method; it doesn't mean "the x you were using." It's only a temporary name for this value. It's a *local* variable.

That's exactly what Ruby does. (And almost every other programming language.)

You've learned about arguments (and the parameters that point to them) at the beginning of a method call. You've learned about using local variables during your method call. Now it's time to learn what happens at the end of a method call.

Return Values: What Comes Out

You may have noticed that some methods give you something back when you call them. For example, you say gets *returns* a string (the string you typed in), and the + method in 5+3 (which is actually 5.+(3)) returns 8. The arithmetic methods for numbers return numbers, and the arithmetic methods for strings return strings.

It's important to understand the difference between a method returning a value (returning it to the code that called the method), and your program outputting text to your screen, like puts does. Notice that 5+3 returns 8; it does *not* print 8 on your screen.

So, what *does* puts return? We never cared before, but let's look at it now:

```
Line 1 return_val = puts "This puts returned:"
   2 puts return_val

❮  This puts returned:
```

The puts on line 1 didn't seem to return anything, and in a way it didn't; actually, it returned nil. Though you didn't test it, the puts on line 2 also did; puts always returns nil. In fact, in Ruby, *every* method returns something, even if it's only nil.

Take a quick break, and write a program to find out what say_moo returns.

Are you surprised? Well, here's how it works: the value returned from a method is simply the last *expression* evaluated in the method (usually the last line of the method). In the case of say_moo, this means it returns puts "mooooooo..."*number_of_moos, which is nil since puts always returns nil. If you wanted all your methods to return the string "yellow submarine", you'd need to put *that* at the end of them:

```
Line 1 def say_moo(number_of_moos)
   2   puts "mooooooo..."*number_of_moos
   3   "yellow submarine"
   4 end
   5
   6 x = say_moo(3)
   7 puts x.capitalize + ", dude..."
   8 puts x            + "."

❮  moooooooo...moooooooo...moooooooo...
   Yellow submarine, dude...
   yellow submarine.
```

(I have no idea why you'd want say_moo to work that way, but now you know how.)

Notice I said "the last expression evaluated" instead of simply "the last line" or even "the last expression"; it's possible for the last line to be only a small part of an expression (like the end in an if expression), and it's possible for the last expression not to be evaluated at all if the method has an explicit return:

```
Line 1  def favorite_food(name)
        if name == "Lister"
          return "vindaloo"
        end
      5
        if name == "Rimmer"
          return "mashed potatoes"
        end
        "hard to say...maybe fried plantain?"
     10 end
        def favorite_drink(name)
        if name == "Jean-Luc"
     15   "tea, Earl Grey, hot"
        elsif name == "Kathryn"
          "coffee, black"
        else
          "perhaps...horchata?"
     20 end
        end
        puts favorite_food("Rimmer")
        puts favorite_food("Lister")
     25 puts favorite_food("Cassandra")
        puts favorite_drink("Kathryn")
        puts favorite_drink("Q")
        puts favorite_drink("Jean-Luc")
```

```
mashed potatoes
vindaloo
hard to say...maybe fried plantain?
coffee, black
perhaps...horchata?
tea, Earl Grey, hot
```

Make sure you follow each of the six delicious examples.

I did two different things in that program: with favorite_food I used explicit returns, and in favorite_drink I didn't. Depending on the situation, I'll write a method one way or the other. If I'm trying to prune off special cases, I might use returns and leave the general case on the last line. If I think the options are all of relatively equal importance, I might use elsif and else, like that…feels more egalitarian, you know?

I feel like I'm supposed to tell you now that it doesn't make any difference which approach you use; it's *just* a matter of style. But I don't believe that. You want your code to reflect your *intent*, not only the solution. You want your code to be beautiful.

Okay, now that you know how to write new methods, let's try that flavor tournament program again. This time you'll write a method to ask the questions for you. It'll need to take a pair of flavors as a parameter and return the winner:

```
Line 1  match_1 = ["vanilla", "chocolate"]
-       match_2 = ["rhubarb", "pistachio"]
-       match_3 = []  # this will hold the winners from 1 & 2
-       winner  = nil # this will hold the final winner
5
-       def ask_for_winner(flavors)
-         puts "0. "+flavors[0]
-         puts "1. "+flavors[1]
-
10        while true
-           answer = gets.chomp.downcase
-           if (answer == "0" || answer == "1")
-             return flavors[answer.to_i]
-           else
15            puts "Please answer '0' or '1'."
-           end
-         end
-       end
-
20      puts "Welcome to ULTIMATE FLAVOR TOURNAMENT!"
-       puts
-       puts "MATCH 1: Which flavor is best?"
-       match_3[0] = ask_for_winner(match_1)
-       puts
25      puts "MATCH 2: Which flavor is best?"
-       match_3[1] = ask_for_winner(match_2)
-       puts
-       puts "CHAMPIONSHIP MATCH!"
-       puts "Which flavor is best?"
30      winner = ask_for_winner(match_3)
-       puts
-       puts "And the Ultimate Flavor Champion is:"
-       puts winner.upcase+"!!"
```

Now let's see it in action:

```
❮ Welcome to ULTIMATE FLAVOR TOURNAMENT!

  MATCH 1: Which flavor is best?
  0. vanilla
  1. chocolate
⇛ 0
❮
  MATCH 2: Which flavor is best?
  0. rhubarb
  1. pistachio
⇛ 1
```

```
❮
CHAMPIONSHIP MATCH!
Which flavor is best?
0. vanilla
1. pistachio
⇛ 1
❮
And the Ultimate Flavor Champion is:
PISTACHIO!!
```

Not bad, huh? The program is about half as long as it used to be, with a lot less repetition of code, so that's always good. Making changes to the asking code is *easy* now because you only have to change it in one place. Nice…a lazy programmer's dream.

One last thing to note: on line 13 you broke out of the loop without using `break`. This is because `return` exits the method immediately, even in a loop. (You could have added a `break` right after `return`, but that code would never be reached because of the `return`, so there wouldn't be much point.)

Now that you know how to write methods, here are some exercises to practice your new skill.

A Few Things to Try

- *More flavors competing.* Add some more flavors to the ULTIMATE FLAVOR TOURNAMENT. Was it difficult to do? What changes would you need to make so that it's easy to add new flavors to the tournament?

- *Old-school Roman numerals.* In the early days of Roman numerals, the Romans didn't bother with any of this new-fangled subtraction "IX" nonsense. Nope, it was straight addition, biggest to littlest—so 9 was written "VIIII," and so on. Write a method that when passed an integer between 1 and 3000 (or so) returns a string containing the proper old-school Roman numeral. In other words, `old_roman_numeral(4)` should return `"IIII"`. Make sure to test your method on a bunch of different numbers. *Hint:* Use the integer division and modulus methods on page 29.

  For reference, these are the values of the letters they used:

  I = 1 V = 5 X = 10 L = 50
  C = 100 D = 500 M = 1000

- *"Modern" Roman numerals.* Eventually, someone thought it would be terribly clever if putting a smaller number before a larger one meant you had to subtract the smaller one. (My bet is that it was a stone carver in some year that ended in a 9, tasked with dating public buildings or statues

or something.) As a result of this development, you must now suffer. Rewrite your previous method to return the new-style Roman numerals, so when someone calls `roman_numeral(4)`, it should return `"IV"`.

Progress Checkpoint

Congratulations! In this chapter you learned how to create methods, how to pass in arguments to your methods, how to return objects from your methods, and how local variables inside methods work. You're finally ready to learn something *dangerous*. If this were a book on sorcery (and bummer for you that it's not), this would be the point where you go from illusion spells to shooting fireballs, and we'd start with some basic fire safety lessons.

In the next chapter you'll learn how to interact with the files on your computer, and how *not* to burn them all to bits in the process.

CHAPTER 9

File Input and Output

Now if you were sitting here next to me, you'd probably ask why I'm still in my pajamas. (Well, I didn't know you were coming; you weren't here like ten seconds ago.) But if you'd given more warning and I was wearing actual grown-up clothes, you'd be more likely to say something like, "Chris, I still can't write a program that really *does* anything."

And I'd say, "Yep."

Really Doing Something

So far, after your program is done running, there's no evidence that it ever ran (aside from your memory of it). Nothing on your computer has been changed at all. The least I can do is show how to save the output of your program. For example, let's say you wanted to save the output of your nifty new "99 Bottles of Beer on the Wall" program. All you have to do is add a little bit onto the command line when you run it:

```
ruby 99bottles.rb > lyrics.txt
```

That's not even programming; it's a command-line trick. And it's not a terribly exciting one since you can't use it for a program with any kind of interactivity. You also can't use it to save to more than one file or to save at a time other than the end of the program. But, hey—it's something. What's happening is that all of the program's output (from every time you called `puts`) is being grabbed and funneled to the named file instead of being printed on your screen.

The Thing about Computers

Before we get to real saving and loading, you and I need to talk about something. Something important. It's about computers. The thing about

computers—desktops, laptops, any of your devices—is that, well, they suck. This isn't by nature, mind you—it isn't intrinsic—and I yearn for the day when they no longer suck. But for the moment, by (poor) design, they do. This is most painfully seen when your computer loses a bunch of your information.

A few years back I was working on a project. (Thankfully, I was the only one on the project at the time.) To make a long story short, I dropped the database. All of the information in the database—gone. The very structure of the database (which was itself days and days of work for me)—gone. It was all completely gone. It felt like Scotty had beamed up my stomach but forgot the rest of me. I walked around for several hours, feeling sick, muttering, "I can't believe I dropped the database…I can't believe I dropped the database…." It was horrible.

You know how I did it? It was a single click of the mouse, about 15 pixels too high, followed by a totally reflexive (at that point) hitting of the OK button on the confirmation pop-up. And it was all gone.

I clicked the button next to the one I wanted to click, and hit OK on the pop-up I didn't ever pay attention to, because seriously that thing was always popping up. User error, you say? Yeah, I suppose it was. I certainly blamed myself. But at some point, you have to ask yourself, why is it so fast and so easy to screw things up so catastrophically? At some point you have to start blaming the computer.

I'm telling you this now because if you start writing programs that can actually do something, it means they can do something *bad*. Now is when you have to be careful. Make backups. Put your programs in Google Drive, Dropbox, iCloud, or OneDrive. Even better, put your programs on GitHub like one of the cool kids.

From this point on, be careful, okay?

Saving and Loading for Grown-Ups

Now that I've freaked you out, let's get to it. A file is merely a sequence of bytes. A string is also, ultimately, a sequence of bytes. This makes saving strings to files pretty easy, at least conceptually. (And Ruby makes it pretty easy in practice.)

Here's a quick example where we save a simple string to a file and then read it back out again:

```
Line 1 # The filename doesn't have to end with ".txt",
  - # but since it is valid text, why not?
  - filename    = "ListerQuote.txt"
  - test_string = "I promise that I swear absolutely that " +
  5               "I will never mention gazpacho soup again."
  -
  - # The "w" here is for write-access to the file,
  - # since we are trying to write to it.
  - File.open(filename, "w") do |f|
 10   f.write(test_string)
  - end
  -
  - read_string = File.read(filename)
  -
 15 puts(read_string == test_string)

 ❮ true
```

File.open is how you open a file, of course. It creates the file object, calls it f (because that's what we said to call it), runs all the code until it gets to the end, and then closes the file. When you open a file, you always have to close it again. (In most programming languages you have to remember to do this yourself, but Ruby takes care of it for you at the end). Reading files is even easier than writing them; with File.read, Ruby takes care of everything behind the scenes. (I'm not sure why they made writing more complicated, but we'll fix that in a bit.)

That's all well and good if all you want to save and load are single strings. But what if you want to save an array of strings? Or an array of integers and floats? And what about all of the other classes of objects that we don't even cover until the next chapter? *What about the bunnies?*

All right, one thing at a time. Now we can definitely save any kind of object as long as we have some well-defined way of converting from a general object to a string and back again. So, maybe an array would be represented as text separated by commas. But what if you wanted to save a string with commas? Well, maybe you could escape the commas somehow....

Figuring this all out would take us a ridiculous amount of time. I mean, it is pretty cool that you can do it at all, but you didn't pay good money for "pretty cool." No sir, this is a De-Luxe-Supremium book you have here. And for that, my friend, we need some serious saving. We need some legit loading. We need JSON.

JSON

JSON is the most popular, most standard way of *serializing* the objects in your running program (encoding them as text), and *deserializing* them (turning the text back into objects).

JSON stands for "JavaScript Object Notation," but it's now used by almost every language (not just JavaScript) as a way to pass objects back and forth, especially over the Internet. JSON is nice because, even though it's a way for computers to talk to each other, it's still a human-readable (and human-editable) format.

JSON isn't actually part of the Ruby core, but it's part of the Ruby standard distribution. What does that mean? Well, when you install Ruby, you install the JSON parts of it, too. But since you probably don't want to use JSON in every Ruby program you write, it's not loaded unless you tell Ruby that you require it, like this:

```
Line 1  require "json" # Sir, I require the JSON.
-
-  test_array = ["Give Quiche A Chance",
-                "Mutants Out!",
5               "Chameleonic Life-Forms, No Thanks"]
-
-  # Here's half of the magic:
-  test_string = test_array.to_json
-  # You see? Kind of like 'to_s', and it is in fact a string,
10 # but it's a JSON description of 'test_array'.
-
-  filename = "RimmerTShirts.txt"
-
-  File.open(filename, "w") do |f|
15   f.write(test_string)
-  end
-
-  read_string = File.read(filename)
-
20 # And the other half of the magic:
-  read_array = JSON.parse(read_string)
-
-  puts(read_string == test_string)
-  puts(read_array  == test_array )

❮ true
  true
```

Simple. With only two extra lines of code (well, three if you count the require line at the top), we went from being able to save and load strings to being able to save and load all kinds of objects: strings, numbers, special objects (such

as true, false, and nil), arrays of any of these, and other kinds of objects we haven't even seen yet.

At this point you might be wondering what these JSON strings look like. Run the previous example yourself, and you'll see this text saved in RimmerTShirts.txt:

```
["Give Quiche A Chance","Mutants Out!","Chameleonic Life-Forms, No Thanks"]
```

Well, check that out! It looks an awful lot like an array of strings.

Before we continue, though, can we take a break? I'm really, like *really* hungry. Let's pick this back up in 15 minutes or so, okay?

Okay.

Back to Our Regularly Scheduled Programming

Now where were we? Ah, yes, JSON. Go ahead and play around with your JSON code. Get familiar with it. Toss in some arrays within arrays; try to fool it with the integer 42 as opposed to the string "42" or with the true object as opposed to the string "true". JSON is pretty smart.

You know what would be even better, though? It'd be cool if you could save an object with one method call, one single line of code. And it'd be cool if you could load with one method call, too.

We are programmers; we aren't limited to the tools provided to us. We can craft our own:

```
Line 1  require "json"
-
-  # First define these fancy methods...
-  def json_save(object, filename)
5    File.open(filename, "w") do |f|
-      f.write(object.to_json)
-    end
-  end
-
10 def json_load(filename)
-    json_string = File.read(filename)
-
-    JSON.parse(json_string)
-  end
15
-  # ...and now use these fancy methods.
-  test_array = [
-    "Slick Shoes",
-    "Bully Blinders",
20   "Pinchers of Peril",
-  ]
```

```
# Hey, time for some 'me' trivia: In Portland once,
# I met the guy who played Troy's dad. True story.
filename = "DatasGadgets.txt"

json_save(test_array, filename)  # We save it...
read_array = json_load(filename) # We load it...
puts(read_array == test_array)   # We verify it.
```

```
true
```

Isn't that lovely? When you start writing larger programs, you'll find that the task involves more than "writing the program." It also involves building your own tools, and in a way adding to the language, so that you can more easily write the programs you want to write.

Renaming Photos

Let's do something more useful now. A while back, my wife wanted a program to download the photos from her camera's memory card and rename them. Let's build something like that program.

We'll need a few more of Ruby's built-in methods to accomplish this. The first is the Dir[] method. We've seen [] used with arrays before—you did know that was a method, didn't you? Yep, it sure is. You say "arr[2]" and I say "arr.[](2)"—it's all the same.

Anyway, rather than using an array's [] method, we're using the object Dir's [] method. (Dir is for *directory*.) And instead of passing in a number, as with arrays, this time you pass in a string. This isn't just any string; it's a string describing which filenames you're looking for. It then searches for those files and returns an array of the filenames (strings) it found. (For simplicity's sake, I'm going to say "filename" when I actually mean "absolute or relative path and filename.")

The format of the input string is pretty easy. It's simply a filename with a few extra goodies. In fact, if you only pass in a filename, you'll get either an array containing the filename (if the file exists) or an empty array (if it doesn't):

```
Line 1 puts Dir["MeaningOfLife.pdf"]
```

```

```

Hmm, I must have misplaced that one. Well, I'm sure I won't miss it.

Anyway, I could search for all jpgs with Dir["*.jpg"]. Actually, since these are case-sensitive searches, I should probably include the all-caps version as well, Dir["*.{JPG,jpg}"], which roughly means "Find me all files starting with

whatever and ending with a dot and either JPG or jpg." Of course, that searches for jpgs only in the *current working directory*, which (unless you change it in your program) is the directory you ran the program from. To search in the parent directory, you'd want something like Dir["../*.{JPG,jpg}"]. If you wanted to search in the current directory and all subdirectories (a recursive search), you'd want something like Dir["**/*.{JPG,jpg}"]. You can do more things with Dir[]; I don't even know what they all are. This will be enough for us, though.

And remember I said you could change your current working directory? You do that with Dir.chdir; you simply pass in the path to your new working directory.

We'll also be using File.rename. It should be fairly obvious how it works when you see how we use it. I do have one thing to say about renaming, though. According to your computer, moving a file and renaming a file are actually the same task. Often, only one of these is presented as an option. And, if you think about it, this kind of makes sense. If you rename a file from ThingsToWrite/book.txt to ThingsToRead/book.txt, you've moved that file. And if you move a file to the same location, but with a different name, you've renamed it.

The last new method we'll be using is print, which is almost exactly like puts, except it doesn't advance to the next line. I don't use it that often, but it's nice for making little progress bars and things.

Finally, let me tell you a bit about my wife's computer. It's a Windows machine, so the absolute paths are going to be C:/is/for/cook.ie and such. Also, the F:/ drive is the card reader for her camera's memory card. (Yes, I'm using forward slashes. Yes, Windows uses backslashes. Yes, it's perfectly okay to use forward slashes in Ruby, even on Windows. This helps make Ruby programs more cross-platform.)

We're going to move the files to a folder on the hard disk and rename them as we do so. (And since, as we all know, move and rename are the same thing, we'll do that in one command.)

Here it is:

```
Line 1  # For Katy, with love.
-
-  # This is where she stores her photos.
-  # Just for my own convenience, I'll go there now.
5  Dir.chdir("C:/Documents and Settings/Katy/PictureInbox")
-
-  # First we find all of the photos to be moved.
-  pic_names = Dir["F:/**/*.jpg"]
```

```
-
10 puts "What would you like to call this batch?"
- batch_name = gets.chomp
-
- puts
- print "Downloading #{pic_names.length} files: "
15
- # This will be our counter. We'll start at 1 today,
- # though normally I like to count from 0.
- pic_number = 1
-
20 pic_names.each do |name|
-   print "." # This is our "progress bar".
-
-   new_name = if pic_number < 10
-     "#{batch_name}0#{pic_number}.jpg"
25   else
-     "#{batch_name}#{pic_number}.jpg"
-   end
-
-   # This renames the photo, but since 'name'
30   # has a big long path on it, and 'new_name'
-   # doesn't, it also moves the file to the
-   # current working directory, which is now
-   # Katy's PictureInbox folder.
-   # Since it's a *move*, this effectively
35   # downloads and deletes the originals.
-   # And since this is a memory card, not a
-   # hard drive, each of these takes a second
-   # or so; hence, the little dots let her
-   # know that the program isn't hanging.
40   File.rename(name, new_name)
-
-   # Finally, we increment the counter.
-   pic_number = pic_number + 1
- end
45
- puts # This is so we aren't on the progress bar line.
- puts "Done, cutie!"
```

Quite lovely. The full program I wrote for her also downloads the movies, extracts the time and date from the actual .jpg or .avi file, and renames the file using that.

The first time I wrote this program, I forgot the part that increments the counter, on line 43. Small, simple mistake. What do you think happened?

This helpful program helpfully copied every photo to the same new filename…*over* the previous photo. This effectively deleted every photo except for the last one to be copied. Good thing I always, always, *always* make

backups when I'm testing a program like this. Because, you know, the thing about computers....

A Few Things to Try

- *Safer photo downloading.* Adapt the photo-downloading/file-renaming program to your computer, and add some safety features to make sure you never overwrite a file. One method you might find useful is `File.exist?` (pass it a filename, and it'll return `true` or `false`). Another method is `exit` which is like an evolved form of `return`—it kills your program right where it stands. This is good for spitting out an error message and then quitting.

Progress Checkpoint

In this chapter, you learned how to accidentally delete files. Ideally, you also learned how *not* to. You also learned about JSON, double-quoted strings, and a few things you can do with the `Dir` object.

Next, you'll learn about some new classes of objects, which will level up your programming skills much like arrays did.

CHAPTER 10

More Classes and Methods

So far, you've seen several kinds (or *classes*) of objects: strings, integers, floats, arrays, a few special objects (true, false, and nil), and so on. In Ruby, these class names are always capitalized: String, Integer, Float, Array, File, and Dir. (Do you remember back on page 71 when you asked the File class to open a file for you, and it handed back an actual file, which you named, in a fit of rabid creativity, f? Good times. And while you never ended up needing an actual directory object from Dir, you could have gotten one if you'd asked nicely.)

You used File.open to get a file back, but that's actually a slightly unusual way to get an object from a class. In general, you'll use the new method:

```
Line 1 alpha = Array.new  + [12345]  # Create empty array.
   2 beta  = String.new + "hello"  # Create empty string.
   3 karma = Time.new              # Current date and time.
   4
   5 puts "alpha = #{alpha}"
   6 puts "beta  = #{beta}"
   7 puts "karma = #{karma}"
```

```
alpha = [12345]
beta  = hello
karma = 2020-10-10 11:56:25 -0700
```

Because you can make arrays with array *literals* using [...], and you can make strings with string literals using "...", you rarely create these using new. Also, numbers are special exceptions: you can't create an integer with Integer.new. (Which number would it create, you know?) You can make one only using an integer literal (that is, by typing it out as you've been doing). But aside from a few exceptions, you normally create Ruby objects with new.

One way you could think of this is that a class is kind of like a cookie cutter: every time you call new on that class, you get a new cookie that's defined by

the shape of that cookie cutter. You'd never try to take a bite out of a cookie cutter, because a cookie cutter isn't a cookie. Similarly, a class is a *different kind of thing* from the objects it creates. (The String class isn't a string. It's a class, a sort of programmatic cookie cutter.)

Let's take a tour of some other frequently-used classes, and learn what you can do with the objects they create.

The Time Class

What's the story with this Time class? Time objects represent (you guessed it) moments in time. You can add (or subtract) numbers to (or from) times to get new times. So, adding 1.5 to a time object makes a new time one-and-a-half seconds later:

```
Line 1 time = Time.new  # The moment this code was run.
   2 soon = time + 60 # One minute later.
   3
   4 puts time
   5 puts soon
```

```
❮ 2020-10-10 11:57:23 -0700
  2020-10-10 11:58:23 -0700
```

You can also pass values into Time.new to construct a specific time:

```
Line 1 puts Time.new(2000, 1, 1)         # Y2K.
   2 puts Time.new(1976, 8, 3, 13, 31) # When I was born.
```

```
❮ 2000-01-01 00:00:00 -0800
  1976-08-03 13:31:00 -0700
```

You'll notice the -0700 and -0800 in these times. That's to account for the difference between the local time and Coordinated Universal Time, or UTC. The difference between local time and UTC could be due to time zone or daylight saving time or who knows what else. So, you can see that I was born in daylight saving time, while it was *not* daylight saving time when Y2K struck. The more values you pass in to new, the more specific your time becomes.

On the other hand, if you want to avoid time zones and daylight saving time altogether and only use UTC, there's always Time.utc:

```
Line 1 puts Time.utc(1955, 11, 5) # A red-letter day.
```

```
❮ 1955-11-05 00:00:00 UTC
```

You can compare times using the comparison methods (an earlier time is *less than* a later time), and if you subtract one time from another, you'll get the number of seconds between them. Play around with it.

If you happen to be using an older version of Ruby, there's this problem with the Time class. It thinks the world began at *epoch*: the stroke of midnight, January 1, 1970, UTC. I don't know of any satisfying way of explaining this, but here goes: at some point, probably before I was even born, some people (Unix folks, I believe) decided that a good way to represent time on computers was to count the number of seconds since the very beginning of the 70s. So, time "zero" stood for the birth of that great decade, and they called it *epoch*.

This was all long before Ruby. In those ancient days (and programming in those ancient languages), you often had to worry about your numbers getting too large. In general, a number would either be from 0 to around 4 billion or be from -2 billion to +2 billion, depending on how they chose to store it.

For whatever reasons (compatibility, tradition, cruelty…whatever), older versions of Ruby decided to go with these conventions. So (and this is the important point), you *couldn't have times more than 2 billion seconds away from epoch*. This restriction wasn't too painful, though, because this span is from sometime in December 1901 to sometime in January 2038.

In modern Ruby (versions 1.9.2 and up), you don't need to worry about any of this. Time can handle any time that has any meaning, from before the Big Bang to after the heat death of the universe (or however this grand tale ends).

Here are some exercises for you to get more familiar with Time.

A Few Things to Try

- *One billion seconds!* Find out the exact second you were born, or as close as you can get. Figure out when you'll turn (or perhaps when you did turn) one billion seconds old. Then go mark your calendar!
- *Happy birthday!* Ask what year a person was born, then the month, and then the day. Figure out how old they are, and give them a ☺ (smiley emoji) for each birthday they've had.

The Hash Class

Another useful class is the Hash class. Hashes are a lot like arrays: they have a bunch of slots that can point to various objects. But in an array, the slots are lined up in a row, and each one is numbered (starting from zero). In a hash, the slots aren't in a row (they're sort of jumbled together), and you can use *any* object to refer to a slot, not only a number. It's good to use hashes when you have a bunch of things you want to keep track of, but

they don't fit into an ordered list. For example, you could store European capitals like this:

```
caps_array = [] # array literal, same as Array.new
caps_hash  = {} # hash literal,  same as Hash.new

caps_array[0]       = "Oslo"
caps_array[1]       = "Paris"
caps_array[2]       = "Madrid"
caps_array[3]       = "Rome"
caps_hash["Norway"] = "Oslo"
caps_hash["France"] = "Paris"
caps_hash["Spain" ] = "Madrid"
caps_hash["Italy" ] = "Rome"

caps_array.each do |city|
  puts city
end

caps_hash.each do |country, city|
  puts "#{country}: #{city}"
end
```

```
Oslo
Paris
Madrid
Rome
Norway: Oslo
France: Paris
Spain: Madrid
Italy: Rome
```

If you use an array, you have to remember that slot `0` is for `"Oslo"`, slot `1` is for `"Paris"`, and so on. But if you use a hash, it's easy. Slot `"Norway"` holds the name `"Oslo"`, of course. There's nothing to remember. You might have noticed that when you used `each`, the objects in the hash came out in the same order you put them in. Older versions of Ruby (1.8 and earlier) didn't work this way, so if you have an older version installed, be aware of this (or better yet, simply upgrade).

That last example included an empty hash literal, and populated it with European capitals one at a time. But as with arrays, you can populate a hash using only the literal:

```
greetings = ["hello", "howdy", "hi"]               # array literal
smoothies = {"mango" => "yum", "garlic" => "yuck"} # hash literal

puts greetings
puts smoothies
puts smoothies["mango"]
```

```
❮ hello
howdy
hi
{"mango"=>"yum", "garlic"=>"yuck"}
yum
```

Though people usually use strings to name the slots in a hash, you could use any kind of object, even arrays and other hashes:

```
Line 1 weird_hash = Hash.new
   2
   3 weird_hash[12] = "monkeys"
   4 weird_hash[[]] = "the void"
   5 weird_hash[Time.new] = "no time like the present"
```

Hashes and arrays are good for different things. It's up to you to decide which one is best for a particular problem. Are you storing things that are sequential or ordered? Then an array makes most sense. Is it more like a collection of different things? Then consider using a hash. In general, arrays are better for collections of *the same kind of thing*, while hashes are better for collections of *different kinds of things*:

```
Line 1 myself = {"name" => "Chris", "pairs_of_shoes" => 17}
   2 imelda = {"name" => "Imelda", "pairs_of_shoes" => 3400}
   3
   4 people = [myself, imelda]
```

Names and numbers of pairs of shoes aren't the same kind of thing at all, so we store those together in a hash. But those two hashes are both representing people (at least a little bit about them), so we use an array to group the people-hashes together.

I probably use hashes at least as often as arrays. They're wonderful.

The Range Class

`Range` is another great class. Ranges represent intervals of numbers. Usually. (You can also make ranges of strings, times, or pretty much anything else you can place in an order—where you can say things like `this < that` and such. In practice, I never use ranges for anything but integers.)

Let's make some range literals and see what they can do:

```
Line 1 # This is a range literal.
   - numbers = 1..5
   -
   - # Convert range to array.
   5 puts([1, 2, 3, 4, 5] == numbers.to_a)
   -
   - # Iterate over a range of strings:
```

```
("a".."z").each do |letter|
  print letter
end
puts

god_bless_the_90s = 1990..1999
puts god_bless_the_90s.min
puts god_bless_the_90s.max
puts(god_bless_the_90s.include?(1999  ))
puts(god_bless_the_90s.include?(2000  ))
puts(god_bless_the_90s.include?(1994.5))
```

```
true
abcdefghijklmnopqrstuvwxyz
1990
1999
true
false
true
```

Do you really need ranges? No, not really. You could write programs without them. It's the same with hashes and times, I suppose. But it's all about style, intention, and capturing snapshots of your brain right there in your code.

And this is all just the tip of the iceberg. Each of these classes has way more methods than covered here, and this isn't even a tenth of the classes that come with Ruby. But you don't *need* most of them. They are simply time-savers. You can pick them up gradually as you go. That's how most of us do it.

Now that you've learned about `Time`, `Hash`, and `Range`, let's revisit `String`. I'd feel like I was doing you a disservice if we didn't look a little more at what you can do with strings. Plus, if we do, I can give you more interesting exercises. Mind you, we're still not going to cover even half of the string methods, but you've *got* to see a little more.

Stringy Superpowers

Remember back in More Array Methods, on page 51 when I said a lot of the string methods also work on arrays? Well, it goes both ways: some of the array methods you've learned also work on strings.

Perhaps the most important and versatile is the `[...]` method. The first thing you can do with it is pass in a number and get the character at that position in the string:

```
Line 1  bosco = "Mr. T"
   2  big_T = bosco[4]
   3  puts big_T
```

```
❮  T
```

And then you can do fun stuff like this:

```
Line 1  puts "Hello. My name is Apollo."
   -  puts "I'm extremely perceptive."
   -  puts "What's your name?"
   -
   5  name = gets.chomp
   -  puts "Hi, #{name}."
   -
   -  if name[0] == "C"
   -    puts "You have excellent taste in footwear."
  10    puts "I can just tell."
   -  end
```

```
❮  Hello. My name is Apollo.
   I'm extremely perceptive.
   What's your name?
⇛  Chris
❮  Hi, Chris.
   You have excellent taste in footwear.
   I can just tell.
```

This is only the beginning of the [...] method. Instead of picking out only one character, you can pick out substrings in two different ways. One way is to pass in two numbers: the first tells you where to start the substring, and the second tells you how long of a substring you're looking for.

The second way, though, is to pass in a range.

And both of these ways have a little twist. If you pass in a negative index, it counts from the *end* of the string.

```
Line 1  prof = "We tore the universe a new space-hole, alright!"
   -  puts prof[12, 8]
   -  puts prof[12..19] # 8 characters total
   -  puts
   5
   -  def is_avi?(filename)
   -    filename.downcase[-4..-1] == ".avi"
   -  end
   -
  10  # Vicarious embarrassment.
   -  puts is_avi?("DANCEMONKEYBOY.AVI")
   -
   -  # Hey, I wasn't even 2 years old at the time...
   -  puts is_avi?("toilet_paper_fiasco.jpg")
```

```
universe
universe

true
false
```

Let all that awesomeness sink in for a bit, and then cement it with some exercises.

A Few Things to Try

- *Party like it's roman_to_integer("mcmxcix")!* Come on, you knew it was coming, didn't you? It's the other half of your roman_numeral(1999) method. Make sure to reject strings that aren't valid Roman numerals.
- *Birthday helper!* Write a program to read in names and birth dates from a text file. It should then ask you for a name. You type one in, and it tells you when that person's next birthday will be (and, for the truly adventurous, how old they'll be). The input file should look something like this:

  ```
  Chris Hemsworth,  Aug 11, 1983
  Chris Evans,      Jun 13, 1981
  Chris Pratt,      Jun 21, 1979
  Chris Pine,       Aug 26, 1980
  Other Chris Pine, Aug  3, 1976
  ```

  Lot of summer birthdays! Must be prime Chris season. (Also, it appears that I'm the oldest of the Five Famous Chrises, which in no way bothers me at all in the slightest.) Anyway, for this exercise, you'll probably want to break each line up and put it in a hash, using the name as your key and the date as your value. In other words:

```
Line 1 birth_dates["Chris Pratt"] = "Jun 21, 1979"
```

  (You can store the dates in some other format if you prefer.)

  Though you can do it without this tip, your program might look prettier if you use the each_line method for strings. It works pretty much like each does for arrays, but it returns each line of the multiline string one at a time (but with the line endings, so you'll need to chomp them). I thought I'd mention that.

Classes and the Class Class

I'll warn you right now: this section is a bit of a brain bender. So, if you're not feeling particularly strong of stomach, you can skip to the next chapter. At least for now, this is mainly of academic interest. But in case you were wondering…

You may have noticed that you can call methods on strings (things such as length and chomp), but you can also call methods (such as new) on the actual String. This is because, in Ruby, classes are real objects. And since every object is in some class, classes must be, too. You can find the class of an object using the class method:

```
Line 1  puts(42.class)
    2  puts("I'll have mayonnaise on mine!".class)
    3  puts(Time.new.class) # No shocker here.
    4  puts(Time.class)     # A little more interesting...
    5  puts(String.class)   # Yeah, OK...
    6
    7  # Drum roll please...
    8  puts(Class.class)
    9  # <gasp!>
```

```
Integer
String
Time
Class
Class
Class
```

If this makes sense to you right now, then *stop thinking about it.* You might screw it up! Otherwise…don't sweat it too much. Move on, and let your subconscious do the work later.

Progress Checkpoint

In this chapter you learned some new classes: Time, Hash, and Range. You also learned more about strings, and you had your first encounter with the mysterious Class class.

In the next chapter, you'll learn how to create new classes of your own.

CHAPTER 11

Custom Classes and Class Extensions

Back when you were writing your own methods on page 66, you wrote a method to take an integer and return the string of the Roman numerals for that integer. It wasn't an integer method, though; it was simply a generic "program" method. Wouldn't it be nice if you could write something like `42.to_roman` instead of `roman_numeral(42)`? Well, it turns out that you *can* create new integer methods. (It would have been insensitive of me to bring it up otherwise.) Here's how:

```
Line 1  class Integer
        def to_roman
          if self == 5
            roman = "V"
        else
            roman = "XLII"
          end

          roman
        end
      end

      # I'd better test on a couple of numbers...
      puts 5.to_roman
      puts 42.to_roman
```

```
V
XLII
```

Good news, everyone! It seems to work. ☺

This example defines an integer method by jumping into the `Integer` class, defining the method there, and jumping back out. Now all integers have this (somewhat incomplete) method. In fact, you can do this with any method in any class, even the built-in methods. So, if you don't like the `reverse` method for strings, you can redefine it in much the same way. But I don't recommend

it. It's best to leave the old methods alone and to make new ones when you want to do something new.

In case that was confusing, let's go over that last program in a bit more detail. So far, whenever you executed any code or defined any methods, you did it in the default "program" object. In your last program, you left that object for the first time and hopped into the Integer class. You defined a method there (which makes it an integer method), and now all integers can use it. Inside that method you use self to refer to the object (the integer) which is "doing" the method.

Now you try it.

A Few Things to Try

- *Extend the built-in classes.* How about making an array method named mult that multiplies all of the elements of the array together? So you could write [2,5,3].mult, and it would return to 30. Or how about making double an integer method? Or maybe a string method: "Austin".leet returns "/\Ü571N" or whatever? In each case, remember to use self to access the object the method is being called on (the "Austin" in "Austin".leet).

Creating Classes

You've now seen a smattering of different classes. But it's easy to come up with all kinds of objects that Ruby doesn't have—objects you'd like it to have. Fear not; creating a new class is as easy as extending an old one. Let's say you wanted to make some dice in Ruby, for example. Here's how you could make the Die class:

```
Line 1  class Die
        -   def roll
        -     1 + rand(6)
        -   end
      5 end
        -
        - # Let's make a couple of dice...
        - dice = [Die.new, Die.new]
        -
     10 # ...and roll them.
        - dice.each do |die|
        -   puts die.roll
        - end

      ❮ 1
        3
```

And that's it! These are objects of your very own. Roll the dice a few times (run the program again), and watch what turns up. Oh, and if you skipped the section on random numbers, `rand(6)` returns a random number between `0` and `5`.

You can define all sorts of methods for your objects…but there's something missing. Working with these objects feels a lot like programming before you learned about variables. Look at your dice, for example. You can roll them, and each time you do they give you a different number. But if you wanted to hang onto that number, you'd have to create a variable to point to the number. It seems like any decent die should be able to *have* a number and that rolling the die should change that number. If you keep track of the die, you shouldn't also have to keep track of the number it's showing.

But if you try to store the number you rolled in a (local) variable in `roll`, it'll be gone as soon as `roll` is finished. You need to store the number in a different kind of variable, an *instance variable*. The next section shows you how.

Instance Variables

Normally, when you want to talk about a string, you simply call it a *string*. But you could also call it a *string object*. Sometimes programmers might call it an *instance* of the class `String`, but that's only another way of saying *string*. An *instance* of a class is an object of that class.

So, instance variables are an object's variables. A method's local variables last until the method is finished. An object's instance variables, on the other hand, will last as long as the object does.

To tell instance variables from local variables, they have `@` in front of their names, like this:

```
Line 1 class Die
  -   def roll
  -     @number_showing = 1 + rand(6)
  -   end
  5
  -   def showing
  -     @number_showing
  -   end
  - end
 10
  - die = Die.new
  - die.roll
  - puts die.showing
  - puts die.showing
 15 die.roll
```

```
puts die.showing
puts die.showing
```

```
2
2
3
3
```

Very nice. The method `roll` rolls the die, and `showing` tells you which number is showing. But what if you try to look at what's showing before you've rolled the die (before you've set `@number_showing`)? Let's try it:

```
class Die
  def roll
    @number_showing = 1 + rand(6)
  end

  def showing
    @number_showing
  end
end
# Since I'm not going to use this die again,
# I don't need to save it in a variable.
puts Die.new.showing
```

```

```

Hmmm…looks like a `nil`. Which makes sense, since you didn't set `@number_showing` to anything yet. At least it didn't give you an error. Still, it doesn't make sense for a die to be "unrolled," or whatever `nil` is supposed to mean here. It would be nice if you could set up your new `Die` object right when it's created. That's what the `initialize` method is for; as soon as an object is created, `initialize` is automatically called on it (if you have defined it). It looks like any other method, except that it's called `initialize`.

Here's an example of it in use:

```
class Die
  def initialize
    # I'll just roll the die, though we could do something else
    # if we wanted to, such as setting the die to have 6 showing.
    roll
  end

  def roll
    @number_showing = 1 + rand(6)
  end

  def showing
    @number_showing
  end
```

```
15 end
-
- puts Die.new.showing
❮ 4
```

Excellent, no more nil value. One thing to note here: in that example, you're first defining what the Die class is by defining the methods initialize, roll, and showing. But none of these methods is actually called until the very last line. At that point, you create a new die, which immediately calls initialize, and initialize calls roll. Then Die.new returns your new die object, on which you call showing.

Cool. Your dice are nearly perfect. The only feature that might be missing is a way to set which side of a die is showing. Why don't you write a cheat method that does exactly that? Come back when you're done, and when you tested that it worked, of course. Make sure that someone can't set the die to have a 7 showing, or 3.5; you're cheating, not bending the laws of nature.

Hopefully that all made sense. One thing that can confuse people at first (myself included) is the relationship between new and initialize. Let's spend some time clearing that up in the next section.

Methods: new vs. initialize

We covered some pretty cool stuff in the previous section. But the relationship between new and initialize is a bit subtle. And "subtle" may as well mean "confusing." What's the deal here?

The methods new and initialize work hand in hand. You use new to create a new object, and initialize is then called automatically (if you defined it in your class). They pretty much happen at the same time. How do you keep them straight?

First, new is a method of the *class*, while initialize is a method of the *instance*. You call new to create the instance, and then initialize is automatically called on that instance. This means that the call to new must come first. Until you call new, there's no instance to call initialize upon.

Second, you define initialize in your class, but you never define new. (It's already built into all classes.) Conversely, you call new to create an object, but you *never* call initialize. The method new takes care of that for you.

Strictly speaking, it's possible to call initialize, and it's possible to define new. But doing so is either very advanced or very foolish. Or both? Either way, this isn't the right book to cover such topics.

The reason for having these two methods is that you need one of them to be a class method and the other to be an instance method. If you think about

it, new has to be a class method, because when you want to create an object, the object you want *doesn't exist yet*. You can't say die.new, for example, because die doesn't exist yet.

And initialize has to be an instance method because you're initializing *that object*. So, you need access to the instance variables and such. You can't do that from a class method because it wouldn't know which instance to get the instance variables from. (You certainly don't want to initialize every single instance of Die every time you create a new one.)

So remember, you *define* the *instance* method initialize, and you *call* the *class* method new, and never the other way around.

Now that you've learned about creating a new class, creating methods and instance variables for objects of that class, and how new and initialize work, it's time to put all of that learning into another, larger example.

The Care and Feeding of Your Baby Dragon

You know how to create your own classes, even some of the subtle bits, but so far you've only seen a small, fluffy, toy example. Let me give you something a bit more interesting. Let's say we want to make a simple virtual pet, a baby dragon.

Like most babies, it should be able to eat, sleep, and poop, which means you'll need to be able to feed it, put it to bed, and take it on walks. Internally, your dragon will need to keep track of whether it's hungry, tired, or needs to go out, but you won't be able to see that when you interact with your dragon, the same way you can't ask a human baby, "Are you hungry?" You'll also add a few other fun ways you can interact with your baby dragon, and when your baby dragon is born, you'll give it a name. (Whatever you pass into the new method is then passed onto the initialize method for you.)

Start by defining the class and the initialize method:

baby_dragon.rb

```
class Dragon
  def initialize(name)
    @name               = name
    @asleep             = false
    @stuff_in_belly     = 10 # baby is full
    @stuff_in_intestine =  0 # baby doesn't need to go

    puts "#{@name} is born."
  end
end
```

Now you've defined the class, the internal state of your baby dragon, and your dragon's name.

Next, you want to define your *public interface*, which is the set of methods that other code is allowed to call on your baby dragon. In other words, these are the things you can do with your dragon.

Each of these methods will also call another method, passage_of_time, which handles things like waking up a hungry baby and moving stuff from belly to intestine. (You'll define passage_of_time in a bit.)

Add these methods to your Dragon class, after initialize:

baby_dragon.rb

```
def feed
  puts "You feed #{@name}."
  @stuff_in_belly = 10
  passage_of_time
end

def walk
  puts "You walk #{@name}."
  @stuff_in_intestine = 0
  passage_of_time
end

def put_to_bed
  puts "You put #{@name} to bed."
  @asleep = true
  3.times do
    if @asleep
      passage_of_time
    end
    # since passage_of_time might wake up the baby,
    # check to see if they are still asleep
    if @asleep
      puts "#{@name} snores, filling the room with smoke."
    end
  end
  if @asleep
    @asleep = false
    puts "#{@name} wakes up slowly."
  end
end

def toss
  puts "You toss #{@name} up into the air."
  puts "#{@name} giggles, which singes your eyebrows."
  passage_of_time
end
```

```
def rock
  puts "You rock #{@name} gently."
  @asleep = true
  puts "#{@name} briefly dozes off..."
  passage_of_time
  if @asleep
    @asleep = false
    puts "...but wakes when you stop."
  end
end
```

Now that you have your public interface, it's time to define your private methods: methods that cannot be called externally, but that your public methods can call. One of these is passage_of_time, and you also want a few others. To make these methods private, you precede them with the keyword private. Add this code to your Dragon class:

baby_dragon.rb

```
private
# "private" means that the methods defined here are
# methods internal to the object. (You can feed your
# dragon, but you can't ask them whether they're hungry.)
def hungry?
  # Method names can end with "?".
  # Usually, we do this only if the method
  # returns true or false, like this:
  @stuff_in_belly <= 2
end

def poopy?
  @stuff_in_intestine >= 8
end

def passage_of_time
  if @stuff_in_belly > 0
    # Move food from belly to intestine.
    @stuff_in_belly     = @stuff_in_belly     - 1
    @stuff_in_intestine = @stuff_in_intestine + 1
  else #  Our dragon is starving!
    if @asleep
      @asleep = false
      puts "#{@name} wakes up suddenly!"
    end
    puts "#{@name} is starving! In desperation, #{@name} ate YOU!"
    exit # This terminates the program.
  end

  if @stuff_in_intestine >= 10
    @stuff_in_intestine = 0
    puts "Whoops! #{@name} had an accident..."
  end
```

```
  if hungry?
    if @asleep
      @asleep = false
      puts "#{@name} wakes up suddenly!"
    end
    puts "#{@name}'s stomach grumbles..."
  end

  if poopy?
    if @asleep
      @asleep = false
      puts "#{@name} wakes up suddenly!"
    end
    puts "#{@name} does the potty dance..."
  end
end
```

That completes your class. Now to see it in action:

baby_dragon.rb

```
# Now that we've described what a dragon is,
# let's actually create one.
pet = Dragon.new("Norbert")
pet.feed
pet.toss
pet.walk
pet.put_to_bed
pet.rock
pet.put_to_bed
pet.put_to_bed
pet.put_to_bed
pet.put_to_bed
```

```
Norbert is born.
You feed Norbert.
You toss Norbert up into the air.
Norbert giggles, which singes your eyebrows.
You walk Norbert.
You put Norbert to bed.
Norbert snores, filling the room with smoke.
Norbert snores, filling the room with smoke.
Norbert snores, filling the room with smoke.
Norbert wakes up slowly.
You rock Norbert gently.
Norbert briefly dozes off...
...but wakes when you stop.
You put Norbert to bed.
Norbert wakes up suddenly!
Norbert's stomach grumbles...
You put Norbert to bed.
Norbert wakes up suddenly!
Norbert's stomach grumbles...
```

```
You put Norbert to bed.
Norbert wakes up suddenly!
Norbert's stomach grumbles...
Norbert does the potty dance...
You put Norbert to bed.
Norbert wakes up suddenly!
Norbert is starving! In desperation, Norbert ate YOU!
```

Well, that's what you get when you neglect your baby dragon.

Note that you could have left out the private keyword and made all of your methods public. But it's good, clean design to distinguish between things you can do to a dragon and things that are internal mechanisms of the dragon. You can think of these as being "under the hood": unless you're an automobile mechanic, all you need to know is the gas pedal, the brake pedal, and the steering wheel. These are the public interface of your car. How your airbag knows when to deploy, on the other hand, is internal to the car; the typical user (driver) doesn't need to know how that works. (And keeping it hidden means the user can't accidentally deploy the airbag when turning up the radio. This is good design.)

Actually, for a bit more concrete example along those lines, let's talk about how you might represent a car in a video game.

First, you'd need to decide how you want your public interface to work: which methods should people be able to call on one of your car objects? Well, people need to be able to push the gas pedal and the brake pedal, but they'd also need to be able to specify how hard they're pushing the pedal. (There's a big difference between flooring it and tapping it.) They'd also need to be able to steer, and again, they'd need to be able to indicate how hard they're turning the wheel. I suppose you could go further and add a clutch, turn signals, rocket launcher, afterburner, flux capacitor, and so. It depends on what type of game you're making.

Internal to a car object, though, much more would need to be going on; other things a car would need are a velocity, an orientation, and a position. These attributes would be modified by pressing on the gas or brake pedals and turning the wheel, of course, but the user wouldn't be able to set the position directly (which would be like warping). You might also want to keep track of skidding, damage, how many jumps you've made, and so on. These would all be internal to your car object—that is, not directly accessible by the player; these would be private.

Now that you've seen another, bigger example of creating and interacting with a class, it's time for you to make your own.

A Few Things to Try

- *Orange tree.* Make an OrangeTree class that has a height method that returns its height and a one_year_passes method that, when called, ages the tree one year. Each year the tree grows taller (however much you think an orange tree should grow in a year), and after some number of years (again, your call) the tree should die. For the first few years, it shouldn't produce fruit, but after a while it should, and I guess that older trees produce more each year than younger trees…whatever you think makes the most sense. And, of course, you should be able to count_the_oranges (which returns the number of oranges on the tree) and pick_an_orange (which reduces the @orange_count by one and returns a string telling you how delicious the orange was, or else it tells you that there are no more oranges to pick this year). Make sure any oranges you don't pick one year fall off before the next year.
- *Interactive baby dragon.* Write a program that lets you enter commands such as feed and walk and calls those methods on your dragon. Of course, since you're inputting just strings, you'll need some sort of *method dispatch*, where your program checks which string was entered and then calls the appropriate method.

Progress Checkpoint

In this chapter, you learned how to extend built-in classes with additional methods (like adding another string method or integer method), and how to change existing methods (by simply defining a method of the same name as the one you want to change). You then learned how to make your own classes, complete with instance variables and an initialize method. Finally, you learned how to hide some methods from the rest of your code by making them private.

Now that you've had a chance to practice by creating some classes of your own, you know even more Ruby. In the next chapter, you'll turn your gaze outward, and learn how to use Ruby (and JSON) to talk to other programs over the Internet.

CHAPTER 12

Remote Data and APIs

This is, in my humble opinion, the most exciting chapter of the book. This chapter is where your code touches the Internet, where your programs learn to talk to other programs, and where you get your first glimpse of what it's like to build cool things using the data and tools that others have provided—this is your chance to play with APIs.

API is an acronym for *Application Programming Interface*, which roughly means "a way for programs to talk to each other." In this chapter, you'll learn how to use web APIs. A web API is the way in which a web server talks with other programs, usually using JSON.

Let's start with writing a program that fetches jokes from the Internet.

Random Internet Jokes

The Official Joke API (which certainly sounds official to me) is about as simple as an API can get: it's a URL that returns a bit of JSON. In fact, you can try it out in your browser if you want.[1] You'll see that this API returns a small JSON blob that includes a joke stuffed inside—*somewhere*.

Believe it or not, *blob* is actually a technical term. It originally was an acronym for Binary Large OBject but over time came to mean a clump of data, not necessarily large or small, so saying "a small JSON blob" is not that weird.

Anyway, the first step is to have your program fetch the JSON blob, parse it, and turn it into a Ruby object. For this to work, you'll need to require the `net/http` library for the fetching, and the `json` library for the parsing.

1. https://official-joke-api.appspot.com/random_joke

This API returns a JSON object (as opposed to a JSON array), which the parser uses to construct a Ruby hash (as opposed to a Ruby array). Most APIs return JSON objects.

What does the JSON object that this API returns look like? What kind of data does it hold? Let's run this code to find out:

```
Line 1  require "net/http"
2  require "json"
3
4  url = URI("https://official-joke-api.appspot.com/random_joke")
5
6  json_response = Net::HTTP.get(url)
7
8  hash_response = JSON.parse(json_response)
9
10 puts hash_response
```

```
{"id"=>282, "type"=>"general", "setup"=>"What's red and bad for your teeth?",
"punchline"=>"A Brick."}
```

No, that's not a typo on line 4. Ruby calls that a URI, but most people call it a URL. What's the difference between a URL and a URI? I'll be honest with you, I had to look it up so that I can sagely tell you the difference, as if I knew. But then I thought, who cares? If I still don't know the difference between a URL and a URI after over two decades of programming, maybe it's just not that important. All you need to know is that Ruby calls them URIs, while most people call them URLs.

Moving on, you can see that this response includes an "id", which looks like a unique number assigned to each joke. It also includes a "type" which might mean there are different kinds of jokes? It's hard to tell from only one response. Finally, you reach the important bits: the "setup" and the "punchline".

From this response, you can figure out how to write a proper joke program. After fetching and parsing the response, you print the "setup", followed by the "punchline":

```
Line 1  require "net/http"
-  require "json"
-
-  url = URI("https://official-joke-api.appspot.com/random_joke")
5
-  response = Net::HTTP.get(url)
-
-  joke = JSON.parse(response)
-
10 puts joke["setup"]
-  gets
-  puts joke["punchline"]
```

```
❮ What do you call an eagle who can play the piano?
⇛
❮ Talonted!
```

Ouch. These jokes are *painful*. (But at least they're official.) Still, how cool is this? You can write code to fetch jokes from an API running on some totally different server.

You explored this API simply by poking at it. Which is fun, and you can do that if you want. Usually, though, it's best to read the documentation.[2] In this case, you can see from the docs that you can use other *endpoints*, meaning other URLs for making different sorts of requests.

From that page, you can see that David Katz provides this API (thanks, David!). David also put all of the code for this API on GitHub. Writing a server and providing an API is outside the scope of this book, but looking at the code there can at least give you an idea of what that might look like.

It was nice of David Katz to set up this free API for us all to use. Others have also created plenty of free APIs. In return, those who use them should do so respectfully, and a big part of that is to not overuse them.

Respect

With the Official Joke API, you can get jokes for free. And because they're all made up of bits that are easily copied, when you get a joke, you aren't taking that joke away from someone else.

Except everything I just said is a lie.

First, it's not free. It's *extremely* cheap. Sure, it's *free for you*, but someone, somewhere, is *paying* for the server that's running the Official Joke API. Although requesting a single joke is *extremely* cheap, it's also *extremely* easy to write a loop that fetches one billion jokes (or continuously fetches as many jokes as it can, forever). And someone would have to pay for that. If that got too expensive, then everyone would lose the Official Joke API.

Second, because of this, and because of the limits of how fast a server can serve up jokes, there actually is a finite number of jokes. If you request too many in too short a time, it could prevent some other program from fetching a joke.

Use free APIs respectfully. If you're using a free API to make money, reach out to the creator to agree upon usage limits and compensation. Free APIs

2. https://github.com/15Dkatz/official_joke_api

are a wonderful resource, and a gift to you from other awesome programmers. Maybe reach out and say thanks.

And only take what you need.

Alright, now that you know the importance of being a good citizen in the world-wide community of programmers, let's check out another free API. Instead of jokes, this one serves up trivia questions.

Trivia Database

Our next fun, free API is the Open Trivia Database. Start by heading to the API page.[3] Instead of documentation, you fill out a form that's used to create the endpoint URL for you, which is pretty cool. Let's explore this one much like we did with the Official (so official) Joke API.

Start by asking for two questions. The form gives you back a URL, which you use to fetch the questions using some familiar-looking code:

```
Line 1  require "net/http"
2  require "json"
3
4  url = URI("https://opentdb.com/api.php?amount=2")
5
6  response = Net::HTTP.get(url)
7
8  trivia = JSON.parse(response)
9
10 puts trivia
```

```
{"response_code"=>0, "results"=>[{"category"=>"Science: Mathematics",
"type"=>"boolean", "difficulty"=>"medium", "question"=>"E = MC3",
"correct_answer"=>"False", "incorrect_answers"=>["True"]},
{"category"=>"Entertainment: Music", "type"=>"multiple",
"difficulty"=>"easy", "question"=>"Sting, the lead vocalist of The Police,
primarily plays what instrument?", "correct_answer"=>"Bass Guitar",
"incorrect_answers"=>["Drums", "Guitar", "Keyboards"]}]}
```

Okay, that's a little difficult to read. Copy that into a text file and clean it up a bit:

```
{
  "response_code" => 0,
  "results"       => [
    {
      "category"          => "Science: Mathematics",
      "type"              => "boolean",
      "difficulty"        => "medium",
```

3. https://opentdb.com/api_config.php

```
      "question"          => "E = MC3",
      "correct_answer"    => "False",
      "incorrect_answers" => ["True"]
    },
    {
      "category"          => "Entertainment: Music",
      "type"              => "multiple",
      "difficulty"        => "easy",
      "question"          => "Sting, the lead vocalist of The Police,
                              primarily plays what instrument?",
      "correct_answer"    => "Bass Guitar",
      "incorrect_answers" => ["Drums", "Guitar", "Keyboards"]
    }
  ]
}
```

Ah, much better!

So, the API returns an object with a response code and a results array. The response code is zero here, which probably means "success". This is an older convention used by some programmers, where a return value of zero means "success" and any other value means "failure". (This was convenient because the return value could be an error code indicating the kind of error, like if Error 72 meant "out of memory" and Error 29 meant "out of mayonnaise" or whatever. Even more convenient than an error code is an actual error *message*, but some folks still use error codes.)

You should test my theory about the response code by sending up a request for something stupid, like mayonnaise:

```
url = URI("https://opentdb.com/api.php?mayo=4")
```

```
❮ {"response_code"=>2, "results"=>[]}
```

"I would like *four of mayo*, please." Yep, looks like an error code. Response code 2 probably means "bad request" or "unknown parameter" or "unreasonable amount of mayonnaise." (Note that, while your request failed in the sense that you got an error code and no results, Ruby didn't give you an error. As far as Ruby can tell, everything worked fine: you made an API call, and got back some JSON. You have to check that JSON yourself to see if there was an error.)

After the response code, comes the results array, where each result is a trivia question. Now you can see the structure of the results: category, type, difficulty, question, and right and wrong answers. Notice how the structure is the same for true/false questions as it is for multiple choice questions. This is a nice API, and it should be pretty easy to use. If you were to write a trivia

program with this data (which you totally should) you ought to be able to treat each question more or less the same.

What do you say? Want to give it a try?

A Few Things to Try

- *Trivia program.* Use the API for the Open Trivia Database to fetch trivia questions, and then ask the questions to the user. Count how many they get right, and tell them how they did at the end. This part won't be too different from the ULTIMATE FLAVOR TOURNAMENT code you wrote in Chapter 8, Custom Methods, on page 55.

  Since the Open Trivia folks went to the trouble of sending a response code, it's probably smart to check that it's zero before you use the rest of the results. (If it's not zero, have the program exit right away with some sort of error message.)

Once you've built that, let's continue our tour of APIs, but this time you're going to do something a little different: you're going to take data from one API and feed it into another.

Location of the ISS

Now you're going to use an API to find the position of the International Space Station (ISS) and then plot where it currently is using Google Maps.

First, check out the beautiful documentation for the ISS Current Location API.[4] You can see the JSON endpoint,[5] and the structure of the response is clearly laid out:

```
{
  "message": "success",
  "timestamp": UNIX_TIME_STAMP,
  "iss_position": {
    "latitude": CURRENT_LATITUDE,
    "longitude": CURRENT_LONGITUDE
  }
}
```

The Google Maps API is *much* trickier to use. Rather than using the actual, official API, I poked around and found that the following URL will open a map with a marker at location "40, 10" (latitude and longitude), centered at that same location, and zoomed all the way out:

4. http://open-notify.org/Open-Notify-API/ISS-Location-Now/
5. http://api.open-notify.org/iss-now.json

```
https://www.google.com/maps?q=40,10&ll=40,10&z=1
```

That should work well enough for us. Let's have a look at this URL: it starts with the *protocol* (https://), then continues with the *domain* (www.google.com), is followed by the *path* (/maps), and ends with a *parameter list* (?q=40,10&ll=40,10&z=1). (It's common to pass in information to APIs via the URL parameter list, as you did with the Open Trivia Database API Mayonnaise Fiasco. Did you actually request that? I can't believe you sometimes.)

With this Google Maps URL, there are three parameters, all with very terse names: *q*, *ll*, and *z*. I have no idea what the *q* stands for, but it's the location of the Google Maps pointer. The *ll* stands for latitude/longitude, and it means the location where the map is centered. (So setting *q* and *ll* to the same value means that the map will be centered on the pointer.) Finally, the *z* is the zoom level, and you're zooming way out because the ISS is frequently over the ocean, and that's not an interesting map unless you're zoomed out.

As you probably already guessed, the parameter list of a URL (if it has any parameters at all) starts with a question mark, and parameters are separated by ampersands (&). Each parameter has a name, followed by an equals sign, and then a string of characters. So the parameter *q* has the value "40,10", which is a string of five characters. (All URL parameters are strings.)

Pulling all of this together, all that's left is to extract the location of the ISS from the first API and then build a new Google Maps URL with that location:

```
Line 1  require "net/http"
        require "json"
        
        url = URI("http://api.open-notify.org/iss-now.json")
    5
        request = Net::HTTP.get(url)
        
        iss_hash = JSON.parse(request)
        
   10   iss_pos = iss_hash["iss_position"]
        
        latitude  = iss_pos["latitude"]
        longitude = iss_pos["longitude"]
        lat_long  = "#{latitude},#{longitude}"
   15
        map_url = "https://maps.google.com/?q=#{lat_long}&ll=#{lat_long}&z=1"
        
        puts map_url

❮       https://maps.google.com/?q=-42.1338,-121.6594&ll=-42.1338,-121.6594&z=1
```

Notice that you didn't need to convert iss_pos['latitude'] into a float because you only use it to construct map_url, which is a string. (If you'd wanted to do something like double the latitude or longitude, then you'd first have to convert it into a float, double that value, and then convert back into a string.) So that was nice.

You know what would be *really* cool, though? If your program actually opened up your browser with that map, instead of printing the link. To do so, replace the last line of your program with one of these, depending on your OS:

```
system("open", map_url)             # macOS
system("xdg-open", map_url)         # Linux
system("start \"\" \"#{map_url}\"") # Windows
```

Pretty cool, huh? Run it once to see where it is, then wait one minute and run it again. Are you surprised at how fast that thing moves? I sure was.

Now that you've seen a few free APIs, you might go looking for more. If you do, you'll soon run into something called *API keys*. Let's talk about what those are, how to get them, and how to use them.

Where Did I Put My Keys?

We lock our homes when we go out, right? But I don't ever lock my shed. (Come to think of it, I don't even know if it has a lock.) There's nothing in there but some potting soil and some extra paving stones, nothing precious. You only lock up the stuff that matters.

APIs are the same way. Some of the simplest ones are open for anyone to use, with the trust that people will use them respectfully. But most APIs require an API key to access them. An API key is a random sequence of letters and numbers, kind of like a password that you don't get to choose (and certainly aren't expected to memorize).

As with a password, every user of an API will get a different API key. This makes it possible for the host of the API to revoke the keys of people who are using the API inappropriately, and it allows the host to impose *rate limiting* for everyone else.

Rate limiting is setting a limit on how often any one user can use an API. For example, let's say an API has a limit of 100 requests per minute. If you tried to use it 105 times in one minute, the last five requests would fail. You'd need to wait a minute before you could use it again. This makes it possible for the hosts of the API to ensure that no one is hogging all of the resources, and that everyone can still access the API at all times.

For some APIs, you have to pay for a key. Other APIs have free keys, and still others have a mixed model, where a free key gets you limited functionality or decreased rates compared to a paid key. Let's check out one of these now, so you can see how they work.

Movie Search

The Open Movie Database (OMDb) has a robust, well-documented API.[6] Right away, the documentation tells you that all commands start with:

```
http://www.omdbapi.com/?apikey=[yourkey]&...
```

In other words, the first URL parameter should be your API key (which you don't have yet), followed by the other parameters, which tell the API what it is you're trying to find.

Let's get you that API key first. There's a link at the top of the page that takes you to a form you need to fill out. You'll need to verify your email address, and then you'll get your API key. I'm not going to use my *actual* key in these examples (never give away your keys!), so I'll use `"1234abcd"` instead. But just note that this code *won't work for you* until you replace that with your own API key.

Once you've got your API key, you can make a request and see what you get. The docs say that you can use the *i* parameter to make a request using an IMDb ID, so let's try that with ID `"tt3896198"`:

```
Line 1  require "net/http"
        require "json"
        
        apikey  = "1234abcd" # REPLACE WITH YOUR API KEY
        request = "http://www.omdbapi.com/?apikey=#{apikey}"
        
        url = URI(request + "&i=tt3896198")
        
        movie_hash = JSON.parse(Net::HTTP.get(url))
        
        puts movie_hash

❮ {"Title"=>"Guardians of the Galaxy Vol. 2", "Year"=>"2017",
"Rated"=>"PG-13", "Released"=>"05 May 2017",...
```

Wow, it actually returns a lot of information, too much to include here: runtime, director, cast, plot, awards, and even a link to the movie poster. Nice.

6. http://www.omdbapi.com/

The docs also say that you can search by title, so let's see what other movies include the word "Guardians" in the title:

```
Line 1 require "net/http"
- require "json"
-
- apikey  = "1234abcd" # REPLACE WITH YOUR API KEY
5 request = "http://www.omdbapi.com/?apikey=#{apikey}"
-
- url = URI(request + "&s=\"Guardians\"")
-
- search_hash = JSON.parse(Net::HTTP.get(url))
10
- puts search_hash
```

Let's reformat the results for readability:

```
❮ {
    "Search"=>[
      {"Title"=>"Guardians of the Galaxy", "Year"=>"2014",
       "imdbID"=>"tt2015381", "Type"=>"movie", "Poster"=>...},
      {"Title"=>"Guardians of the Galaxy Vol. 2", "Year"=>"2017",
       "imdbID"=>"tt3896198", "Type"=>"movie", "Poster"=>...},
      ...
    ],
    "totalResults"=>"175",
    "Response"=>"True"
  }
```

It returns quite a few results, but that is only a fraction of the 175 results it says it has. The docs say you could use a *page* parameter in your request, implying that they paginate the results. In this case, it looks like they return ten results per page (and each request returns only one page). So, to get all of the results, you'd have to make 18 requests: 17 full pages of ten results each, and one partial page with the last five results.

Instead of that, though, what if you only wanted to see the name and type of each of the first ten results? And maybe also open a tab in your browser with the poster of the first result, and another tab with its IMDb page?

You could do something like this:

```
Line 1 require "net/http"
- require "json"
-
- apikey  = "1234abcd" # REPLACE WITH YOUR API KEY
5 request = "http://www.omdbapi.com/?apikey=#{apikey}"
-
- url = URI(request + "&s=\"Guardians\"")
-
- search_hash = JSON.parse(Net::HTTP.get(url))
```

```
10
movies = search_hash["Search"]

movies.each do |movie|
  puts "#{movie["Type"]}: #{movie["Title"]}"
end

poster_url = movies[0]["Poster"]
imdb_url = "https://www.imdb.com/title/#{movies[0]["imdbID"]}"

# adapt this for your OS, like in the previous ISS example
system("open", poster_url)
system("open", imdb_url)
```

```
movie: Guardians of the Galaxy
movie: Guardians of the Galaxy Vol. 2
movie: Rise of the Guardians
movie: Legend of the Guardians: The Owls of Ga'Hoole
movie: Guardians
movie: 7 Guardians of the Tomb
movie: Naruto the Movie 3: Guardians of the Crescent Moon Kingdom
series: Guardians of the Galaxy
game: Halo 5: Guardians
movie: The Guardians
```

There you go: a bunch of movies, but also an animated series and a video game. (I didn't even know they had video games in there.)

From here, you could go in a few different directions. Maybe you want to write a loop to fetch all of the results instead of only the first ten. Maybe you want to show the results with each one preceded by a number, like this:

```
1. (movie) Guardians of the Galaxy
2. (movie) Guardians of the Galaxy Vol. 2
3. (movie) Rise of the Guardians
...
```

Then, the user could type "3" to learn more about the third entry in the list, and from there maybe they get a second menu with options to list the cast, or open a browser tab with the poster or IMDb page, or whatever you want.

Do what sounds cool to you, and then go build it.

After you play around with that, try using another API all on your own. You got this.

A Few Things to Try

- *Build something cool with an API.* It's time for you to discover and use an API all on your own. Search online for an API that you find interesting,

and use it to build something cool. You can find APIs for Chuck Norris, Pokémon (that one looks *incredibly* thorough), Spotify, generating memes, and so on. Pick one and run with it.

Progress Checkpoint

In this chapter, you learned how to write programs that interact with other programs running somewhere else on the Internet. You learned what APIs are, how to explore them, how to use them, and how to use them *respectfully*.

In the next two chapters, you're going to get a bit more nerdy and abstract and focus on increasing your power and expressivity as a programmer. You'll start by learning some advanced techniques, which will allow you to create your own iterators and control flow structures. It's a level of expressivity that I, personally, couldn't live without.

CHAPTER 13

Blocks and Procs

Blocks and procs are definitely some of the coolest features of Ruby. Some other languages have this feature, though they may call it something else (like *closures* or *lambdas*), but many don't, and it's a shame. They are a joy to use.

With blocks and procs, you have the ability to take a *block* of code (code in between `do` and `end`), wrap it up in an object (known as a *proc*), store it in a variable or pass it to a method, and run the code in the block whenever you feel like it (more than once, if you want). So, it's kind of like a method itself, except it isn't bound to an object (it *is* an object), and you can store it or pass it around like you can with any object. Let's start with a short example:

```
Line 1  toast = Proc.new do
    2    puts "Cheers!"
    3  end
    4
    5  toast.call
    6  toast.call
    7  toast.call
```

```
Cheers!
Cheers!
Cheers!
```

Here, you create a proc—which I think is supposed to be short for *procedure*, but far more important, it rhymes with *block*—that holds the block of code. You then call the proc three times. This process is a lot like a method. In fact, blocks (and thus procs) can take parameters exactly like methods do. Here's how it looks:

```
Line 1  do_you_like = Proc.new do |good_stuff|
    2    puts "I *really* like #{good_stuff}!"
    3  end
    4
```

```
do_you_like.call("chocolate")
do_you_like.call("Ruby")
```

```
I *really* like chocolate!
I *really* like Ruby!
```

So, what's the point? Why not use methods?

As it turns out, there are some things you can't do with methods. In particular, you can't pass methods into other methods (but you can pass procs into methods), and methods can't return other methods (but they can return procs). This is simply because procs are objects; methods aren't.

By the way, is any of this looking familiar? Yep, you've seen blocks before…when you learned about iterators. But let's talk more about that in a bit.

Methods That Take Procs

When you pass a proc into a method, you can control when the proc is called, how many times it's called, or even if it's called at all. For example, let's say you want to do something before and after some code runs:

```
def do_self_importantly(some_proc)
  puts "Everybody just HOLD ON! I'm doing something..."
  some_proc.call
  puts "OK everyone, I'm done. As you were."
end

say_hello = Proc.new do
  puts "hello"
end

say_goodbye = Proc.new do
  puts "goodbye"
end

do_self_importantly(say_hello)
do_self_importantly(say_goodbye)
```

```
Everybody just HOLD ON! I'm doing something...
hello
OK everyone, I'm done. As you were.
Everybody just HOLD ON! I'm doing something...
goodbye
OK everyone, I'm done. As you were.
```

Maybe that doesn't appear particularly fabulous…but it is. It's all too common in programming to have strict requirements about what must be done when.

Remember opening and closing a file? If you want to save or load a file, you have to open the file, write or read the relevant data, and then close the file. If you forget to close the file, Bad Things can happen. But each time you want to save or load a file, you have to do the same thing: open the file, do what you *really* want to do, and then close the file. It's tedious and easy to forget. But with this trick, it's not even an issue.

You can also write methods that can determine how many times (or even *whether*) to call a proc. Here's a method that calls the proc passed in about half of the time and another that calls it twice:

```
def maybe_do(some_proc)
  if rand(2) == 0
    some_proc.call
  end
end

def twice_do(some_proc)
  some_proc.call
  some_proc.call
end

wink = Proc.new do
  puts "<wink>"
end

glance = Proc.new do
  puts "<glance>"
end

twice_do(wink)
twice_do(glance)
maybe_do(wink)
maybe_do(glance)
```

```
<wink>
<wink>
<glance>
<glance>
<wink>
```

These are some of the more common uses of procs that enable you to do things you simply can't do using methods alone. Sure, you could write a method to print "wink" twice, and you could probably do it in your sleep. But you couldn't write one to do *anything* twice.

Before moving on, let's look at one last example. So far, the procs you have passed in have been fairly similar to each other. This time they'll be different, so you can see how much such a method depends on the procs passed into

it. Your method will take a proc called some_proc and an object to pass into that proc. It'll then compute some_proc.call(input). If some_proc returns false, you quit; otherwise, you call the proc again, this time passing in the returned object. You keep doing this until the proc returns false (which it had better do eventually, or the program will never finish). The method will return the last non-false value returned by the proc. Here's what it looks like in action:

```
Line 1  def do_until_false(first_input, some_proc)
        output = first_input
        
        while output
          input  = output
          output = some_proc.call(input)
        end
        
        input
      end
      
      build_array_of_squares = Proc.new do |array|
        last_number = array.last
        if last_number <= 0
          false
        else
          #  Take off the last number...
          array.pop
          #  ...and replace it with its square...
          array.push(last_number*last_number)
          #  ...followed by the next smaller number.
          array.push(last_number-1)
        end
      end
      
      puts do_until_false([5], build_array_of_squares).inspect
      
      # now let's try a different proc and object
      
      always_false = Proc.new do |just_ignore_me|
        false
      end
      
      yum = "lemonade with a hint of orange blossom water"
      puts do_until_false(yum, always_false)
```

```
[25, 16, 9, 4, 1, 0]
lemonade with a hint of orange blossom water
```

Okay, so that was a pretty weird example, I'll admit. But it shows how differently a method can act when given different procs. (Do yourself a favor, and try that lemonade. It's unbelievable. Just a *hint*, though.)

You may have noticed the inspect method on line 26. It's a lot like to_s, except the string it returns tries to show you the Ruby code for building the object you passed it. Here it shows the whole array returned by your first call to do_until_false. Also, you might notice that you never actually squared that 0 on the end of that array, but since 0 squared is still 0, we didn't have to do this. And since always_false was, you know, always false, do_until_false didn't do anything at all the second time you called it; it simply returned what was passed in.

Now you know what blocks and procs are, how to create them, how to call them, and how to write methods that take procs. Next, you'll learn how to write methods that *return* procs.

Methods That Return Procs

One of the cool things you can do with procs is create them in methods and return them. This is the doorway to all sorts of mad genius programming power (things with impressive names, such as *lazy evaluation*, *infinite data structures*, and *currying*). I don't actually do these things often, but they are quite possibly the coolest programming techniques around.

In this example, compose takes two procs and returns a new proc that, when called, calls the first proc and passes its result into the second, like this:

```
Line 1  def compose(proc1, proc2)
        -   Proc.new do |x|
        -     proc2.call(proc1.call(x))
        -   end
      5 end
        -
        - square_it = Proc.new do |x|
        -   x * x
        - end
     10
        - double_it = Proc.new do |x|
        -   x + x
        - end
        -
     15 double_then_square = compose(double_it, square_it)
        - square_then_double = compose(square_it, double_it)
        -
        - puts double_then_square.call(5)
        - puts square_then_double.call(5)

      ❮ 100
        50
```

Now that you understand how to write a method to return procs, it's time to learn how to make this all more user-friendly.

Passing Blocks (Not Procs) into Methods

Okay, so this has been more theoretically cool than actually cool, partly because this is all a bit of a hassle to use. I can admit that. A lot of the problem is that you have to go through three steps (defining the method, making the proc, and calling the method with the proc) instead of only two (defining the method and passing the *block* of code right into the method, without using a proc at all), since you usually won't want to use the proc/block after you pass it into the method.

It should be…more like how iterators work. And you know what? It is. In this example, you'll create a new array iterator, like `each`, but instead of visiting each element in the array, you only look at every other element, like this:

```
Line 1  class Array
        def each_even(&was_a_block__now_a_proc)
          # We start with `true` because array
          # indexing starts with 0, which is even.
        5 is_even = true
          self.each do |object|
            if is_even
              was_a_block__now_a_proc.call object
       10   end
            # Toggle from even to odd, or odd to even.
            is_even = !is_even
          end
       15 end
        end

        fruits = ["apple", "bad apple", "cherry", "durian"]
        fruits.each_even do |fruit|
       20 puts "Yum! I just love #{fruit} pies, don't you?"
        end

        # Remember, we are getting the even-numbered *elements*
        # of the array, which in this case are all odd numbers,
       25 # because I live only to irritate you.
        [1, 2, 3, 4, 5].each_even do |odd_ball|
          puts "#{odd_ball} is NOT an even number!"
        end
```

```
Yum! I just love apple pies, don't you?
Yum! I just love cherry pies, don't you?
1 is NOT an even number!
3 is NOT an even number!
5 is NOT an even number!
```

To pass in a block to each_even, all you had to do was stick the block after the method. You can pass a block into any method this way, though many methods will ignore the block. To make your method *not* ignore the block but grab it and turn it into a proc, put the name you want the proc to have at the end of your method's parameter list, preceded by an ampersand (&). So, that part is a little tricky but not too bad, and you have to do that only once (when you define the method). Then you can use the method over and over again, like the built-in methods that take blocks, such as each and times. (Remember 3.times do…? That was so cute.)

If you get confused (I mean, there's this each and its block inside each_even), remember what each_even is supposed to do: call the block passed in with every other element in the array. Once you've written it and it works, you don't need to think about what it's actually doing under the hood ("which block is called when?"); in fact, that's exactly *why* we write methods like this—so we never have to think about how they work again. We simply use them.

I remember one time I wanted to *profile* some code I was writing; you know, I wanted to time how long it took to run. I wrote a method that takes the time before running the code block, then runs it, then takes the time again at the end, and finally figures out the difference. And it went a little something like this:

```
Line 1  def profile(block_description, &block)
        start_time = Time.new
        block.call
        duration = Time.new - start_time
      5 puts "#{block_description}:  #{duration} seconds"
        end

        profile("25000 doublings") do
        number = 1
     10
        25000.times do
          number = number + number
        end

     15 puts "#{number.to_s.length} digits"
        # That's the number of digits in this HUGE number.
        end

        profile("count to a million") do
     20 number = 0
        1000000.times do
          number = number + 1
        end
        end
```

```
7526 digits
25000 doublings:  0.020516 seconds
count to a million:  0.042442 seconds
```

How cool is that? How simple, how elegant? With that tiny method, we can now easily time any section of any program; you throw the code in a block, send it to profile, and do a little dance.... What could be simpler? Though you didn't do it, you could find the slow parts of your code and add more profiling calls nested *inside* your original calls. It's beautiful. In most languages, you'd have to explicitly add that timing code (the stuff in profile) around every section you wanted to time. What a hassle. But in Ruby you get to keep it all in one place and (more importantly) out of your way.

Now it's time to practice working with blocks on your own.

A Few Things to Try

- *Even better profiling.* After you do your profiling, see the slow parts of your program, and make them faster (or learn to love them as they are), you probably don't want to see all of that profiling anymore. But I hope you're too lazy (in the good way) to go back and delete it all…especially because you might want to use it again someday. Modify the profile method so you can turn all profiling on and off by changing a single line of code. It's actually one single word, in fact.

- *Grandfather clock.* Write a method that takes a block and calls it once for each hour that has passed today. That way, if I were to pass in the block:

  ```
  do
    puts "BONG!"
  end
  ```

  it would chime (sort of) like a grandfather clock. Test your method out with a few different blocks.

  Hint: You can use Time.new.hour to get the current hour. But this returns a number between 0 and 23, so you'll have to alter those numbers to get ordinary clock-face numbers (1 to 12).

- *Program logger.* Write a method called log that takes a string description of a block (and, of course, a block). Similar to the method do_self_importantly, it should puts one string telling you it started the block and another string at the end telling you that it finished and what the block returned. Test your method by sending it a code block. Inside the block, put *another* call to log, passing a block to it. In other words, your output should look something like this:

```
Beginning 'outer block'...
Beginning 'some little block'...
...'some little block' finished, returning:
5
Beginning 'yet another block'...
...'yet another block' finished, returning:
I like Thai food!
...'outer block' finished, returning:
false
```

- *Better program logger.* The output from that last logger was kind of hard to read, and it would only get worse the more you used it. It would be so much easier to read if it indented the lines in the inner blocks. So, you'll need to keep track of how deeply nested you are every time the logger wants to write something. To do this, use a *global variable*, which is a variable you can see from anywhere in your code. To make a global variable, precede your variable name with `$`, like so: `$global`, `$nesting_depth`, and `$big_top_pee_wee`. In the end, your logger should output code like this:

```
Beginning 'outer block'...
  Beginning 'some little block'...
    Beginning 'teeny-tiny block'...
    ...'teeny-tiny block' finished, returning:
    lots of love
  ...'some little block' finished, returning:
  42
  Beginning 'yet another block'...
  ...'yet another block' finished, returning:
  I love Indian food!
...'outer block' finished, returning:
true
```

Progress Checkpoint

You learned some pretty deep stuff in this chapter. With blocks and procs, you can now create your own custom iterators and control flow structures. That's *awesome*, but it might take a bit for this new knowledge to sink in.

Fortunately, you're all done with this book. Sort of. I've already taught you all the Ruby I'm going to teach you, anyway. In the next chapter, I'm going to bend your mind with an almost magical programming technique that you can use in any programming language. It's a little advanced, but if you made it this far, you can handle it.

CHAPTER 14

The Magic of Recursion

Congratulations, you're a programmer. At this point, you've learned most of the basics of programming a computer.

I'm sure it hasn't been easy. If your brain isn't already hurting by this point, it's probably because you already knew some programming before picking up this book.

Since you've done so well in getting this far, I'll tell you what: I won't teach you anything new about Ruby in this chapter. Instead, I'll show you new ways of using what you've already learned. And maybe turn your brain inside out in the process.

Because recursion will do that.

A Method So Easy, It Calls Itself

You know how to write methods, and you know how to call methods. When you write methods, you'll usually fill them with more method calls. You can make methods, and those methods can call methods…

You can write a method that calls itself.

That's *recursion*.

Well, on the surface, it's an absurd idea; if all a method did was call itself, which would call itself again, it would loop like that forever. (Although this isn't technically a loop, it's similar to one; you can usually replace loops with recursion if you want.) But of course, it could do other things as well and maybe call itself only some of the time.

Recursion is particularly well-suited for situations where solving part of the problem "looks like" solving the entire problem. One classic example

is computing the factorial of a number. The factorial of five is `5*4*3*2*1`. This isn't difficult to write without using recursion, of course:

```
Line 1  def factorial(n)
        # This is where we will store the product
        # as we compute it.
        product = 1
    5
        while n > 0
          product = product * n
          n = n - 1
        end
   10
        product # return the result
      end
        
      puts factorial(8)
```

```
40320
```

Recursion comes in handy when you realize that the factorial of five is actually five times the factorial of four. With that insight, you could rewrite `factorial` like this:

```
Line 1  def factorial(n)
    2     if n <= 1
    3       1
    4     else
    5       n * factorial(n-1)
    6     end
    7   end
    8
    9   puts factorial(8)
```

```
40320
```

Look at how clean that is, and it definitely uses less code, which is nice. Personally, I feel like this is easier to read and better expresses the *meaning* of the computation, as opposed to the *process* of performing the computation.

Let's look at another example, one in which the solution is much easier if you use recursion. The problem: write a method that takes an integer `n` and returns the nth Fibonacci number. The Fibonacci numbers are 1, 1, 2, 3, 5, 8..., where each number is the sum of the two before it: 1+1 is 2, 1+2 is 3, 2+3 is 5, etc.

Using recursion, this is pretty simple to write:

```
Line 1  def fibonacci(n)
  2       if n <= 1
  3         1
  4       else
  5         fibonacci(n-1) + fibonacci(n-2)
  6       end
  7     end
  8
  9     puts fibonacci(20)
```

```
❮ 10946
```

Admittedly, this isn't the most efficient way to compute Fibonacci numbers, because it does the same calculation over and over again. (For example, when computing fibonacci(n), you compute fibonacci(n-2) twice, since fibonacci(n-1) also computes fibonacci(n-2).) But computers are ridiculously fast, so often you don't care. You know what isn't ridiculously fast? You and me. In most cases, I'd rather save myself time and make the computer do the extra work.

Also, a more efficient method would be harder to write *correctly*, and much more prone to having a bug in it. Constrast that with the recursive fibonacci method: it's so dead-simple that of course it's correct.

Actually, you should write a non-recursive fibonacci method so you can see how much harder it is to write. It's like writing code "against the grain." Pay attention to how it feels, how solving the parts *feels* like solving the whole. Go ahead, I'll wait.

Now let's work through a more interesting example. Let's say you want to flatten an array of objects, some of which are themselves (possibly nested) arrays, so that what you have at the end is one flat array. You could write a method named flatten that would take a nested array and return a new, flat array like this:

```
Line 1  def flatten(arr)
  2       # do something
  3     end
  4
  5     a = [[2, 3], [4, [[5], 6]]]
  6     b = [8, [[[[9]]]]]
  7     arr = [[1, a], [7], b]
  8
  9     puts arr.inspect
 10     puts flatten(arr).inspect
```

```
❮ [[1, [[2, 3], [4, [[5], 6]]]], [7], [8, [[[[9]]]]]]
  [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

To write flatten, you'd start with an empty array named flat. Then, you'd iterate over the array and check each element to see if it's an array or not. If not, no problem: push it onto the array. If it's an array, though, you want to add that to the array.

In code, the solution would look like this:

```
Line 1  def flatten(arr)
        flat = []

        arr.each do |elem|
5         if elem.class == Array
            flat = flat + elem
          else
            flat.push(elem)
          end
10      end

        flat
      end

15    a = [[2, 3], [4, [[5], 6]]]
      b = [8, [[[[9]]]]]
      arr = [[1, a], [7], b]

      puts arr.inspect
20    puts flatten(arr).inspect
```

```
[[1, [[2, 3], [4, [[5], 6]]]], [7], [8, [[[[9]]]]]]
[1, [[2, 3], [4, [[5], 6]]], 7, 8, [[[[9]]]]]
```

Hmm… I guess it's flatt*er*, but it's not all the way flattened—and there's this feeling, like *this would be easier if the elements of the array were themselves flattened arrays*. That's the hint, that feeling. When you follow that, you can see how close to a solution you already are:

```
Line 1  def flatten(arr)
        flat = []

        arr.each do |elem|
5         if elem.class == Array
            flat = flat + flatten(elem) # this line changed
          else
            flat.push(elem)
          end
10      end

        flat
      end

15    a = [[2, 3], [4, [[5], 6]]]
```

```
b = [8, [[[[9]]]]]
arr = [[1, a], [7], b]

puts arr.inspect
puts flatten(arr).inspect
```

```
[[1, [[2, 3], [4, [[5], 6]]]], [7], [8, [[[[9]]]]]]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

And there you have it—a nice solution. The only change is on line 6 where, instead of adding `elem`, you add a flattened version of `elem`.

It feels like magic. You had this broken `flatten` method, and you fixed it by *assuming it worked properly* and calling it on each `elem`. That's what recursive programming feels like.

How about another, real-world example? When I was generating the worlds for the tile-based strategy game *Civilization III*, I wanted worlds with two primary supercontinents as they tend to be a lot of fun and sort of feel "earthy" and *real*. So, after I generated the land masses, I wanted to test them to see what the sizes were of the different continents. If I had two of relatively equal size (say, differing by a factor of 2 or less) and no others close in size, then I'd call that a pretty good world.

The process, then, was something like this:

1. Build the world.
2. Find a "continent" (which could be a one-tile island…at this point I wouldn't know).
3. Compute its size.
4. Find another continent (making sure not to count any of them twice but also making sure each gets counted), and repeat the process.
5. Then find the largest two, and see whether they look like fun to play on.

The fun part was in computing each continent's size, because the best way to do that, was recursively.

Let's look at a trimmed-down version. Let's say you have an 11x11 world represented as an array of arrays—basically a grid—and you want to find the size of the continent in the middle (that is, the continent of which tile (5,5) is a part). You don't want to count any land tiles belonging to any of the other continents. Also, as in *Civilization III*, let's say that tiles touching only at the corners are still considered to be on the same continent since land units could move along diagonals.

Before we get to the code, let's solve the problem in English first. One plan is to look at every tile on the map, and if that tile is a land tile on the continent you're looking for, you add 1 to the running total. The problem, though, is how do you know whether a land tile is on the same continent as some other land tile? All the ways that solve this problem feel kind of messy.

Here's the key: two tiles are on the same continent if you can walk from one to the other. (That was essentially the operating definition of continent in *Civilization III*.) So that's how the code should work. First, you count the spot you're standing on (makes sense); in this case, that means tile (5,5). You then send out eight little friends, one in each direction, and tell them to count the rest of the continent in that direction. The only rule is that no one can count a tile that someone else has already counted. When those eight friends return, you add their answers to your already-running total (which is 1, the tile you started with), and that's your answer.

Cool, except for one little detail…how are those eight little helpers supposed to determine the size of the continent? Essentially, you shrugged the problem onto them. The only tile you counted was the one you were standing on. That's pretty darn lazy, which is probably a good thing.

So, how are your eight little helpers supposed to compute the size of the continent? The same way you do. Yet, somehow, thanks to the help of a small band of lazy, imaginary helpers counting only the tile they're on, you get the size of the entire continent. (You still need to make sure no tile is counted twice, but you can mark each tile as it's visited to keep track.) Without further ado, behold the magic of recursion:

```
Line 1  # These are just to make the map easier to read.
    -   # "M" is visually more dense than "_".
    -   M = "land"
    -   _ = "water"
    5
    -   world = [[_,_,_,_,_,_,_,_,_,_,_],
    -            [_,_,_,_,M,M,_,_,_,_,_],
    -            [_,_,_,_,_,_,_,_,M,M,_],
    -            [_,_,_,M,_,_,_,_,_,M,_],
   10           [_,_,_,M,_,M,M,_,_,_,_],
    -            [_,_,_,_,M,M,M,M,_,_,_],
    -            [_,_,_,M,M,M,M,M,M,M,_],
    -            [_,_,_,M,M,_,M,M,M,_,_],
    -            [_,_,_,_,_,_,M,M,_,_,_],
   15           [_,M,_,_,_,M,_,_,_,_,_],
    -            [_,_,_,_,_,_,_,_,_,_,_]]
    -
    -   def continent_size(world, x, y)
    -     if world[y][x] != "land"
```

```
    # Either it's water or we already counted it,
    # but either way, we don't want to count it now.
    return 0
  end
  # So first we count this tile...
  size = 1
  world[y][x] = "counted land"

  # ...then we count all of the neighboring eight tiles
  # (and of course, their neighbors, recursively).
  size = size + continent_size(world, x-1, y-1)
  size = size + continent_size(world, x  , y-1)
  size = size + continent_size(world, x+1, y-1)
  size = size + continent_size(world, x-1, y  )
  size = size + continent_size(world, x+1, y  )
  size = size + continent_size(world, x-1, y+1)
  size = size + continent_size(world, x  , y+1)
  size = size + continent_size(world, x+1, y+1)
  size
end

puts continent_size(world, 5, 5)
```

Drumroll, please....

```
❮ 23
```

And there you have it! Even if the world was much larger and the continent was totally bizarre and oddly shaped, it would still work fine.

Well, there's actually one small bug you need to fix. This code works fine because the continent doesn't border the edge of the world. If it did, then when you send your little helpers out (that is, call `continent_size` on a new tile), some of them would fall off the edge of the world (that is, call `continent_size` with invalid values for `x` and/or `y`), which would probably crash on the very first line of the method.

It seems that the obvious way to fix this would be to do a check before each call to `continent_size` (sort of like sending your little helpers out only if they aren't going to fall over the edge of the world), but that would require eight separate (yet nearly identical) checks in your method. Yuck! It would be lazier to simply send your helpers out and have them shout back "ZERO!" if they fall off the edge of the world. In other words, put the check right at the top of the method, very much like the check you put in to see whether the tile was uncounted land. Go for it! Of course, you'll have to make sure it works; test it by extending the continent to touch one (or better yet, all four) of the edges of the world.

And that, my friend, is recursion. It's not anything new, in some sense. It's merely a new way of thinking of the same old stuff you already knew.

Now that you have a grip on recursion, you're ready for an important rite of passage: writing a sorting method.

Rite of Passage: Sorting

Remember the sorting program you wrote on page 53 where you asked for a list of words, sorted it, and then displayed the sorted list? The program was made much easier because you used the array's sort method. But, like the Jedi who constructs their own lightsaber, you'll demonstrate a deeper mastery if you write your own sorting method. It's not easy, but this kind of problem solving is part of nearly every program you'll write. We've all done it, and now you will, too.

But where do you begin? Much like with continent_size, it's probably best to try to solve the problem in English first. You can then translate it into Ruby after you've wrapped your head around it.

So, you want to sort an array of words, and you know how to find out which of two words comes first in the dictionary (using <).

For this exercise, you're going to use the "quicksort" algorithm, which goes something like this:

1. Pick one element out of the array, which we'll call the *pivot*.
2. Now split everything else in the array into two piles (arrays): stuff that's smaller than the pivot, and stuff that's larger.
3. Recursively sort each of those piles, using this same quicksort algorithm.
4. Return the fully sorted array, which is the sorted pile of smaller things, then the pivot, then the sorted pile of larger things.

Translating that into code and comments:

```
Line 1  def sort(arr)
        # grab one item out of the array, call it `pivot`
        pivot = arr.pop
        
      5 # get the elements in arr that are
        # smaller than `pivot`
        smaller_elements = # needs code here
        
        # get the elements in arr that are
     10 # larger than `pivot`
        larger_elements = # needs code here
```

```
-
-   # now the recursive magic
-   smaller_sorted = sort(smaller_elements)
15  larger_sorted  = sort(larger_elements )
-
-   # return the result: a completely sorted array
-   smaller_sorted + [pivot] + larger_sorted
- end
```

You break the array into elements that are smaller than the pivot and elements larger than the pivot, and then you sort those. This recursively calls `sort` on smaller and smaller arrays, until it gets down to empty arrays at the bottom. You'll need to handle the case of sorting an empty array because in that case there will be no `pivot` to get out of the array. (It's pretty easy, though, an empty array is already sorted, you can simply return it.)

There's one more tricky bit to this: what if the array has multiple elements which are equal? When you split the elements into the larger and smaller groups, you'll need to pick one of those to hold the elements that are equal to `pivot`. It doesn't matter which you use, so long as every element ends up in exactly one of the two groups.

Hopefully that doesn't sound too bad, but it's keeping all of the details straight that makes it so tricky. Go ahead and try it, and see how it looks.

When you're done, make sure to test your code. Type in duplicate words and things like that. A great way to test would be to use the built-in `sort` method to get a sorted version of your array right away. Then, after you've sorted it for yourself, make sure the two lists are equal.

Now that you've got that down, here are a few more exercises for you. Some of these are going to be tricky, but you can do this.

A Few Things to Try

- *Shuffle.* Now that you've finished your new sorting algorithm, how about the opposite? Write a `shuffle` method that takes an array and returns a totally shuffled version. As always, you'll want to test it, but testing this one is trickier: How can you test to make sure you're getting a perfect shuffle? What would you even say a perfect shuffle would be? Now test for that.

- *Dictionary sort.* Your sorting algorithm is pretty good, sure. But there was always that slightly embarrassing thing you were hoping I'd gloss over, right? About the capital letters? Your sorting algorithm is good for general-purpose sorting, but when you sort strings, you're using the ordering of

the characters in your fonts (their Unicode values) rather than true dictionary ordering. In a dictionary, case (upper or lower) is irrelevant to the ordering. So, make a new method to sort words (something like dictionary_sort). Remember, though, that if the user gives your program words starting with capital letters, it should return the same words with those same capital letters, but ordered as you'd find them in a dictionary.

- *English number.* Write a method named english_number. It'll take an integer, like 22, and return the English version of it (in this case, the string "twenty-two"). To start, have it handle only integers from 0 to 100. Then add the ability to handle larger numbers, so that a thousand becomes "ten hundred" and a million becomes "one hundred hundred hundred". The key is to use recursion.

  Now expand upon english_number. First, put in thousands; it should return "one thousand" instead of "ten hundred" and "ten thousand" instead of "one hundred hundred".

  Then add millions, so you get "one million" instead of "one thousand thousand". Then add billions, trillions, and any other -illions you want to add.

- *"Ninety-nine Bottles of Beer on the Wall."* Using english_number and your old program on page 45, write out the lyrics to this song the *right* way this time. Super-size it: have it start at 9999. (Don't pick a number too large, though, because writing all of that to the screen takes Ruby a while. A hundred thousand bottles of beer takes some time; and if you pick a million, you'll probably regret it.)

Progress Checkpoint

In this chapter, you learned what recursion is, how to write recursive methods, and some tips on how to tell if a recursive solution is the way to go when solving a problem. You also completed your rite of passage. There's not a whole lot more for me to teach you, except where to find answers from other resources, which is what you'll find in the next, and final, chapter.

CHAPTER 15

Beyond This Book

Well, that's about all you're going to learn from this book. Congratulations, programmer! You've learned a *lot*. Of course, you're going to forget bits here and there (we all do). And that's fine. Programming isn't about what you know; it's about what you can figure out. As long as you know where to look to find the things you forgot, you're doing fine. (I was looking up stuff constantly as I was writing this book.)

Where do you look stuff up (besides here)? ♪♫ If there's something strange, and it don't look good…who you gonna call? ♫

I look for help with Ruby in a few places. If it's a small question, and I think I can experiment on my own to find the answer, I use irb. If it's a bigger question, I look it up in my PickAxe or on Ruby-Doc.org, or I do a Google search. And if I can't figure it out on my own, then I ask for help on Stack Overflow.

Interactive Ruby (irb)

If you installed Ruby, then you installed irb. To use it, go to your command prompt and type `irb`. When you're in irb, you can type in any Ruby expression you want, and it'll tell you the value of that expression. Type in `1+2`, and it'll tell you `3` (note that you don't have to use `puts`). It's kind of like a giant Ruby calculator. When you're done, type `exit`.

I use irb when I have something I want to test real quick. For example, is it `array.len` or `array.length`? (I program in several languages, so it's easy to get confused.) Or maybe I want to know if arrays have a `reverse` method like strings do? I can test these in a few seconds with irb. You can even define methods and classes in irb (though they'll only last until you exit irb).

There's a lot more to irb than this, but you can learn all about it in the PickAxe.

The PickAxe: Programming Ruby

Absolutely *the* Ruby book to get is *Programming Ruby 1.9 & 2.0, The Pragmatic Programmers' Guide* by Dave Thomas and others (from the Pragmatic Bookshelf).[1]

You can find almost everything about Ruby, from the basic to the advanced, in this book. It's easy to read, it's comprehensive, and it's close to perfect. I wish every language had a book of this quality. At the back of the book, you'll find a huge section detailing every method in every class, explaining it, and giving examples. (This is why you want the fourth edition.) I love this book!

The title of the book suggests it's only for Ruby 1.9 and 2.0, but it's good for all 2.x versions of Ruby. And since it's mostly only advanced features that have changed in Ruby 3, it's still a great reference.

And why is it referred to as the *PickAxe?* There's a picture of a pickaxe on the cover of the book. It's a silly nickname, I guess, but it stuck.

Ruby-Doc.org

Aside from the PickAxe, another place to find the latest documentation for all of Ruby's built-in classes is on Ruby-Doc.org.[2] This site documents everything Ruby comes with right out of the box. Check it out.

Online Search

I probably don't need to even mention this one, but using Google is a great way to find answers to your programming questions. Whatever your question, I guarantee someone has asked it before.

One thing you'll soon notice is that more than half of the time, Google finds an answer to your programming question on Stack Overflow.

Stack Overflow

Stack Overflow is the greatest source of answers to technical questions that has ever existed. More often than not, Google will send you there. Kind,

1. https://pragprog.com/titles/ruby4/
2. https://ruby-doc.org/

amazing, brilliant people have been answering questions there for years, and they have you covered.

In the rare circumstance that you have a question that hasn't been asked yet, go to Stack Overflow[3] and ask it yourself. Not only will you get your answer, but someone in the future won't need to ask the same question, and they'll get their answer, too.

Now that you know where to go to learn more and get help when you're stuck, there are a few last things I wanted to share before I go.

Tim Toady

Something I've tried to shield you from, but that you'll surely run into soon, is the concept of TMTOWTDI (pronounced *Tim Toady*, I think): There's More Than One Way To Do It. The idea is that, in Ruby, there's more than one way to do just about anything.

Some people will tell you what a wonderful thing TMTOWTDI is, while others will feel the opposite. I think it's pretty cool, because having more than one way to do something feels more expressive. Nonetheless, I think it's a *terrible* way to teach someone how to program. (Learning one way to do something is challenging and confusing enough.)

But now that you're moving beyond this book, you'll be seeing much more diverse code. For example, I can think of at least five other ways to make a string (aside from surrounding some text in double quotes), and each one works slightly differently. I showed you only the most common of the six.

And when we talked about branching, I showed you `if`, but I didn't show you `unless`. I'll let you figure that one out in irb.

Another nice little shortcut you can use with `if`, `unless`, and `while` is the cute one-line version:

```
Line 1 # These words are from a program I wrote to generate
  2 # English-like babble. English is so weird.
  3
  4 puts "combergearl thememberate" if 5 == 2**2 + 1**1
  5 puts "supposine follutify" unless "Chris".length == 5

  ❮ combergearl thememberate
```

And finally, there's another way of writing methods that take blocks (not procs). You saw the thing where you grabbed the block and turned it into a

3. https://stackoverflow.com/questions

proc using the &block trick in your parameter list when you define the method. Then, to call the block, you use block.call. Well, there's a shorter way that some prefer. Instead of this:

```
Line 1  def do_it_twice(&block)
  2       block.call
  3       block.call
  4     end
  5
  6     do_it_twice do
  7       puts "murditivent flavitemphan siresent litics"
  8     end

❮   murditivent flavitemphan siresent litics
    murditivent flavitemphan siresent litics
```

You do this:

```
Line 1  def do_it_twice
  2       yield
  3       yield
  4     end
  5
  6     do_it_twice do
  7       puts "buritiate mustripe lablic acticise"
  8     end

❮   buritiate mustripe lablic acticise
    buritiate mustripe lablic acticise
```

I don't know. What do you think? Personally, I found yield confusing at first. If it was something like call_the_hidden_block, that would have made a lot more sense to me. On the other hand, a lot of people say yield makes more sense to them. But I guess that's what TMTOWTDI is all about: they do it their way, and you can do it however you want.

You Did It

Programming is difficult. Maybe you feel like you didn't *really* understand everything up to this point, or maybe you skipped some of the exercises, so maybe you feel like you aren't *really* a programmer yet. This is normal, and it even has a name: *imposter syndrome*.

Let me tell you a little secret: feeling like an imposter is one of the most common traits shared by programmers. Let me tell you another secret: you know who never feels like a programming imposter? *People who can't actually program a computer at all.* After over two decades of programming, I still feel intimidated and frustrated at times.

So welcome to the club. I feel like this deserves a high five, so how about you just smack the book right here, and I'll high-five this exact line on my screen, and we'll call it success.

And if you liked the book, or if you didn't (but especially if you did), drop me a line (chris@pine.fm) or find me on Twitter (@OtherChrisPine).

Use it for good and not for evil. ☺

APPENDIX 1

Installation and Setup on Windows

Let's get you all set up. First, you need to install Ruby. Use the convenient One-Click Installer.[1] When you run it, it'll ask you where you want to install Ruby and which parts of it you want installed. You can accept all the defaults.

Now let's make a folder on your desktop in which you'll keep all of your programs. Right-click your desktop, select New, and then select Folder. Name it something truly memorable, such as `programs`. Now double-click the folder to open it.

To make a blank Ruby program, right-click in the folder, select New, and then select Text Document. Rename the document to have the `.rb` file extension. So, if it was `New Text Document.txt`, rename it to `ramen.rb` (if your program is about ramen, I mean).

Now you need a text editor. I'm a fan of Sublime Text,[2] so unless you already have a favorite text editor, go ahead and download and install that one.

To actually run your programs, you'll need to go to your command line. In your Start menu, select Accessories, and then choose Command Prompt. You'll see something like this:

```
Microsoft Windows [Version 10.0.18362.959]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Chris>_
```

This is telling you that you're in the `C` drive, in the `Users` folder, in the `Chris` subfolder. (Unless your name is also Chris, it'll probably say a different name.) And the cursor at the end will probably be blinking; it's your computer's way of asking, "What would you like?"

1. https://rubyinstaller.org/
2. https://www.sublimetext.com/

You're now at the command line, which is your direct connection to the soul of your computer. You want to be somewhat careful way down here, since it's not *too* hard to do Bad Things (such as erase everything on your computer). But if you don't try anything too wacky, you should be fine.

So, here you are, sitting and staring at your computer. It would only be polite to say "hello" at this point, so type `echo hello` on the command line, and press `Enter`. Your computer should reply with a friendly `hello` as well, making your screen look something like this:

```
C:\Users\Chris> echo hello
hello

C:\Users\Chris>_
```

And your cursor is blinking again in a "What's next?" sort of way. Now that you're acquainted, ask it to make sure Ruby is installed properly and to tell you the version number. You do this with `ruby -v`:

```
C:\Users\Chris> ruby -v
ruby 3.0.0p0 (2020-12-25) [x64-mingw32]

C:\Users\Chris>_
```

Excellent, it's there. All you have left now is to find your `programs` folder through your command line. It's on your desktop, so you need to go there first. You do this with `cd desktop`:

```
C:\Users\Chris> cd desktop

C:\Users\Chris\Desktop>_
```

And now you see that you're in the `Desktop` subfolder of `Chris` (or whatever your name is).

Why `cd`? Well, way back in the olden days, before CDs (when people were "getting down" to eight-track cassettes and phonographs and such) and when command lines roamed the earth in their terrible splendor, people didn't call them *folders* on your computer. After all, there were no pictures of folders, so people didn't think of them as folders. They called them *directories*. So, they didn't "move from folder to folder." They "changed directories." But if you actually try typing `change_directory desktop` all day long, you barely have time to get down to your funky eight-tracks; so, it was shortened to `cd`.

If you want to go back up a directory, you use `cd ..` :

```
C:\Users\Chris\Desktop> cd ..

C:\Users\Chris>_
```

And to see all of the directories you can cd into from where you are, use dir /ad:

```
C:\Users\Chris> dir /ad
 Volume in drive C is System
 Volume Serial Number is 843D-8EDC

 Directory of C:\Users\Chris

07.10.2020  14:30    <DIR>          ..
02.09.2020  10:45    <DIR>          Application Data
04.10.2020  16:19    <DIR>          Cookies
07.10.2020  14:30    <DIR>          .
07.10.2020  14:24    <DIR>          Desktop
15.08.2020  13:17    <DIR>          Favorites
10.02.2020  02:50    <DIR>          Local Settings
05.09.2020  13:17    <DIR>          My Documents
15.08.2020  14:14    <DIR>          NetHood
10.02.2020  02:50    <DIR>          PrintHood
07.10.2020  15:23    <DIR>          Recent
10.02.2020  02:50    <DIR>          SendTo
10.02.2020  02:50    <DIR>          Start Menu
25.02.2020  14:57    <DIR>          Templates
25.02.2020  12:07    <DIR>          UserData
               0 File(s)              0 bytes
              15 Dir(s)   6 720 483 328 bytes free

C:\Users\Chris>_
```

And there you have it.

All right, are you ready? Take a deep breath. Let's program!

APPENDIX 2

Installation and Setup on macOS

Let's get you all set up. If you're using macOS, you're in luck: Ruby is already installed for you.

Now you need a text editor. I'm a fan of Sublime Text,[1] so unless you already have a favorite text editor, go ahead and download and install that one.

Next, you should make a folder on your desktop in which to keep your programs. Right-click on your desktop, and select New Folder. You want to give it a name both descriptive and alluring, such as programs. Nice.

Now, let's get to know your computer a little better. The best way to have a one-on-one with your computer is on the command line. You get there through the Terminal application (found in the Finder by navigating to Applications/Utilities). Open it, and you'll see something like this:

```
Last login: Thu Oct 8 11:21:05 on ttys006
mezzaluna:~ chris$ _
```

That cursor at the end might be blinking, and it might be a vertical line or box instead of an underscore. Whatever it looks like, it's your computer's way of asking, "What would you like?"

It's telling me when I last logged in (but if it's your first time, it might not say that) and giving me a *command prompt* and cursor. Prompts, like hairdos, come in a variety of shapes, sizes, colors, and levels of expressivity. This isn't the prompt I normally use, but it's the default prompt. It's showing the name of this computer ("mezzaluna"), something that looks like two dots (":"), something else I'll tell you about in a bit ("~"), who I am ("chris"), and then a dollar sign ("$"). This is for good luck, I guess. Maybe it's trying to give my name a little bling. I don't know.

1. https://www.sublimetext.com/

You are now at the command line, which is the heart and soul of your computer. You want to be somewhat careful what you do down here, since it's not *too* hard to do Bad Things here. (It's easier to delete everything on your computer than it is to get rid of that dollar sign, for example.) But if you don't try anything too rambunctious, you should be fine.

Here you are, sitting and staring at your computer. It would only be polite to say "hello" at this point, so type `echo hello` on the command line, and press Return. Your computer should reply with a friendly `hello` as well, making your screen look something like this:

```
mezzaluna:~ chris$ echo hello
hello
mezzaluna:~ chris$ _
```

And your cursor is blinking again in a "What's next?" sort of way. Now that you're acquainted, ask your computer which version of Ruby is installed. Do this with `ruby -v`:

```
mezzaluna:~ chris$ ruby -v
ruby 3.0.0p0 (2020-12-25 revision 95aff21468) [x86_64-darwin20]
```

That's good—I have Ruby installed.

Now that Ruby is ready to rumble, it's time to learn how to get around your computer from the command line and what that ~ in the prompt is all about.

The ~ is a short way of saying "your home directory," which is a nerdy way of saying "your default folder," which is still kind of nerdy anyway. And I'm okay with that.

That's where you are: your home directory. If you want to change to a different directory, you use `cd`. (No one wants to type `change-directory`, not even once. I mean, I had to just then, to make a point, but in general you don't want to type it. Too many letters.)

```
mezzaluna:~ chris$ cd Desktop
mezzaluna:~/Desktop chris$ _
```

So, my prompt changed, telling me that I'm now on my desktop, which is in my home directory. (Notice that `Desktop` was capitalized. If you don't capitalize it, your computer will get angry and begin to swear at you in computerese, with such insults as "No such" and "file" and the worst one of all: "bash.") You can go back up a directory with `cd ..`, which in this case would put you back in your home directory. And at any time, if you simply type `cd`, that takes you to your home directory, no matter where you are. Like clicking the heels of your ruby slippers.

But you don't want to go to either of those folders. You want to go to your programs folder (or directory, or whatever). Assuming you're still in your Desktop folder (if not, get there quickly!), do this:

```
mezzaluna:~/Desktop chris$ cd programs
mezzaluna:~/Desktop/programs chris$ _
```

But you probably could have guessed that.

Perfect, now you're ready to program.

All right, are you ready? Take a deep breath. Let's program!

APPENDIX 3

Installation and Setup on Linux

Let's get you all set up. If you're using Linux, you probably already have a favorite text editor, you know how to install Ruby with your package manager, and you'd better already know where to find your command line. ☺

If you don't have a text editor you're fond of, though, I'm a fan of Sublime Text.[1] It's made specifically for programming, and it plays well with Ruby.

Next, you'll want to see whether you have Ruby installed already. Type `which ruby` on your command line. If you see a scary-looking message that looks something like `/usr/bin/which: no ruby in (...)`, then you'll need to install it. Otherwise, see what version of Ruby you're running with `ruby -v`.

If you need to install Ruby, you can do so through your package manager.

Once that's done, all that's left is to create a directory somewhere to keep your programs in, `cd` into that directory, and you're all set.

All right, are you ready? Take a deep breath. Let's program!

1. https://www.sublimetext.com/

APPENDIX 4

A Few Things to Try: Possible Solutions

Since the first edition of this book, the single question people ask me the most is "Where are the answers to the exercises?"

My reluctance centered around the first occurrence of the word *the* in that question.

The answers? There's more than one right answer, of course. Many, many more. These aren't math problems. Even the first exercises, which are sort of like math problems, have many possible solutions. If, instead of writing a program about orange trees or the minutes in a decade, you were asked to write a poem about them, it would be silly (if not downright harmful) to include "the answers."

That was my reasoning, anyway. Kind of silly, in retrospect—while these aren't math problems, neither are they poems.

Still, I'm attached to the idea that there's no one right answer here, so I did a few things to emphasize this. First, notice the title to this appendix: *possible* solutions, not *the* solutions.

Then I went through and did each exercise twice. The first time is to show one possible way that you *could* have done it, given what you've learned up to that point in the book. The second time is to show you how I'd do it, using whatever techniques tickled my fancy. Some of these techniques aren't covered in this book, so it's okay if you don't understand exactly what's going on. These programs tend to be more complex but also shorter (sometimes *much* shorter) and sometimes more correct or robust. Often cuter. (I like cute code.)

Ignore them or study them as you prefer.

Exercises from Chapter 1

Hours in a Year

(Found on page 4.)

Write a program that tells you how many hours are in a year.

How You Could Do It

```
Line 1  puts 24*365
```

```
❮ 8760
```

How I Would Do It

```
Line 1  # depends on if it's a leap year
   2  puts 24*365
   3  puts "(or #{24*366} on a leap year)"
```

```
❮ 8760
  (or 8784 on a leap year)
```

Minutes in a Decade

(Found on page 4.)

Write a program that tells you how many minutes are in a decade.

How You Could Do It

```
Line 1  puts 60*24*(365*10 + 2)
```

```
❮ 5258880
```

How I Would Do It

```
Line 1  # depends on how many leap years in that decade
   2  puts "#{60*24*(365*10 + 2)} or #{60*24*(365*10 + 3)}"
```

```
❮ 5258880 or 5260320
```

Your Age in Seconds

(Found on page 4.)

Write a program that tells you how many seconds old you are.

How You Could Do It

```
Line 1  puts 60*60*24*365*44
```

```
❮ 1387584000
```

How I Would Do It

```
Line 1 puts(Time.new - Time.gm(1976, 8, 3, 13, 31))
```

```
1394433250.961471
```

Our Dear Author's Age

(Found on page 4.)

If I'm 1,390 million seconds old, how old am I?

How You Could Do It

```
Line 1 puts 1390000000/(60*60*24*365)
```

```
44
```

How I Would Do It

I'd do it pretty much the same way you would. ☺

Exercises from Chapter 2

404: Exercises Not Found

There were no exercises in Chapter 2.

How You Could Do It

Stare at this page for a few seconds.

How I Would Do It

Close this book, make a nice cup of tea, and then pick back up where you left off.

Exercises from Chapter 3

Please see the possible solutions to the exercises from Chapter 2 on page 151.

Exercises from Chapter 4

Full Name Greeting

(Found on page 19.)

Write a program that asks for a person's first name, then middle name, and then last name. Finally, have the program greet the person using their full name.

How You Could Do It

```
Line 1  puts "What is your first name?"
   2  f_name = gets.chomp
   3  puts "What is your middle name?"
   4  m_name = gets.chomp
   5  puts "What is your last name?"
   6  l_name = gets.chomp
   7
   8  full_name = f_name + " " + m_name + " " + l_name
   9
  10  puts "Hello, " + full_name + "!"
```

```
❮ What is your first name?
⇛ Sam
❮ What is your middle name?
⇛ I
❮ What is your last name?
⇛ Am
❮ Hello, Sam I Am!
```

How I Would Do It

```
Line 1  puts "What's your first name?"
   2  f_name = gets.chomp
   3  puts "What's your middle name?"
   4  m_name = gets.chomp
   5  puts "What's your last name?"
   6  l_name = gets.chomp
   7
   8  puts "Hello, #{f_name} #{m_name} #{l_name}."
```

```
❮ What's your first name?
⇛ Marvin
❮ What's your middle name?
⇛ K.
❮ What's your last name?
⇛ Mooney
❮ Hello, Marvin K. Mooney.
```

Bigger, Better Favorite Number

(Found on page 19.)

Write a program that asks for a person's favorite number. Have your program add 1 to the number, and then suggest the result as a bigger and better favorite number.

How You Could Do It

```
Line 1  puts "Hey! What's your favorite number?"
   2  fav_num = gets.chomp.to_i
   3  better_num = fav_num + 1
```

```
4 puts "That's ok, I guess, but isn't #{better_num} just a bit better?"

❮ Hey! What's your favorite number?
⇒ 5
❮ That's ok, I guess, but isn't 6 just a bit better?
```

How I Would Do It

```
Line 1 puts "Hey! What's your favorite number?"
2 fav_num = gets.chomp.to_i
3 puts "That's ok, I guess, but isn't #{fav_num + 1} just a bit better?"

❮ Hey! What's your favorite number?
⇒ 5
❮ That's ok, I guess, but isn't 6 just a bit better?
```

Exercises from Chapter 5

Angry Boss

(Found on page 28.)

Write an angry boss program that rudely asks what you want. Whatever you answer, the angry boss should yell it back to you and then fire you.

How You Could Do It

```
Line 1 puts "CAN'T YOU SEE I'M BUSY?! MAKE IT FAST, JOHNSON!"
2 request = gets.chomp
3 puts "WHADDAYA MEAN \"#{request.upcase}\"?!? YOU'RE FIRED!!"

❮ CAN'T YOU SEE I'M BUSY?! MAKE IT FAST, JOHNSON!
⇒ I want a raise
❮ WHADDAYA MEAN "I WANT A RAISE"?!? YOU'RE FIRED!!
```

How I Would Do It

```
Line 1 names = ["johnson", "smith", "weinberg", "filmore"]
2 puts "CAN'T YOU SEE I'M BUSY?! MAKE IT FAST, #{names[rand(4)].upcase}!"
3 request = gets.chomp
4 puts "WHADDAYA MEAN \"#{request.upcase}\"?!? YOU'RE FIRED!!"

❮ CAN'T YOU SEE I'M BUSY?! MAKE IT FAST, WEINBERG!
⇒ I quit
❮ WHADDAYA MEAN "I QUIT"?!? YOU'RE FIRED!!
```

Table of Contents

(Found on page 28.)

Write a program that displays a table of contents so that it looks like this:

```
                Table of Contents

Chapter 1: Numbers                        page 1
Chapter 2: Letters                        page 5
Chapter 3: Variables                      page 9
```

How You Could Do It

```
title  = "Table of Contents".center(50)
chap_1 = "Chapter 1: Numbers".ljust(30)   + "page 1".rjust(20)
chap_2 = "Chapter 2: Letters".ljust(30)   + "page 5".rjust(20)
chap_3 = "Chapter 3: Variables".ljust(30) + "page 9".rjust(20)

puts title
puts
puts chap_1
puts chap_2
puts chap_3
```

How I Would Do It

How I'd do it? Well, that's a different exercise (at the end of Chapter 8).

Exercises from Chapter 6

"99 Bottles of Beer on the Wall"

(Found on page 45.)

Write a program that prints out the lyrics to that beloved classic, "99 Bottles of Beer on the Wall."

How You Could Do It

```
num_at_start = 5 # change to 99 if you want

num_now = num_at_start

while num_now > 2
  puts "#{num_now} bottles of beer on the wall, " +
       "#{num_now} bottles of beer!"
  num_now = num_now - 1

  puts "Take one down, pass it around, " +
       "#{num_now} bottles of beer on the wall!"
end

puts "2 bottles of beer on the wall, 2 bottles of beer!"
puts "Take one down, pass it around, 1 bottle of beer on the wall!"
puts "1 bottle of beer on the wall, 1 bottle of beer!"
puts "Take one down, pass it around, no more bottles of beer on the wall!"
```

```
5 bottles of beer on the wall, 5 bottles of beer!
Take one down, pass it around, 4 bottles of beer on the wall!
4 bottles of beer on the wall, 4 bottles of beer!
Take one down, pass it around, 3 bottles of beer on the wall!
3 bottles of beer on the wall, 3 bottles of beer!
Take one down, pass it around, 2 bottles of beer on the wall!
2 bottles of beer on the wall, 2 bottles of beer!
Take one down, pass it around, 1 bottle of beer on the wall!
1 bottle of beer on the wall, 1 bottle of beer!
Take one down, pass it around, no more bottles of beer on the wall!
```

How I Would Do It

```
num_at_start = 5 # change to 99 if you want

num_bot = proc { |n| "#{n} bottle#{n == 1 ? '' : 's'}" }

num_at_start.downto(2) do |num|
  puts "#{num_bot[num]} of beer on the wall, #{num_bot[num]} of beer!"
  puts "Take one down, pass it around, #{num_bot[num-1]} of beer on the wall!"
end

puts "#{num_bot[1]} of beer on the wall, #{num_bot[1]} of beer!"
puts "Take one down, pass it around, no more bottles of beer on the wall!"
```

```
5 bottles of beer on the wall, 5 bottles of beer!
Take one down, pass it around, 4 bottles of beer on the wall!
4 bottles of beer on the wall, 4 bottles of beer!
Take one down, pass it around, 3 bottles of beer on the wall!
3 bottles of beer on the wall, 3 bottles of beer!
Take one down, pass it around, 2 bottles of beer on the wall!
2 bottles of beer on the wall, 2 bottles of beer!
Take one down, pass it around, 1 bottle of beer on the wall!
1 bottle of beer on the wall, 1 bottle of beer!
Take one down, pass it around, no more bottles of beer on the wall!
```

Deaf Grandma

(Found on page 45.)

Write a program to simulate talking to your grandmother, who is hard of hearing. Whatever you say to Grandma (whatever you type in), she should respond with this:

```
HUH?! SPEAK UP, SONNY!
```

unless you shout it (type in all capitals). If you shout, she can hear you (or at least she thinks so) and yells back:

```
NO, NOT SINCE 1938!
```

How You Could Do It

```
Line 1  puts "HEY THERE, SONNY! GIVE GRANDMA A KISS!"
-
-   while true
-     said = gets.chomp
5
-     if said == "BYE"
-       puts "BYE SWEETIE!"
-       break
-     end
10
-     if said != said.upcase
-       puts "HUH?! SPEAK UP, SONNY!"
-     else
-       random_year = 1930 + rand(21)
15      puts "NO, NOT SINCE #{random_year}!"
-     end
-   end

❮   HEY THERE, SONNY! GIVE GRANDMA A KISS!
⇛   hi, grandma
❮   HUH?! SPEAK UP, SONNY!
⇛   HI, GRANDMA!
❮   NO, NOT SINCE 1939!
⇛   HOW YOU DOING?
❮   NO, NOT SINCE 1942!
⇛   I SAID, HOW YOU DOING?
❮   NO, NOT SINCE 1937!
⇛   OK
❮   NO, NOT SINCE 1947!
⇛   BYE
❮   BYE SWEETIE!
```

How I Would Do It

```
Line 1  puts "HEY THERE, SONNY! GIVE GRANDMA A KISS!"
-
-   while true
-     said = gets.chomp
5     break if said == "BYE"
-
-     response = if said != said.upcase
-       "HUH?! SPEAK UP, SONNY!"
-     else
10      "NO, NOT SINCE #{rand(1930..1950)}!"
-     end
-
-     puts response
-   end
15
-   puts "BYE SWEETIE!"
```

```
 HEY THERE, SONNY! GIVE GRANDMA A KISS!
hi, grandma
 HUH?! SPEAK UP, SONNY!
HI, GRANDMA!
 NO, NOT SINCE 1935!
HOW YOU DOING?
 NO, NOT SINCE 1941!
I SAID, HOW YOU DOING?
 NO, NOT SINCE 1940!
OK
 NO, NOT SINCE 1944!
BYE
 BYE SWEETIE!
```

Deaf Grandma Extended

(Found on page 46.)

What if Grandma doesn't want you to leave? When you shout `BYE`, she could pretend not to hear you. Change your previous program so that you have to shout `BYE` three times *in a row*.

How You Could Do It

```
Line 1 puts "HEY THERE, PEACHES! GIVE GRANDMA A KISS!"
  - bye_count = 0
  - while true
  -   said = gets.chomp
  5   if said == "BYE"
  -     bye_count = bye_count + 1
  -   else
  -     bye_count = 0
  -   end
 10   if bye_count >= 3
  -     puts "BYE-BYE CUPCAKE!"
  -     break
  -   end
  -   if said != said.upcase
 15     puts "HUH?! SPEAK UP, SONNY!"
  -   else
  -     random_year = 1930 + rand(21)
  -     puts "NO, NOT SINCE #{random_year}!"
  -   end
 20 end
```

```
 HEY THERE, PEACHES! GIVE GRANDMA A KISS!
HI, GRANDMA!
 NO, NOT SINCE 1942!
BYE
 NO, NOT SINCE 1939!
```

```
⇛ BYE
❮ NO, NOT SINCE 1942!
⇛ ADIOS, ABUELITA!
❮ NO, NOT SINCE 1930!
⇛ BYE
❮ NO, NOT SINCE 1950!
⇛ BYE
❮ NO, NOT SINCE 1932!
⇛ BYE
❮ BYE-BYE CUPCAKE!
```

How I Would Do It

```
Line 1 puts "HEY THERE, PEACHES! GIVE GRANDMA A KISS!"
  - bye_count = 0
  -
  - while true
  5   said = gets.chomp
  -   if said == "BYE"
  -     bye_count += 1
  -   else
  -     bye_count  = 0
 10   end
  -   break if bye_count >= 3
  -
  -   response = if said != said.upcase
  -     "HUH?! SPEAK UP, SONNY!"
 15   else
  -     "NO, NOT SINCE #{rand(1930..1950)}!"
  -   end
  -
  -   puts response
 20 end
  -
  - puts "BYE-BYE CUPCAKE!"
```

```
❮ HEY THERE, PEACHES! GIVE GRANDMA A KISS!
⇛ HI, GRANDMA!
❮ NO, NOT SINCE 1946!
⇛ BYE
❮ NO, NOT SINCE 1937!
⇛ BYE
❮ NO, NOT SINCE 1936!
⇛ ADIOS, ABUELITA!
❮ NO, NOT SINCE 1932!
⇛ BYE
❮ NO, NOT SINCE 1946!
⇛ BYE
❮ NO, NOT SINCE 1934!
⇛ BYE
❮ BYE-BYE CUPCAKE!
```

Leap Years

(Found on page 46.)

Write a program that asks for a starting year and an ending year and then prints all the leap years between them (and including them, if they are also leap years).

How You Could Do It

```
Line 1 puts "Pick a starting year (like 1973 or something):"
  - starting = gets.chomp.to_i
  -
  - puts "Now pick an ending year:"
  5 ending = gets.chomp.to_i
  -
  - puts "Check it out, these years are leap years:"
  - year = starting
  -
 10 while year <= ending
  -   if year%4 == 0
  -     if year%100 != 0 || year%400 == 0
  -       puts year
  -     end
 15   end
  -
  -   year = year + 1
  - end
```

```
❮ Pick a starting year (like 1973 or something):
⇛ 1973
❮ Now pick an ending year:
⇛ 1977
❮ Check it out, these years are leap years:
  1976
```

How I Would Do It

```
Line 1 puts "Pick a starting year (like 1973 or something):"
  - starting = gets.chomp.to_i
  - puts "Now pick an ending year:"
  - ending = gets.chomp.to_i
  5
  - puts "Check it out, these years are leap years:"
  - (starting..ending).each do |year|
  -   next if year%4   != 0
  -   next if year%100 == 0 && year%400 != 0
 10   puts year
  - end
```

```
❮ Pick a starting year (like 1973 or something):
⇛ 1973
❮ Now pick an ending year:
```

```
⇛ 1977
❮ Check it out, these years are leap years:
  1976
```

Exercises from Chapter 7

Building and Sorting an Array

(Found on page 53.)

Write a program that asks you to type as many words as you want (one word per line, continuing until you press Enter on an empty line) and then repeats the words back in alphabetical order.

How You Could Do It

```
Line 1 puts "Give me some words, and I will sort them:"
  - words = []
  -
  - while true
  5   word = gets.chomp
  -   if word == ""
  -     break
  -   end
  -
 10   words.push(word)
  - end
  -
  - sorted_words = words.sort
  -
 15 puts "Sweet! Here they are, sorted:"
  - puts sorted_words

❮ Give me some words, and I will sort them:
⇛ banana
⇛ apple
⇛ cherry
⇛
❮ Sweet! Here they are, sorted:
  apple
  banana
  cherry
```

How I Would Do It

```
Line 1 puts "Give me some words, and I will sort them:"
  - words = []
  -
  - while true
  5   word = gets.chomp
  -   break if word.empty?
  -
```

```
  words << word
end

puts "Sweet! Here they are, sorted:"
puts words.sort
```

```
Give me some words, and I will sort them:
banana
apple
cherry

Sweet! Here they are, sorted:
apple
banana
cherry
```

Table of Contents, Revisited

(Found on page 53.)

Rewrite your table of contents program on page 28. Start the program with an array holding all of the information for your table of contents (chapter names, page numbers, and so on). Then print out the information from the array in a beautifully formatted table of contents.

How You Could Do It

```
title = "Table of Contents"

chapters = [["Numbers",   1],
            ["Letters",   5],
            ["Variables", 9]]

puts title.center(50)
puts

chap_num = 1

chapters.each do |chap|
  name = chap[0]
  page = chap[1]

  beginning = "Chapter #{chap_num}: #{name}"
  ending    = "page #{page}"

  puts beginning.ljust(30) + ending.rjust(20)
  chap_num = chap_num + 1
end
```

```
                Table of Contents

Chapter 1: Numbers                          page 1
```

```
Chapter 2: Letters                          page 5
Chapter 3: Variables                        page 9
```

How I Would Do It

```
Line 1  title = "Table of Contents"
-
-   chapters = [["Numbers",   1],
-               ["Letters",   5],
5               ["Variables", 9]]
-
-   puts title.center(50)
-   puts
-
10  chapters.each_with_index do |chap, idx|
-     name, page = chap
-     chap_num   = idx + 1
-
-     beginning = "Chapter #{chap_num}: #{name}"
15    ending    = "page #{page}"
-
-     puts beginning.ljust(30) + ending.rjust(20)
-   end
```

```
                Table of Contents

Chapter 1: Numbers                          page 1
Chapter 2: Letters                          page 5
Chapter 3: Variables                        page 9
```

Exercises from Chapter 8

More Flavors Competing

(Found on page 66.)

Add some more flavors to the ULTIMATE FLAVOR TOURNAMENT program, and make it so that it's easy to add new flavors.

How You Could Do It

```
Line 1  # note: match_A_B means "round A, match B"
-   match_1_1 = ["vanilla", "chocolate"]
-   match_1_2 = ["rhubarb", "pistachio"]
-   match_1_3 = ["spumoni", "green tea"]
5   match_1_4 = ["cherry", "strawberry"]
-   match_2_1 = []  # this will hold the winners from 1_1 & 1_2
-   match_2_2 = []  # this will hold the winners from 1_3 & 1_4
-   match_3_1 = []  # this will hold the winners from 2_1 & 2_2
-   winner    = nil # this will hold the final winner
10
-   def ask_for_winner(flavors)
```

```
  puts "0. "+flavors[0]
  puts "1. "+flavors[1]

  while true
    answer = gets.chomp.downcase
    if (answer == "0" || answer == "1")
      return flavors[answer.to_i]
    else
      puts "Please answer '0' or '1'."
    end
  end
end

puts "Welcome to ULTIMATE FLAVOR TOURNAMENT!"
puts
puts "ROUND 1, MATCH 1: Which flavor is best?"
match_2_1[0] = ask_for_winner(match_1_1)
puts
puts "ROUND 1, MATCH 2: Which flavor is best?"
match_2_1[1] = ask_for_winner(match_1_2)
puts
puts "ROUND 1, MATCH 3: Which flavor is best?"
match_2_2[0] = ask_for_winner(match_1_3)
puts
puts "ROUND 1, MATCH 4: Which flavor is best?"
match_2_2[1] = ask_for_winner(match_1_4)
puts
puts "ROUND 2, MATCH 1: Which flavor is best?"
match_3_1[0] = ask_for_winner(match_2_1)
puts
puts "ROUND 2, MATCH 2: Which flavor is best?"
match_3_1[1] = ask_for_winner(match_2_2)
puts
puts "CHAMPIONSHIP MATCH!"
puts "Which flavor is best?"
winner = ask_for_winner(match_3_1)
puts
puts "And the Ultimate Flavor Champion is:"
puts winner.upcase+"!!"
```

```
Welcome to ULTIMATE FLAVOR TOURNAMENT!

ROUND 1, MATCH 1: Which flavor is best?
0. vanilla
1. chocolate
0

ROUND 1, MATCH 2: Which flavor is best?
0. rhubarb
1. pistachio
1
```

```
ROUND 1, MATCH 3: Which flavor is best?
0. spumoni
1. green tea
⇛ 0
❮
ROUND 1, MATCH 4: Which flavor is best?
0. cherry
1. strawberry
⇛ 1
❮
ROUND 2, MATCH 1: Which flavor is best?
0. vanilla
1. pistachio
⇛ 1
❮
ROUND 2, MATCH 2: Which flavor is best?
0. spumoni
1. strawberry
⇛ 0
❮
CHAMPIONSHIP MATCH!
Which flavor is best?
0. pistachio
1. spumoni
⇛ 1
❮
And the Ultimate Flavor Champion is:
SPUMONI!!
```

How I Would Do It

```
Line 1  # This program should work fine,
  -     # even with an odd number of flavors.
  -     flavors = [
  -       "vanilla",
  5       "chocolate",
  -       "rhubarb",
  -       "pistachio",
  -       "spumoni",
  -       "green tea",
 10       "strawberry",
  -     ]
  -
  -     def pair_up(array)  # [1,2,3,4,5] -> [[1,2], [3,4], [5]]
  -       array.each_slice(2).to_a
 15     end
  -
  -     def ask_for_winner(match)
  -       puts "1. "+match[0]
  -       puts "2. "+match[1]
 20
  -       while true
```

```
    answer = gets.chomp.downcase
    if (answer == "1" || answer == "2")
      return match[answer.to_i-1]
    else
      puts "Please answer '1' or '2'."
    end
  end
end

# play just one round of the tournament
def play_round(round, round_num)
  winners = []

  round.each_with_index do |match, idx|
    if match.length == 1
      # We had an odd number of flavors for this round,
      # so this one silently advances by default.
      winners << match[0]
    else
      match_num = idx+1
      puts
      if round.length == 1
        puts "CHAMPIONSHIP MATCH!"
      else
        puts "ROUND #{round_num}, MATCH #{match_num}"
      end
      puts "Which flavor is best?"
      winners << ask_for_winner(match)
    end
  end

  winners
end

# "round" will be an array of arrays of matches:
# rounds = [
#   [match_1, match_2, ...],  # round 1
#   [match_1, match_2, ...],  # round 2
#   ...
# ]
# each match is an array of two flavors
rounds = []

# populate "rounds" with Round 1 matches
rounds << pair_up(flavors)
round_idx = 0


puts "Welcome to ULTIMATE FLAVOR TOURNAMENT!"

# each pass of this loop is a new round
while true
```

```
-   winners = play_round(rounds[round_idx], round_idx+1)
75
-   if winners.length >= 2
-     # prepare for next round
-     rounds << pair_up(winners)
-     round_idx += 1
80  else
-     # We have a winner!
-     puts
-     puts "And the Ultimate Flavor Champion is:"
-     puts winners[0].upcase+"!!"
85    break
-   end
- end

❮ Welcome to ULTIMATE FLAVOR TOURNAMENT!

  ROUND 1, MATCH 1
  Which flavor is best?
  1. vanilla
  2. chocolate
⇛ could I just get some chips, pls?
❮ Please answer '1' or '2'.
⇛ 1
❮
  ROUND 1, MATCH 2
  Which flavor is best?
  1. rhubarb
  2. pistachio
⇛ 2
❮
  ROUND 1, MATCH 3
  Which flavor is best?
  1. spumoni
  2. green tea
⇛ 1
❮
  ROUND 2, MATCH 1
  Which flavor is best?
  1. vanilla
  2. pistachio
⇛ 2
❮
  ROUND 2, MATCH 2
  Which flavor is best?
  1. spumoni
  2. strawberry
⇛ 1
❮
  CHAMPIONSHIP MATCH!
  Which flavor is best?
  1. pistachio
```

```
2. spumoni
⇛ 2
❮
And the Ultimate Flavor Champion is:
SPUMONI!!
```

Old-School Roman Numerals

(Found on page 66.)

Write a method that when passed an integer between 1 and 3000 (or so) returns a string containing the proper old-school Roman numeral.

How You Could Do It

```
Line 1 def old_roman_numeral(num)
  -   roman = ""
  -
  -   roman = roman + "M" * (num        / 1000)
  5   roman = roman + "D" * (num % 1000 /  500)
  -   roman = roman + "C" * (num %  500 /  100)
  -   roman = roman + "L" * (num %  100 /   50)
  -   roman = roman + "X" * (num %   50 /   10)
  -   roman = roman + "V" * (num %   10 /    5)
 10   roman = roman + "I" * (num %    5 /    1)
  -
  -   roman
  - end
  -
 15 puts(old_roman_numeral(1999))

❮ MDCCCCLXXXXVIIII
```

How I Would Do It

```
Line 1 def old_roman_numeral(num)
  -   raise "Must use positive integer" if num <= 0
  -   roman = ""
  -
  5   roman << "M" * (num        / 1000)
  -   roman << "D" * (num % 1000 /  500)
  -   roman << "C" * (num %  500 /  100)
  -   roman << "L" * (num %  100 /   50)
  -   roman << "X" * (num %   50 /   10)
 10   roman << "V" * (num %   10 /    5)
  -   roman << "I" * (num %    5 /    1)
  -
  -   roman
  - end
 15
  - puts(old_roman_numeral(1999))

❮ MDCCCCLXXXXVIIII
```

"Modern" Roman Numerals

(Found on page 66.)

Write a method that when passed an integer between 1 and 3000 (or so) returns a string containing the "new" style Roman numeral.

How You Could Do It

```
Line 1  def roman_numeral(num)
        thous = (num        / 1000)
        hunds = (num % 1000 /  100)
        tens  = (num %  100 /   10)
    5   ones  = (num %   10       )

        roman = "M" * thous

        if hunds == 9
   10     roman = roman + "CM"
        elsif hunds == 4
          roman = roman + "CD"
        else
          roman = roman + "D" * (num % 1000 / 500)
   15     roman = roman + "C" * (num %  500 / 100)
        end

        if tens == 9
          roman = roman + "XC"
   20   elsif tens == 4
          roman = roman + "XL"
        else
          roman = roman + "L" * (num %  100 /  50)
          roman = roman + "X" * (num %   50 /  10)
   25   end

        if ones == 9
          roman = roman + "IX"
        elsif ones == 4
   30     roman = roman + "IV"
        else
          roman = roman + "V" * (num %   10 /   5)
          roman = roman + "I" * (num %    5 /   1)
        end
   35
        roman
      end

      puts(roman_numeral(1999))
```

```
MCMXCIX
```

How I Would Do It

```
Line 1  def roman_numeral(num)
        raise "Must use positive integer" if num <= 0

        digit_vals = [["I",    5,    1],
                      ["V",   10,    5],
                      ["X",   50,   10],
                      ["L",  100,   50],
                      ["C",  500,  100],
                      ["D", 1000,  500],
                      ["M",  nil, 1000]]

        roman     = ""  # string we are building
        remaining = nil # string of 'remainder' letters

        # Build string 'roman' in reverse.
        digit_vals.each do |l, m, n|
          num_l = m ? (num % m / n) : (num / n)
          full  = m && (num_l == (m/n - 1))

          if full && (num_l > 1 || remaining)
            # must carry
            remaining ||= l # carry l if not already carrying
          else
            if remaining
              roman << l + remaining
              remaining = nil
            end

            roman << l * num_l
          end
        end

        roman.reverse
      end

      puts(roman_numeral(1999))
```

```
❮ MIM
```

Exercises from Chapter 9

Safer Photo Downloading

(Found on page 77.)

Adapt the photo-downloading/file-renaming program to your computer, and add some safety features to make sure you never overwrite a file.

How You Could Do It

Well, since I was asking you to adapt it to *your* computer, I can't show you how to do it. I'll show you the program I *actually* wrote, though.

It's a bit more complex than the other examples here, partly because it's a real, working tool.

How I Would Do It

```
Line 1  # For Katy, with love.
-
-   ### Download pictures from camera card.
-
5   require "win32ole"
-
-   STDOUT.sync = true
-   Thread.abort_on_exception = true
-
10  Dir.chdir("C:\Documents and Settings\Katy\Desktop\pictureinbox")
-
-   # Always look here for pics.
-   pic_names = Dir["!undated/**/*.{jpg,avi}"]
-   thm_names = Dir["!undated/**/*.{thm}"    ]
15
-   # Scan for memory cards in the card reader.
-   WIN32OLE.new("Scripting.FileSystemObject").Drives.each() do |x|
-     #driveType 1 is removable disk
-     if x.DriveType == 1 && x.IsReady
20     pic_names += Dir[x.DriveLetter+":/**/*.{jpg,avi}"]
-      thm_names += Dir[x.DriveLetter+":/**/*.{thm}"    ]
-    end
-  end
-
25  months = %w(jan feb mar apr may jun jul aug sep oct nov dec)
-
-   encountered_error = false
-
-   print "Downloading #{pic_names.size} files: "
30
-   pic_names.each do |name|
-     print "."
-     is_movie = (name[-3..-1].downcase == "avi")
-
35   if is_movie
-      orientation = 0
-      new_name = File.open(name) do |f|
-        f.seek(0x144,IO::SEEK_SET)
-        f.read(20)
40    end
-
-     new_name[0...3] = "%.2d" % (1 + months.index(new_name[0...3].downcase))
-     new_name = new_name[-4..-1] + " " + new_name[0...-5]
```

```
  else
45  new_name, orientation = File.open(name) do |f|
    f.seek(0x36, IO::SEEK_SET)
    orientation_ = f.read(1)[0]
    f.seek(0xbc, IO::SEEK_SET)
    new_name_ = f.read(19)
50  [new_name_, orientation_]
  end
end

[4,7,10,13,16].each {|n| new_name[n] = "."}
55 if new_name[0] != "2"[0]
  encountered_error = true
  puts "\nERROR: Could not process '#{name}'"+
    " because it's not in the proper format!"
  next
60 end

save_name = new_name + (is_movie ? ".orig.avi" : ".jpg")
# Make sure we don't save over another file!!
while FileTest.exist?(save_name)
65  new_name += "a"
  save_name = new_name + (is_movie ? ".orig.avi" : ".jpg")
end

case orientation
70  when 6
    `convert "#{name}" -rotate "90>"  "#{save_name}"`
    File.delete name
  when 8
    `convert "#{name}" -rotate "-90>" "#{save_name}"`
75  File.delete name
  else
    File.rename(name, save_name)
end
end
80

print "\nDeleting #{thm_names.size} THM files: "
thm_names.each do |name|
  print "."
  File.delete name
85 end
# If something bad happened, make sure she
# sees the error message before the window closes.
if encountered_error
  puts
90  puts "Press [Enter] to finish."
  puts
  gets
end
```

Exercises from Chapter 10

One Billion Seconds!

(Found on page 81.)

Find out the exact second you were born, or as close as you can get. Figure out when you'll turn (or perhaps when you did turn) one billion seconds old.

How You Could Do It

Well, I don't know your birthday, so I don't know how you'd do it.

How I Would Do It

```
Line 1  # I don't know what second I was born.
        puts(Time.utc(1976, 8, 3, 13, 31) + 10**9)
        
        # And yes, I had a party. It was awesome!
```

```
2008-04-11 15:17:40 UTC
```

Happy Birthday!

(Found on page 81.)

Ask what year a person was born, then the month, and then the day. Figure out how old they are, and give them a ☺ (smiley emoji) for each birthday they've had.

How You Could Do It

```
Line 1  puts "What year were you born?"
        b_year = gets.chomp.to_i
        
        puts "What month were you born? (1-12)"
        b_month = gets.chomp.to_i
        
        puts "What day of the month were you born?"
        b_day = gets.chomp.to_i
        
        b = Time.local(b_year, b_month, b_day)
        t = Time.new
        
        age = 1
        
        while Time.local(b_year + age, b_month, b_day) <= t
          puts "☺"
          age = age + 1
        end
```

```
What year were you born?
```

```
⇛ 2017
❮ What month were you born? (1-12)
⇛ 2
❮ What day of the month were you born?
⇛ 2nd
❮ ☺
  ☺
  ☺
```

How I Would Do It

```
Line 1  puts "Hey, when were you born? (Please use YYYYMMDD format.)"
  -     input = gets.chomp
  -
  -     b_year  = input[0..3].to_i
  5     b_month = input[4..5].to_i
  -     b_day   = input[6..7].to_i
  -
  -     t = Time.new
  -
 10     t_year  = t.year
  -     t_month = t.month
  -     t_day   = t.day
  -
  -     age = t_year - b_year
 15
  -     if t_month < b_month || (t_month == b_month && t_day < b_day)
  -       age -= 1
  -     end
  -
 20     if t_month == b_month && t_day == b_day
  -       puts "HAPPY BIRTHDAY!!"
  -     end
  -
  -     puts("☺ "*age)
```

```
❮ Hey, when were you born? (Please use YYYYMMDD format.)
⇛ 20151212
❮ ☺ ☺ ☺ ☺ ☺
```

Party Like It's MCMXCIX!

(Found on page 86.)

Write a method that takes a string containing a Roman numeral, and returns the corresponding integer.

How You Could Do It

```
Line 1  def roman_to_integer(roman)
  -       digit_vals = {"i" =>    1,
  -                     "v" =>    5,
  -                     "x" =>   10,
```

```
                "l" =>   50,
                "c" =>  100,
                "d" =>  500,
                "m" => 1000}
  total = 0
  prev  = 0
  index = roman.length - 1
  while index >= 0
    c = roman[index].downcase
    index = index - 1
    val = digit_vals[c]
    if !val
      puts "This is not a valid roman numeral!"
      return
    end

    if val < prev
      val = val * -1
    else
      prev = val
    end
    total = total + val
  end

  total
end

puts(roman_to_integer("mcmxcix"))
puts(roman_to_integer("CCCLXV"))
```

```
1999
365
```

How I Would Do It

```
def roman_to_integer(roman)
  digit_vals = {"i" =>    1,
                "v" =>    5,
                "x" =>   10,
                "l" =>   50,
                "c" =>  100,
                "d" =>  500,
                "m" => 1000}
  total = 0
  prev  = 0
  roman.reverse.each_char do |c_or_C|
    c   = c_or_C.downcase
    val = digit_vals[c]
    if !val
      puts "This is not a valid roman numeral!"
      return
    end
```

```
    if val < prev
      val *= -1
    else
      prev = val
    end
    total += val
  end

  total
end

puts(roman_to_integer("mcmxcix"))
puts(roman_to_integer("CCCLXV"))
```

```
1999
365
```

Birthday Helper!

(Found on page 86.)

Write a program to read in names and birth dates from a text file. It should then ask you for a name. You type one in, and it tells you when that person's next birthday will be (and, for the truly adventurous, how old they'll be). The input file should look something like this:

```
Chris Hemsworth,  Aug 11, 1983
Chris Evans,      Jun 13, 1981
Chris Pratt,      Jun 21, 1979
Chris Pine,       Aug 26, 1980
Other Chris Pine, Aug  3, 1976
```

How You Could Do It

```
# First, load in the birthdates.
birth_dates = {}
File.read("birthdates.txt").each_line do |line|
  line = line.chomp
  # Find the index of first comma,
  # so we know where the name ends.
  first_comma = 0
  while line[first_comma] != "," &&
        first_comma < line.length
    first_comma = first_comma + 1
  end

  name = line[0..(first_comma - 1)]
  date = line[-12..-1]

  birth_dates[name] = date
end
```

```
# Now ask the user which one they want to know.
puts "Whose birthday would you like to know?"
name = gets.chomp
date = birth_dates[name]

if date == nil
  puts "Oooh, I don't know that one..."
else
  puts date[0..5]
end
```

```
Whose birthday would you like to know?
Chris Pratt
Jun 21
```

How I Would Do It

```
# First, load in the birthdates.
birth_dates = {}

File.readlines("birthdates.txt").each do |line|
  name, date, year = line.split(",")
  birth_dates[name] = Time.gm(year, *(date.split))
end

# Now ask the user which one they want to know.
puts "Whose birthday would you like to know?"
name = gets.chomp
bday = birth_dates[name]

if bday == nil
  puts "Oooh, I don't know that one..."
else
  now = Time.new
  age = now.year - bday.year

  if now.month > bday.month || (now.month == bday.month && now.day > bday.day)
    age += 1
  end

  if now.month == bday.month && now.day == bday.day
    puts "#{name} turns #{age} TODAY!!"
  else
    date = bday.strftime "%b %d"
    puts "#{name} will be #{age} on #{date}."
  end
end
```

```
Whose birthday would you like to know?
Other Chris Pine
Other Chris Pine will be 45 on Aug 03.
```

Exercises from Chapter 11

Extend the Built-in Classes

(Found on page 90.)

Extend some of Ruby's built-in classes with new methods.

How You Could Do It

```
Line 1  class Array
        -   def mult
        -     product = 1
        -
        5     self.each do |elem|
        -       product = product * elem
        -     end
        -
        -     product
       10   end
        - end
        -
        - puts [2,5,3].mult
```

```
❮ 30
```

How I Would Do It

```
Line 1  class Array
        2   def mult
        3     self.reduce(1, :*)
        4   end
        5 end
        6
        7 puts [2,5,3].mult
```

```
❮ 30
```

Orange Tree

(Found on page 99.)

Make an `OrangeTree` class that has a `height` method that returns its height and a `one_year_passes` method that, when called, ages the tree one year.

How You Could Do It

```
Line 1  class OrangeTree
        -   def initialize
        -     @height       = 0
        -     @orange_count = 0
        5     @alive        = true
```

```
  end

  def height
    if @alive
      @height
    else
      "A dead tree is not very tall. :("
    end
  end

  def count_the_oranges
    if @alive
      @orange_count
    else
      "A dead tree has no oranges. :("
    end
  end

  def one_year_passes
    if @alive
      @height = @height + 0.5
      @orange_count = 0 # old oranges fall off

      if @height > 10 && rand(2) > 0
        # tree dies
        @alive = false
        "Oh, no! The tree is too old, and has died. :("
      elsif @height > 2
        # new oranges grow
        @orange_count = (@height * 15 - 25).to_i
        "This year your tree grew to #{@height}m tall," +
        " and produced #{@orange_count} oranges."
      else
        "This year your tree grew to #{@height}m tall," +
        " but is still too young to bear fruit."
      end
    else
      "A year later, the tree is still dead. :("
    end
  end

  def pick_an_orange
    if @alive
      if @orange_count > 0
        @orange_count = @orange_count - 1
        "You pick a juicy, delicious orange!"
      else
        "You search every branch, but find no oranges."
      end
    else
      "A dead tree has nothing to pick. :("
```

```
    end
  end
end

ot = OrangeTree.new
18.times do
  ot.one_year_passes
end
puts(ot.one_year_passes)
puts(ot.count_the_oranges)
puts(ot.height)
puts(ot.one_year_passes)
puts(ot.one_year_passes)
puts(ot.one_year_passes)
puts(ot.height)
puts(ot.count_the_oranges)
puts(ot.pick_an_orange)
```

```
This year your tree grew to 9.5m tall, and produced 117 oranges.
117
9.5
This year your tree grew to 10.0m tall, and produced 125 oranges.
Oh, no! The tree is too old, and has died. :(
A year later, the tree is still dead. :(
A dead tree is not very tall. :(
A dead tree has no oranges. :(
A dead tree has nothing to pick. :(
```

How I Would Do It

I'd do it pretty much the same way you would: clean and simple.

Interactive Baby Dragon

(Found on page 99.)

Write a program that lets you enter commands such as `feed` and `walk` and calls those methods on your baby dragon.

How You Could Do It

```
# using the Dragon class from the chapter
puts "What would you like to name your baby dragon?"
name   = gets.chomp
dragon = Dragon.new(name)

while true
  puts
  puts "commands: feed, toss, walk, rock, put to bed, exit"
  command = gets.chomp

  if command == "exit"
```

```
    break
  elsif command == "feed"
    dragon.feed
  elsif command == "toss"
    dragon.toss
  elsif command == "walk"
    dragon.walk
  elsif command == "rock"
    dragon.rock
  elsif command == "put to bed"
    dragon.put_to_bed
  else
    puts "Huh? Please type one of the commands."
  end
end
```

How I Would Do It

```
# using the Dragon class from the chapter
puts "What would you like to name your baby dragon?"
name   = gets.chomp
dragon = Dragon.new(name)
object = Object.new # just a blank, dummy object

while true
  puts
  puts "commands: feed, toss, walk, rock, put_to_bed, exit"
  command = gets.chomp
  if command == "exit"
    break
  elsif dragon.respond_to?(command) && !object.respond_to?(command)
    # I only want to accept methods that dragons have,
    # but that regular objects *don't* have.
    dragon.send(command)
  else
    puts "Huh? Please type one of the commands."
  end
end
```

Exercises from Chapter 12

Trivia Program

(Found on page 106.)

Use the API for the Open Trivia Database to fetch trivia questions, and then ask the questions to the user. Count how many they get right, and tell them how they did at the end. (Make sure you don't ignore the response code that the API returns.)

How You Could Do It

```
Line 1  require "net/http"
        require "json"
        
        # make it easy to see how to increase the number of questions
        num_questions = 1
        
        url = URI("https://opentdb.com/api.php?amount=#{num_questions}")
        
        response = JSON.parse(Net::HTTP.get(url))
        
        if response["response_code"] != 0
          puts "Failed to fetch questions from server."
          puts "(response code: #{response["response_code"]})"
          exit
        end
        
        def ask_for_choice(answers)
          idx = 1
        
          answers.each do |a|
            puts "#{idx}. #{a}"
            idx = idx + 1
          end
        
          while true
            choice = gets.chomp.to_i
            if (choice >= 1 && choice <= answers.length)
              return choice
            else
              puts "Please pick a number between 1 and #{answers.length}."
            end
          end
        end
        
        num_correct = 0
        
        response["results"].each do |result|
          # Now we want to create an array of all answers,
          # with the correct answer randomly inserted. Here
          # is where we will store them.
          answers = []
        
          # index of the correct answer in the array of answers
          correct_idx = rand(result["incorrect_answers"].length + 1)
        
          # now build the `answers` array
          if correct_idx == 0
            answers.push(result["correct_answer"])
          end
        
          result["incorrect_answers"].each do |a|
```

```
    answers.push(a)

    if answers.length == correct_idx
      answers.push(result["correct_answer"])
    end
  end

  # ok, now we actually ask the question
  puts result["question"]

  # subtract 1 because array indexing starts at 0
  choice = ask_for_choice(answers)-1

  if choice == correct_idx
    num_correct = num_correct + 1
  end
end

puts "Thanks for playing!"
puts "Score: #{num_correct}/#{num_questions}"
```

```
California is larger than Japan.
1. True
2. False
1
Thanks for playing!
Score: 1/1
```

How I Would Do It

```
require "net/http"
require "json"

# to run this program and get 5 questions:
#   ruby trivia.rb 5
num_questions = ARGV[0].to_i
num_questions = 1 if num_questions < 1

url = URI("https://opentdb.com/api.php?amount=#{num_questions}")

response = JSON.parse(Net::HTTP.get(url))

if response["response_code"] != 0
  puts "Failed to fetch questions from server."
  puts "(response code: #{response["response_code"]})"
  exit
end

def ask_for_choice(answers)
  idx = 1

  answers.each do |a|
    puts "#{idx}. #{a}"
```

```
    idx += 1
  end

  while true
    choice = STDIN.gets.chomp.to_i
    if (choice >= 1 && choice <= answers.length)
      return choice
    else
      puts "Please pick a number between 1 and #{answers.length}."
    end
  end
end

num_correct = 0

response["results"].each do |result|
  # Now we want to create an array of all answers,
  # with the correct answer randomly inserted. Here
  # is where we will store them.
  answers = []
  if result["type"] == "boolean"
    # always ask true/false questions in this order
    answers = ["True", "False"]
  else
    # this is a multiple choice question
    answers =  result["incorrect_answers"].dup
    answers << result["correct_answer"]
    answers = answers.sort_by{rand} # shuffle answers
  end

  # ok, now we actually ask the question
  puts
  puts result["question"]
  puts

  # subtract 1 because array indexing starts at 0
  choice = ask_for_choice(answers)-1

  if answers[choice] == result["correct_answer"]
    num_correct += 1
  end
end

puts
puts "Thanks for playing!"
puts "Score: #{num_correct}/#{num_questions}"
```

```
❮

What was the first Disney movie to use CGI?

1. Fantasia
2. Tron
```

```
3. Toy Story
4. The Black Cauldron
⇒ moo
❮ Please pick a number between 1 and 4.
⇒ 2
❮
Thanks for playing!
Score: 0/1
```

Build Something Cool with an API

(Found on page 111.)

It's time for you to discover and use an API all on your own. Search online for an API that you find interesting, and use it to build something cool.

(Since the exercise is for you to come up with the idea on your own, I can't show you how you could do it or how I'd do it. But I hope you had fun building it!)

Exercises from Chapter 13

Even Better Profiling

(Found on page 120.)

Modify your profile method so you can turn all profiling on and off by changing only one word.

How You Could Do It

```
Line 1  def profile(block_description, &block)
        # To turn profiling on/off, set this to true/false.
        profiling_on = false

        if profiling_on
          start_time = Time.new
          block.call

          duration = Time.new - start_time
          puts "#{block_description}: #{duration} seconds"
        else
          block.call
        end
      end
```

How I Would Do It

```
Line 1  $OPT_PROFILING_ON = false

      - def profile(block_description, &block)
```

```
  if $OPT_PROFILING_ON
    start_time = Time.new
    block[]
    duration = Time.new - start_time
    puts "#{block_description}: #{duration} seconds"
  else
    block[]
  end
end
```

Grandfather Clock

(Found on page 120.)

Write a method that takes a block and calls it once for each hour that has passed today.

How You Could Do It

```
Line 1 def grandfather_clock(&block)
  hour = Time.new.hour

  if hour >= 13
    hour = hour - 12
  end

  if hour == 0
    hour = 12
  end

  hour.times do
    block.call
  end
end

grandfather_clock do
  puts "BONG!"
end
```

```
BONG!
BONG!
BONG!
```

How I Would Do It

```
Line 1 def grandfather_clock(&block)
  hour = (Time.new.hour + 11)%12 + 1

  hour.times(&block)
end

grandfather_clock { puts "BONG!" }
```

```
BONG!
BONG!
BONG!
```

Program Logger

(Found on page 120.)

Write a method called `log` that takes a string description of a block (and, of course, a block). It should `puts` a string telling you it started the block and another string at the end telling you that it finished and what the block returned.

How You Could Do It

```
def log(desc, &block)
  puts "Beginning '#{desc}'..."
  result = block.call
  puts "...'#{desc}' finished, returning: #{result}"
end

log("outer block") do
  log("some little block") do
    1**1 + 2**2
  end

  log("yet another block") do
    "!doof iahT ekil I".reverse
  end

  "0" == 0
end
```

```
Beginning 'outer block'...
Beginning 'some little block'...
...'some little block' finished, returning: 5
Beginning 'yet another block'...
...'yet another block' finished, returning: I like Thai food!
...'outer block' finished, returning: false
```

How I Would Do It

```
def log(desc, &block)
  puts "Beginning #{desc.inspect}..."
  result = block[]
  puts "...#{desc.inspect} finished, returning: #{result}"
end

log("outer block") do
  log("some little block") do
    1**1 + 2**2
  end
```

```
  log("yet another block") do
    "!doof iahT ekil I".reverse
  end

  "0" == 0
end
```

```
Beginning "outer block"...
Beginning "some little block"...
..."some little block" finished, returning: 5
Beginning "yet another block"...
..."yet another block" finished, returning: I like Thai food!
..."outer block" finished, returning: false
```

Better Program Logger

(Found on page 121.)

Improve your program logger with indentation for nested calls to the logger.

How You Could Do It

```
$logger_depth = 0

def log(desc, &block)
  prefix = "  "*$logger_depth

  puts prefix + "Beginning '#{desc}'..."

  $logger_depth = $logger_depth + 1
  result = block.call

  $logger_depth = $logger_depth - 1
  puts prefix + "...'#{desc}' finished, returning: #{result}"
end

log("outer block") do
  log("some little block") do
    log("teeny-tiny block") do
      "lOtS oF lOVe".downcase
    end

    7 * 3 * 2
  end

  log("yet another block") do
    "!doof naidnI evol I".reverse
  end

  "0" == 0
end
```

```
Beginning 'outer block'...
  Beginning 'some little block'...
    Beginning 'teeny-tiny block'...
    ...'teeny-tiny block' finished, returning: lots of love
  ...'some little block' finished, returning: 42
  Beginning 'yet another block'...
  ...'yet another block' finished, returning: I love Indian food!
...'outer block' finished, returning: false
```

How I Would Do It

```
$logger_depth = 0

def log(desc, &block)
  prefix = "  "*$logger_depth
  puts prefix+"Beginning #{desc.inspect}..."
  $logger_depth += 1
  result = block[]
  $logger_depth -= 1
  puts prefix+"...#{desc.inspect} finished, returning: #{result}"
end

log("outer block") do
  log("some little block") do
    log("teeny-tiny block") do
      "lOtS oF lOVe".downcase
    end

    7 * 3 * 2
  end

  log("yet another block") do
    "!doof naidnI evol I".reverse
  end

  "0" == '0'
end
```

```
Beginning "outer block"...
  Beginning "some little block"...
    Beginning "teeny-tiny block"...
    ..."teeny-tiny block" finished, returning: lots of love
  ..."some little block" finished, returning: 42
  Beginning "yet another block"...
  ..."yet another block" finished, returning: I love Indian food!
..."outer block" finished, returning: true
```

Exercises from Chapter 14

Rite of Passage: Sorting

(Found on page 130.)

Write a sort method that takes an array and returns a sorted version of it.

How You Could Do It

```
Line 1  def sort(arr)
        -   if arr.length < 1
        -     arr # an empty array is already sorted
        -   else
        5     pivot = arr.pop
        -
        -     smaller_elements = []
        -     larger_elements  = []
        -
       10     arr.each do |elem|
        -       if elem < pivot
        -         smaller_elements.push(elem)
        -       else
        -         larger_elements.push(elem)
       15       end
        -     end
        -
        -     smaller_sorted = sort(smaller_elements)
        -     larger_sorted  = sort(larger_elements )
       20
        -     smaller_sorted + [pivot] + larger_sorted
        -   end
        - end
        -
       25 puts(sort(["can","feel","singing","like","a","can"]))
```

```
a
can
can
feel
like
singing
```

How I Would Do It

```
Line 1  def sort(arr)
        -   return arr if arr.length < 2
        -
        -   pivot = arr.pop
        5   less  = arr.select{|x| x <  pivot}
```

```
  more  = arr.select{|x| x >= pivot}

  sort(less) + [pivot] + sort(more)
end

puts(sort(["can","feel","singing","like","a","can"]).join(" "))
```

```
a can can feel like singing
```

Shuffle

(Found on page 131.)

Write a shuffle method that takes an array and returns a totally shuffled version of it.

How You Could Do It

```
def shuffle(arr)
  shuf = []
  while arr.length > 0
    # Randomly pick one element of the array.
    rand_index = rand(arr.length)

    # Now go through each item in the array,
    # putting them all into new_arr except for the
    # randomly chosen one, which goes into shuf.
    curr_index = 0
    new_arr = []

    arr.each do |item|
      if curr_index == rand_index
        shuf.push(item)
      else
        new_arr.push(item)
      end

      curr_index = curr_index + 1
    end

    # Replace the original array with the new,
    # smaller array.
    arr = new_arr
  end

  shuf
end

puts(shuffle([0,1,2,3,4,5,6,7,8,9]).inspect)
```

```
[2, 6, 3, 8, 5, 9, 4, 0, 7, 1]
```

How I Would Do It

```
Line 1  def shuffle(arr)
        2    arr.sort_by{rand}
        3  end
        4
        5  p(shuffle([0,1,2,3,4,5,6,7,8,9]))
```

```
❮ [9, 3, 5, 7, 8, 1, 6, 2, 4, 0]
```

Dictionary Sort

(Found on page 131.)

Sort words like a dictionary would (without regard to case).

How You Could Do It

```
Line 1  def dictionary_sort(arr)
        -    if arr.length < 1
        -      arr # an empty array is already sorted
        -    else
        5      pivot = arr.pop
        -
        -      smaller_elements = []
        -      larger_elements  = []
        -
       10      arr.each do |elem|
        -        if elem.downcase < pivot.downcase
        -          smaller_elements.push(elem)
        -        else
        -          larger_elements.push(elem)
       15        end
        -      end
        -
        -      smaller_sorted = dictionary_sort(smaller_elements)
        -      larger_sorted  = dictionary_sort(larger_elements )
       20
        -      smaller_sorted + [pivot] + larger_sorted
        -    end
        -  end
        -
       25  puts(dictionary_sort(["can","feel","singing.","like","A","can"]))
```

```
❮ A
  can
  can
  feel
  like
  singing.
```

How I Would Do It

```
Line 1  def dictionary_sort(arr)
        -   return arr if arr.length < 2
        -
        -   pivot = arr.pop
        5   less  = arr.select{|x| x.downcase <  pivot.downcase}
        -   more  = arr.select{|x| x.downcase >= pivot.downcase}
        -
        -   dictionary_sort(less) + [pivot] + dictionary_sort(more)
        - end
       10
        - words = ["can","feel","singing.","like","A","can"]
        - puts(dictionary_sort(words).join(' '))
```

```
A can can feel like singing.
```

Expanded english_number

(Found on page 132.)

Write a method called english_number. It'll take an integer, like 22, and return the English version of it (in this case, the string "twenty-two").

How You Could Do It

```
Line 1  def english_number(number)
        -   if number < 0 # No negative numbers.
        -     return "Please enter a number that isn't negative."
        -   end
        5   if number == 0
        -     return "zero"
        -   end
        -
        -   # no more special cases, no more returns
       10   num_string = "" # This is the string we will return.
        -   ones_place = ["one",       "two",      "three",
        -                 "four",      "five",     "six",
        -                 "seven",     "eight",    "nine"]
        -   tens_place = ["ten",       "twenty",   "thirty",
       15                 "forty",     "fifty",    "sixty",
        -                 "seventy",   "eighty",   "ninety"]
        -   teenagers  = ["eleven",    "twelve",   "thirteen",
        -                 "fourteen",  "fifteen",  "sixteen",
        -                 "seventeen", "eighteen", "nineteen"]
       20
        -   zillions = [["hundred",           2],
        -               ["thousand",          3],
        -               ["million",           6],
        -               ["billion",           9],
       25               ["trillion",         12],
        -               ["quadrillion",      15],
        -               ["quintillion",      18],
```

```
              ["sextillion",        21],
              ["septillion",        24],
              ["octillion",         27],
              ["nonillion",         30],
              ["decillion",         33],
              ["undecillion",       36],
              ["duodecillion",      39],
              ["tredecillion",      42],
              ["quattuordecillion", 45],
              ["quindecillion",     48],
              ["sexdecillion",      51],
              ["septendecillion",   54],
              ["octodecillion",     57],
              ["novemdecillion",    60],
              ["vigintillion",      63],
              ["googol",           100]]

# `remaining` is how much of the number
#    we still have remaining to write out.
# `writing` is the part we are
#    writing out right now.
remaining = number

while zillions.length > 0
  zil_pair = zillions.pop
  zil_name =       zil_pair[0]
  zil_base = 10 ** zil_pair[1]
  writing = remaining/zil_base # How many zillions remaining?
  remaining = remaining - writing*zil_base # Subtract off those zillions.

  if writing > 0
    # Here's the recursion:
    prefix = english_number(writing)
    num_string = num_string + prefix + " " + zil_name

    if remaining > 0
      # So we don't write "two billionfifty-one"
      num_string = num_string + " "
    end
  end
end

# by this point, the number in `remaining` is less than 100

writing = remaining/10 # How many tens remaining?
remaining = remaining - writing*10 #  ubtract off those tens.

if writing > 0
  if ((writing == 1) and (remaining > 0))
    # Since we can't write "tenty-two" instead of
    # "twelve", we have to make a special exception
    # for these.
```

```
      num_string = num_string + teenagers[remaining-1]
      # The `-1` is because teenagers[3] is
      # "fourteen", not "thirteen".

      # Since we took care of the digit in the
      # ones place already, we have nothing remaining to write.
      remaining = 0
    else
      num_string = num_string + tens_place[writing-1]
      # The `-1` is because tens_place[3] is
      # "forty", not "thirty".
    end

    if remaining > 0
      # So we don't write "sixtyfour"
      num_string = num_string + "-"
    end
  end

  writing = remaining # How many ones remaining to write out?
  remaining = 0 # Subtract off those ones.

  if writing > 0
    num_string = num_string + ones_place[writing-1]
    # The `-1` is because ones_place[3] is
    # "four", not "three".
  end

  # Now we just return `num_string`
  num_string
end

puts english_number(  0)
puts english_number(  9)
puts english_number( 10)
puts english_number( 11)
puts english_number( 17)
puts english_number( 22)
puts english_number( 88)
puts english_number( 99)
puts english_number(100)
puts english_number(101)
puts english_number(234)
puts english_number(3211)
puts english_number(999999)
puts english_number(1000000000000)
puts english_number(109238745102938560129834709285360238475982374561034)
```

```
zero
nine
ten
eleven
```

```
seventeen
twenty-two
eighty-eight
ninety-nine
one hundred
one hundred one
two hundred thirty-four
three thousand two hundred eleven
nine hundred ninety-nine thousand nine hundred ninety-nine
one trillion
one hundred nine quindecillion two hundred thirty-eight
  quattuordecillion seven hundred forty-five tredecillion...
```

How I Would Do It

I'd do it pretty much the way you would.

"Ninety-nine Bottles of Beer on the Wall"

(Found on page 132.)

Write out the lyrics to "Ninety-nine Bottles of Beer on the Wall" using the English names of the numbers instead of numerals.

How You Could Do It

```
# english_number as above, plus this:

num_at_start = 5 # change to 9999 if you want
num_now = num_at_start

while num_now > 2
  puts english_number(num_now).capitalize +
    " bottles of beer on the wall, " +
    english_number(num_now) + " bottles of beer!"
  num_now = num_now - 1
  puts "Take one down, pass it around, " +
    english_number(num_now) + " bottles of beer on the wall!"
end

puts "Two bottles of beer on the wall, two bottles of beer!"
puts "Take one down, pass it around, one bottle of beer on the wall!"
puts "One bottle of beer on the wall, one bottle of beer!"
puts "Take one down, pass it around, no more bottles of beer on the wall!"
```

```
Five bottles of beer on the wall, five bottles of beer!
Take one down, pass it around, four bottles of beer on the wall!
Four bottles of beer on the wall, four bottles of beer!
Take one down, pass it around, three bottles of beer on the wall!
Three bottles of beer on the wall, three bottles of beer!
Take one down, pass it around, two bottles of beer on the wall!
Two bottles of beer on the wall, two bottles of beer!
```

```
Take one down, pass it around, one bottle of beer on the wall!
One bottle of beer on the wall, one bottle of beer!
Take one down, pass it around, no more bottles of beer on the wall!
```

How I Would Do It

```
# english_number as above, plus this:

num_at_start = 5  #  change to 9999 if you want

num_bot = proc { |n| "#{english_number(n)} bottle#{n == 1 ? "" : "s"}" }

num_at_start.downto(2) do |num|
  puts "#{num_bot[num]} of beer on the wall, #{num_bot[num]} of beer!".capitalize
  puts "Take one down, pass it around, #{num_bot[num-1]} of beer on the wall!"
end
puts "#{num_bot[1]} of beer on the wall, #{num_bot[1]} of beer!".capitalize
puts "Take one down, pass it around, no more bottles of beer on the wall!"
```

```
Five bottles of beer on the wall, five bottles of beer!
Take one down, pass it around, four bottles of beer on the wall!
Four bottles of beer on the wall, four bottles of beer!
Take one down, pass it around, three bottles of beer on the wall!
Three bottles of beer on the wall, three bottles of beer!
Take one down, pass it around, two bottles of beer on the wall!
Two bottles of beer on the wall, two bottles of beer!
Take one down, pass it around, one bottle of beer on the wall!
One bottle of beer on the wall, one bottle of beer!
Take one down, pass it around, no more bottles of beer on the wall!
```

Index

F

G

H

I

J

T

U

V

W

Y

Z

Thank you!

How did you enjoy this book? Please let us know. Take a moment and email us at support@pragprog.com with your feedback. Tell us your story and you could win free ebooks. Please use the subject line "Book Feedback."

Ready for your next great Pragmatic Bookshelf book? Come on over to https://pragprog.com and use the coupon code BUYANOTHER2021 to save 30% on your next ebook.

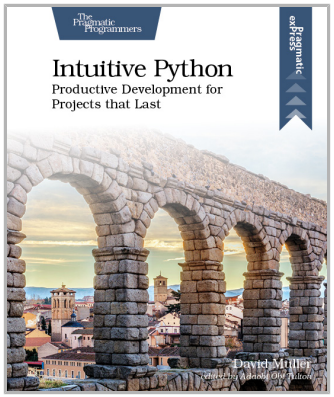
Void where prohibited, restricted, or otherwise unwelcome. Do not use ebooks near water. If rash persists, see a doctor. Doesn't apply to *The Pragmatic Programmer* ebook because it's older than the Pragmatic Bookshelf itself. Side effects may include increased knowledge and skill, increased marketability, and deep satisfaction. Increase dosage regularly.

And thank you for your continued support,

Andy Hunt, Publisher

Intuitive Python

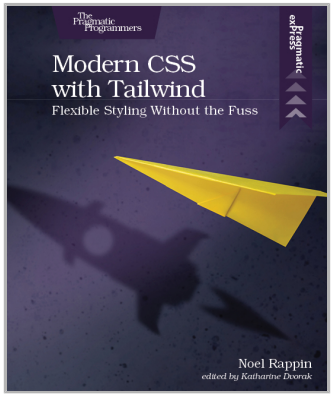
Developers power their projects with Python because it emphasizes readability, ease of use, and access to a meticulously maintained set of packages and tools. The language itself continues to improve with every release: writing in Python is full of possibility. But to maintain a successful Python project, you need to know more than just the language. You need tooling and instincts to help you make the most out of what's available to you. Use this book as your guide to help you hone your skills and sculpt a Python project that can stand the test of time.



David Muller
(140 pages) ISBN: 9781680508239. $26.95
https://pragprog.com/book/dmpython

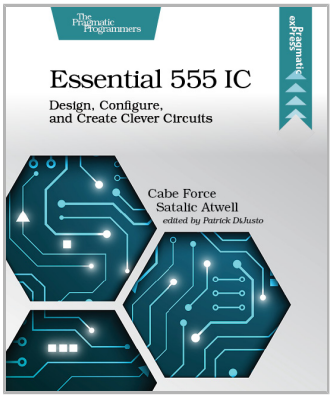
Modern CSS with Tailwind

Tailwind CSS is an exciting new CSS framework that allows you to design your site by composing simple utility classes to create complex effects. With Tailwind, you can style your text, move your items on the page, design complex page layouts, and adapt your design for devices from a phone to a wide-screen monitor. With this book, you'll learn how to use the Tailwind for its flexibility and its consistency, from the smallest detail of your typography to the entire design of your site.



Noel Rappin
(90 pages) ISBN: 9781680508185. $26.95
https://pragprog.com/book/tailwind

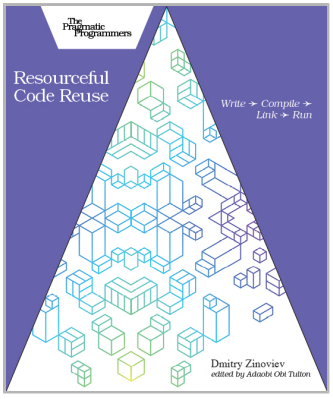
Essential 555 IC



Learn how to create functional gadgets using simple but clever circuits based on the venerable "555." These projects will give you hands-on experience with useful, basic circuits that will aid you across other projects. These inspiring designs might even lead you to develop the next big thing. The 555 Timer Oscillator Integrated Circuit chip is one of the most popular chips in the world. Through clever projects, you will gain permanent knowledge of how to use the 555 timer will carry with you for life.

Cabe Force Satalic Atwell
(104 pages) ISBN: 9781680507836. $19.95
https://pragprog.com/book/catimers

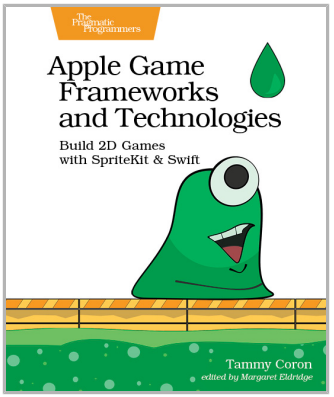
Resourceful Code Reuse



Reusing well-written, well-debugged, and well-tested code improves productivity, code quality, and software configurability and relieves pressure on software developers. When you organize your code into self-contained modular units, you can use them as building blocks for your future projects and share them with other programmers, if needed. Understand the benefits and downsides of seven code reuse models so you can confidently reuse code at any development stage. Create static and dynamic libraries in C and Python, two of the most popular modern programming languages. Adapt your code for the real world: deploy shared functions remotely and build software that accesses them using remote procedure calls.

Dmitry Zinoviev
(64 pages) ISBN: 9781680508208. $14.99
https://pragprog.com/book/dzreuse

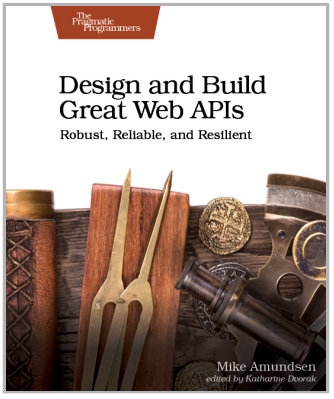
Apple Game Frameworks and Technologies



Design and develop sophisticated 2D games that are as much fun to make as they are to play. From particle effects and pathfinding to social integration and monetization, this complete tour of Apple's powerful suite of game technologies covers it all. Familiar with Swift but new to game development? No problem. Start with the basics and then layer in the complexity as you work your way through three exciting—and fully playable—games. In the end, you'll know everything you need to go off and create your own video game masterpiece for any Apple platform.

Tammy Coron
(504 pages) ISBN: 9781680507843. $51.95
https://pragprog.com/book/tcswift

Design and Build Great Web APIs



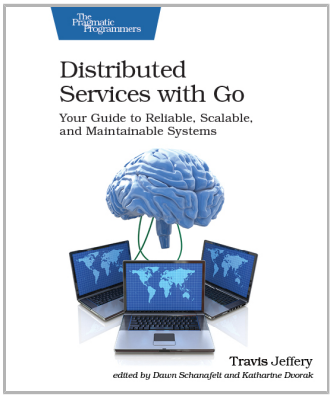
APIs are transforming the business world at an increasing pace. Gain the essential skills needed to quickly design, build, and deploy quality web APIs that are robust, reliable, and resilient. Go from initial design through prototyping and implementation to deployment of mission-critical APIs for your organization. Test, secure, and deploy your API with confidence and avoid the "release into production" panic. Tackle just about any API challenge with more than a dozen open-source utilities and common programming patterns you can apply right away.

Mike Amundsen
(330 pages) ISBN: 9781680506808. $45.95
https://pragprog.com/book/maapis

Distributed Services with Go

This is the book for Gophers who want to learn how to build distributed systems. You know the basics of Go and are eager to put your knowledge to work. Build distributed services that are highly available, resilient, and scalable. This book is just what you need to apply Go to real-world situations. Level up your engineering skills today.

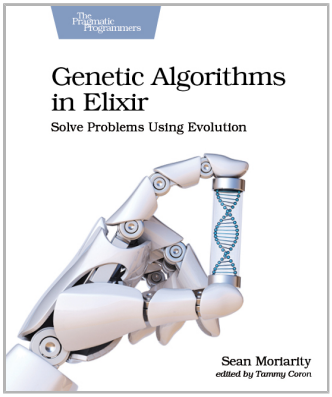
Travis Jeffery
(258 pages) ISBN: 9781680507607. $45.95
https://pragprog.com/book/tjgo



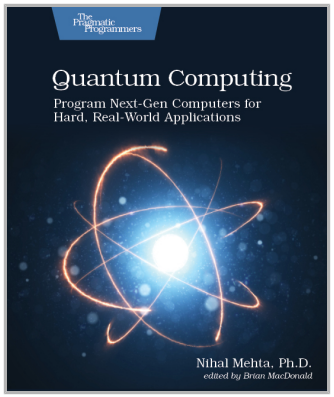
Genetic Algorithms in Elixir

From finance to artificial intelligence, genetic algorithms are a powerful tool with a wide array of applications. But you don't need an exotic new language or framework to get started; you can learn about genetic algorithms in a language you're already familiar with. Join us for an in-depth look at the algorithms, techniques, and methods that go into writing a genetic algorithm. From introductory problems to real-world applications, you'll learn the underlying principles of problem solving using genetic algorithms.

Sean Moriarity
(242 pages) ISBN: 9781680507942. $39.95
https://pragprog.com/book/smgaelixir



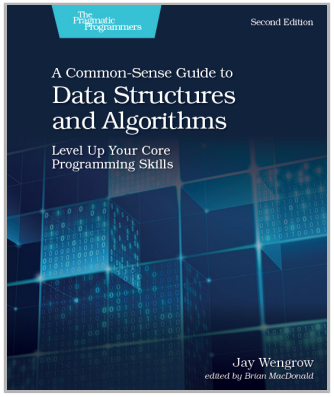
Quantum Computing



You've heard that quantum computing is going to change the world. Now you can check it out for yourself. Learn how quantum computing works, and write programs that run on the IBM Q quantum computer, one of the world's first functioning quantum computers. Develop your intuition to apply quantum concepts for challenging computational tasks. Write programs to trigger quantum effects and speed up finding the right solution for your problem. Get your hands on the future of computing today.

Nihal Mehta, Ph.D.
(580 pages) ISBN: 9781680507201. $45.95
https://pragprog.com/book/nmquantum

A Common-Sense Guide to Data Structures and Algorithms, Second Edition



If you thought that data structures and algorithms were all just theory, you're missing out on what they can do for your code. Learn to use Big O notation to make your code run faster by orders of magnitude. Choose from data structures such as hash tables, trees, and graphs to increase your code's efficiency exponentially. With simple language and clear diagrams, this book makes this complex topic accessible, no matter your background. This new edition features practice exercises in every chapter, and new chapters on topics such as dynamic programming and heaps and tries. Get the hands-on info you need to master data structures and algorithms for your day-to-day work.

Jay Wengrow
(506 pages) ISBN: 9781680507225. $45.95
https://pragprog.com/book/jwdsal2

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by professional developers for professional developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page
https://pragprog.com/book/ltp3
Source code from this book, errata, and other resources. Come give us feedback, too!

Keep Up to Date
https://pragprog.com
Join our announcement mailing list (low volume) or follow us on twitter @pragprog for new titles, sales, coupons, hot tips, and more.

New and Noteworthy
https://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this ebook, perhaps you'd like to have a paper copy of the book. Paperbacks are available from your local independent bookstore and wherever fine books are sold.

Contact Us

Online Orders:	*https://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://write-for-us.pragprog.com*
Or Call:	+1 800-699-7764