Forward

A Place to Start for the Future Programmer

I guess this all began back in 2002. I was thinking about teaching programming, and what a great language Ruby would be for learning how to program. I mean, we were all excited about Ruby because it was powerful, elegant, and really just fun, but it seemed to me that it would also be a great way to get into programming in the first place.

Unfortunately, there wasn't much Ruby documentation geared for newbies at the time. Some of us in the community were talking about what such a "Ruby for the Nuby" tutorial would need, and more generally, how to teach programming at all. The more I thought about this, the more I had to say (which surprised me a bit). Finally, someone said, "Chris, why don't you just write a tutorial instead of talking about it?" So I did.

And it wasn't very good. I had all these ideas that were good in theory, but the actual task of making a great tutorial for non-programmers was vastly more challenging than I had realized. (I mean, it seemed good to me, but I already knew how to program.)

What saved me was that I made it really easy for people to contact me, and I always tried to help people when they got stuck. When I saw a lot of people getting stuck in one place, I'd rewrite it. It was a lot of work, but it slowly got better and better.

A couple of years later, it was getting pretty good. 😃 So good, in fact, that I was ready to pronounce it finished, and move on to something else. And right about then came an opportunity to turn the tutorial into a book. Since it was already basically done, I figured this would be no problem. I'd just clean up a few spots, add some more exercises, maybe some more examples, a few more chapters, run it by 50 more reviewers...

It took me another year, but now I think it's really really good, mostly because of the hundreds of brave souls who have helped me write it.

What's here on this site is the original tutorial, more or less unchanged since 2004. For the latest and greatest, you'll want to check out the book.

Thoughts For Teachers

There were a few guiding principles that I tried to stick to. I think they make the learning process much smoother; learning to program is hard enough as it is. If you're teaching or guiding someone on the road to hackerdom, these ideas might help you, too.

First, I tried to separate concepts as much as possible, so that the student would only have to learn one concept at a time. This was difficult at first, but a little too easy after I had some practice. Some things must be taught before others, but I was amazed at how little of a precedence hierarchy there really is. Eventually, I just had to pick an order, and I tried to arrange things so that each new section was motivated by the previous ones.

Another principle I've kept in mind is to teach only one way to do something. It's an obvious benefit in a tutorial for people who have never programmed before. For one thing, one way to do something is easier to learn than two. Perhaps the more important benefit, though, is that the fewer things you teach a new programmer, the more creative and clever they have to be in their programming. Since so much of programming is problem solving, it's crucial to encourage that as much as possible at every stage.

I have tried to piggy-back programming concepts onto concepts the new programmer already has; to present ideas in such a way that their intuition will carry the load, rather than the tutorial. Object-Oriented programming lends itself to this quite well. I was able to begin referring to "objects" and different "kinds of objects" pretty early in the tutorial, slipping those phrases in at the most innocent of moments. I wasn't saying anything like "everything in Ruby is an object," or "numbers and strings are kinds of objects," because these statements really don't mean anything to a new programmer. Instead, I would talk about strings (not "string objects"), and sometimes I would refer to "objects", simply meaning "the things in these programs." The fact that all these things in Ruby are objects made this sort of sneakiness on my part work so well.

Although I wanted to avoid needless OO jargon, I wanted to make sure that, if they did need to learn a word, they learned the right one. (I don't want them to have to learn it twice, right?) So I called them "strings," not "text." Methods needed to be called something, so I called them "methods."

As far as the exercises are concerned, I think I came up with some good ones, but you can never have too many. Honestly, I bet I spent half of my time just trying to come up with fun, interesting exercises. Boring exercises absolutely kill any desire to program, while the perfect exercise creates an itch the new programmer can't help but scratch. In short, you just can't spend too much time coming up with good exercises.

About the Original Tutorial

The pages of the tutorial (and even this page) are generated by a big Ruby program, of course. 😃 All of the code samples were automatically run, and the output shown is the output they generated. I think this is the best, easiest, and certainly the coolest way to make sure that all of the code I present works exactly as I say it does. You don't have to worry that I might have copied the output of one of the examples wrong, or forgotten to test some of the code; it's all been tested.

powered by Ruby

Acknowledgements

Finally, I'd like to thank everyone on the ruby-talk mailing list for their thoughts and encouragement, all of my wonderful reviewers for their help in making the book far better than I could have alone, my dear wife especially for being my main reviewer/tester/guinea-pig/muse, Matz for creating this fabulous language, and the Pragmatic Programmers for telling me about it and, of course, for publishing my book!

If you notice any errors or typos, or have any comments or suggestions or good exercises I could include, please let me know.

Getting Started

Chapter 0

When you program a computer, you have to "speak" in a language your computer understands: a programming language. There are lots and lots of different languages out there, and many of them are excellent. In this tutorial I chose to use my favorite programming language, Ruby.

Aside from being my favorite, Ruby is also the easiest programming language I have seen (and I've seen quite a few). In fact, that's the real reason I'm writing this tutorial: I didn't decide to write a tutorial, and then choose Ruby because it's my favorite; instead, I found Ruby to be so easy that I decided there really ought to be a good beginner's tutorial which uses it. It's Ruby's simplicity which prompted this tutorial, not the fact that it's my favorite. (Writing a similar tutorial using another language, like C++ or Java, would have required hundreds and hundreds of pages.) But don't think that Ruby is a beginner's language just because it is easy! It is a powerful, professional-strength programming language if ever there was one.

When you write something in a human language, what is written is called text. When you write something in a computer language, what is written is called code. I have included lots of examples of Ruby code throughout this tutorial, most of them complete programs you can run on your own computer. To make the code easier to read, I have colored parts of the code different colors. (For example, numbers are always green.) Anything you are supposed to type in will be in a dotted box, and anything a program prints out will be in a grey box.

If you come across something you don't understand, or you have a question which wasn't answered, write it down and keep reading! It's quite possible that the answer will come in a later chapter. However, if your question was not answered by the last chapter, I will tell you where you can go to ask it. There are lots of wonderful people out there more than willing to help; you just need to know where they are.

But first we need to download and install Ruby onto your computer.

Windows Installation

The Windows installation of Ruby is a breeze. First, you need to download Ruby. There might be a couple of versions to choose from; this tutorial is using version 2.2.3, so make sure what you download is at least as recent as that. (I would just get the latest version available.) Then simply run the installation program. It will ask you where you want to install Ruby. Unless you have a good reason for it, I would just install it in the default location.

In order to program, you need to be able to write programs and to run programs. To do this, you will need a text editor and a command line. My favorite text editor is Sublime Text.

It would also be a good idea to create a folder somewhere to keep all of your programs. Make sure that when you save a program, you save it into this folder.

To get to your command line, select Command Prompt from the Accessories folder in your start menu. You will want to navigate to the folder where you are keeping your programs. Typing cd .. will take you up one folder, and cd foldername would put you inside the folder named foldername. To see all of the folders in your current folder, type dir /ad.

And that's it! You're all set to learn to program.

Macintosh Installation

If you have Mac OS X 10.2 (Jaguar) or later, then you already have Ruby on your system! What could be easier?

In order to program, you need to be able to write programs and to run programs. To do this, you will need a text editor and a command line.

Your command line is accessible through the Terminal application (found in Applications/Utilities).

For a text editor, you can use whatever one you are familiar or comfortable with. My favorite text editor is Sublime Text. If you use TextEdit, however, make sure you save your programs as text-only! Otherwise your programs will not work.

And that's it! You're all set to learn to program.

Linux Installation

First, you will want to check and see if you have Ruby installed already. Type which ruby. If it says something like /usr/bin/which: no ruby in (...), then you need to download Ruby, otherwise see what version of Ruby you are running with ruby -v. If it is older than the latest stable build on the above download page, you might want to upgrade.

If you are the root user, then you probably don't need any instructions for installing Ruby. If you aren't, you might want to ask your system administrator to install it for you. (That way everyone on that system could use Ruby.)

Otherwise, you can just install it so that only you can use it. Move the file you downloaded to a temporary directory, like $HOME/tmp. If the name of the file is ruby-1.6.7.tar.gz, you can open it with tar zxvf ruby-1.6.7.tar.gz. Change directory to the directory you just created (in this example, cd ruby-1.6.7).

Configure your installation by typing ./configure --prefix=$HOME). Next type make, which will build your Ruby interpreter. This might take a few minutes. After that is done, type make install to install it.

Next, you'll want to add $HOME/bin to your command search path by editing your $HOME/.bashrc file. (You might have to log out and back in again for this to take effect.) After you do that, test your installation: ruby -v. If that tells you what version of Ruby you have, you can now delete the files in $HOME/tmp (or wherever you put them).

And that's it! You're all set to learn to program.

Numbers

Chapter 1

Now that you've gotten everything setup, let's write a program! Open up your favorite text editor and type in the following:

```
puts 1 + 2
```

Save your program (yes, that's a program!) as calc.rb (the .rb is what we usually put at the end of programs written in Ruby). Now run your program by typing ruby calc.rb into your command line. It should have put a 3 on your screen. See, programming isn't so hard, now is it?

Introduction to puts

So what's going on in that program? I'm sure you can guess what the 1+2 does; our program is basically the same as:

```
puts 3
```

puts simply writes onto the screen whatever comes after it.

Integer and Float

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

Here are some integers:

```
5
-205
9999999999999999999999999
0
```

And here are some floats:

```
54.321
0.001
-205.3884
0.0
```

In practice, most programs don't use floats; only integers. (After all, no one wants to look at 7.4 emails, or browse 1.8 webpages, or listen to 5.24 of their favorite songs...) Floats are used more for academic purposes (physics experiments and such) and for 3D graphics. Even most money programs use integers; they just keep track of the number of pennies!

Simple Arithmetic

So far, we've got all the makings of a simple calculator. (Calculators always use floats, so if you want your computer to act just like a calculator, you should also use floats.) For addition and subtraction, we use + and -, as we saw. For multiplication, we use *, and for division we use /. Most keyboards have these keys in the numeric keypad on the far right side. If you have a smaller keyboard or a laptop, though, you can just use Shift 8 and / (same key as the ? key). Let's try to expand our calc.rb program a little. Type in the following and then run it.

```
puts 1.0 + 2.0
puts 2.0 * 3.0
puts 5.0 - 8.0
puts 9.0 / 2.0
```

This is what the program returns:

```
3.0
6.0
-3.0
4.5
```

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

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

Mostly the same, right?

```
3
6
-3
4
```

Uh... except for that last one! But when you do arithmetic with integers, you'll get integer answers. When your computer can't get the "right" answer, it always rounds down. (Of course, 4 is the right answer in integer arithmetic for 9/2; just maybe not the answer you were expecting.)

Perhaps you're wondering what integer division is good for. Well, let's say you're going to the movies, but you only have $9. Here in Portland, you can see a movie at the Bagdad for 2 bucks. How many movies can you see there? 9/2... 4 movies. 4.5 is definitely not the right answer in this case; they will not let you watch half of a movie, or let half of you in to see a whole movie... some things just aren't divisible.

So now experiment with some programs of your own! If you want to write more complex expressions, you can use parentheses. For example:

```
puts 5 * (12 - 8) + -15
puts 98 + (59872 / (13 * 8)) * -52
```

```
5
-29802
```

A Few Things to Try

Write a program which tells you:

how many hours are in a year? how many minutes are in a decade? how many seconds old are you? how many chocolates do you hope to eat in your life? Warning: This part of the program could take a while to compute! Here's a tougher question:

If I am 1245 million seconds old, how old am I? When you're done playing around with numbers, let's have a look at some letters.

Letters

Chapter 2

So we've learned all about numbers, but what about letters? words? text?

We refer to groups of letters in a program as strings. (You can think of printed letters being strung together on a banner.) To make it easier to see just what part of the code is in a string, I'll color strings red. Here are some strings:

```
'Hello.'
'Ruby rocks.'
'5 is my favorite number... what is yours?'
'Snoopy says #%^?&*@! when he stubs his toe.'
'     '
''
```

As you can see, strings can have punctuation, digits, symbols, and spaces in them... more than just letters. That last string doesn't have anything in it at all; we would call that an empty string.

We have been using puts to print numbers; let's try it with some strings:

```
puts 'Hello, world!'
puts ''
puts 'Good-bye.'
```

```
Hello, world!

Good-bye.
```

That worked out well. Now try some strings of your own.

String Arithmetic

Just as you can do arithmetic on numbers, you can also do arithmetic on strings! Well, sort of... you can add strings, anyway. Let's try to add two strings and see what puts does with that.

```
puts 'I like' + 'apple pie.'
```

```
I likeapple pie.
```

Whoops! I forgot to put a space between 'I like' and 'apple pie.'. Spaces don't matter usually, but they matter inside strings. (It's true what they say: computers don't do what you want them to do, only what you tell them to do.) Let's try that again:

```
puts 'I like ' + 'apple pie.'
puts 'I like' + ' apple pie.'
```

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

(As you can see, it didn't matter which string I added the space to.)

So you can add strings, but you can also multiply them! (By a number, anyway.) Watch this:

```
puts 'blink ' * 4
```

```
batting her eyes
```

(Just kidding... it really does this:)

```
blink blink blink blink
```

If you think about it, this makes perfect sense. After all, 7*3 really just means 7+7+7, so 'moo'*3 just means 'moo'+'moo'+'moo'.

12 vs '12'

Before we get any further, we should make sure we understand the difference between numbers and digits. 12 is a number, but '12' is a string of two digits.

Let's play around with this for a while:

```
puts  12  +  12
puts '12' + '12'
puts '12  +  12'
```

```
24
1212
12  +  12
```

How about this:

```
puts  2  *  5
puts '2' *  5
puts '2  *  5'
```

```
10
22222
```

```
2 * 5
```

These examples were pretty straightforward. However, if you're not too careful with how you mix your strings and your numbers, you might run into...

Problems

At this point you may have tried out a few things which didn't work. If not, here are a few:

```
puts '12' + 12
```

```
TypeError: no implicit conversion of Fixnum into String
```

```
puts '2' * '5'
```

```
TypeError: no implicit conversion of String into Integer
```

Hmmm... an error message. The problem is that you can't really add a number to a string, or multiply a string by another string. It doesn't make any more sense than does this:

```
puts 'Betty' + 12
puts 'Fred' * 'John'
```

Something else to be aware of: you can write 'pig'*5 in a program, since it just means 5 sets of the string 'pig' all added together. However, you can't write 5*'pig', since that means 'pig' sets of the number 5, which is just silly.

Finally, what if I want a program to print out You're swell!? We can try this:

```
irb(main):001:0> puts 'You're swell!'
irb(main):002:0' '
SyntaxError: (irb):1: syntax error, unexpected tIDENTIFIER, expecting end-of-input
puts 'You're swell!'
            ^
```

Well, that won't work; I won't even try to run it. The computer thought we were done with the string. (This is why it's nice to have a text editor which does syntax coloring for you.) So how do we let the computer know we want to stay in the string? We have to escape the apostrophe, like this:

```
puts 'You\'re swell!'
```

```
You're swell!
```

The backslash is the escape character. In other words, if you have a backslash and another character, they are sometimes translated into a new character. The only things the backslash escapes, though, are the apostrophe and the backslash itself. (If you think about it, escape characters must always escape themselves.) A few examples are in order here, I think:

```
puts 'You\'re swell!'
puts 'backslash at the end of a string:  \\'
puts 'up\\down'
puts 'up\down'
```

```
You're swell!
backslash at the end of a string:  \
up\down
up\down
```

Since the backslash does not escape a 'd', but does escape itself, those last two strings are identical. They don't look the same in the code, but in your computer they really are the same.

If you have any other questions, just keep reading! I couldn't answer every question on this page, after all.

Variables and Assignment

Chapter 3

So far, whenever we have put a string or a number, the thing we put is gone. What I mean is, if we wanted to print something out twice, we would have to type it in twice:

```
puts '...you can say that again...'
puts '...you can say that again...'
```

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

It would be nice if we could just type it in once and then hang on to it... store it somewhere. Well, we can, of course otherwise, I wouldn't have brought it up!

To store the string in your computer's memory, we need to give the string a name. Programmers often refer to this process as assignment, and they call the names variables. This variable can be just about any sequence of letters and numbers, but the first character needs to be a lowercase letter. Let's try that last program again, but this time I will give the string the name myString (though I could just as well have named it str or myOwnLittleString or henryTheEighth).

```
myString = '...you can say that again...'
puts myString
puts myString
```

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

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

```
name = 'Patricia Rosanna Jessica Mildred Oppenheimer'
puts 'My name is ' + name + '.'
puts 'Wow!  ' + name + ' is a really long name!'
```

```
My name is Patricia Rosanna Jessica Mildred Oppenheimer.
Wow!  Patricia Rosanna Jessica Mildred Oppenheimer is a really long name!
```

Also, just as we can assign an object to a variable, we can reassign a different object to that variable. (This is why we call them variables: because what they point to can vary.)

```
composer = 'Mozart'
puts composer + ' was "da bomb", in his day.'
```

```
Mozart was "da bomb", in his day.
```

```
composer = 'Beethoven'
puts 'But I prefer ' + composer + ', personally.'
```

```
But I prefer Beethoven, personally.
```

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

```
var = 'just another ' + 'string'
puts var
```

```
just another string
```

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

```
15
```

In fact, variables can point to just about anything... except other variables. So what happens if we try?

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

puts ''

var1 = 'eight'
puts var1
puts var2
```

```
8
8

eight
8
```

So first, when we tried to point var2 to var1, it really pointed to 8 instead (just like var1 was pointing to). Then we had var1 point to 'eight', but since var2 was never really pointing at var1, it stays pointing at 8.

So now that we've got variables, numbers, and strings, let's learn how to mix them all up!

Mixing It Up

Chapter 4

We've looked at a few different kinds of objects (numbers and letters), and we made variables to point to them; the next thing we want to do is to get them all to play nicely together.

We've seen that if we want a program to print 25, the following does not work, because you can't add numbers and strings:

```
var1 = 2
var2 = '5'

puts var1 + var2
```

Part of the problem is that your computer doesn't know if you were trying to get 7 (2 + 5), or if you wanted to get 25 ('2' + '5').

Before we can add these together, we need some way of getting the string version of var1, or to get the integer version of var2.

Conversions

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

```
var1 = 2
var2 = '5'

puts var1.to_s + var2
```

```
25
```

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

```
var1 = 2
var2 = '5'

puts var1.to_s + var2
puts var1 + var2.to_i
```

```
25
7
```

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

Now let's try some more interesting (and a few just weird) conversions:

```
puts '15'.to_f
puts '99.999'.to_f
puts '99.999'.to_i
puts ''
puts '5 is my favorite number!'.to_i
puts 'Who asked you about 5 or whatever?'.to_i
puts 'Your momma did.'.to_f
puts ''
puts 'stringy'.to_s
puts 3.to_i
```

```
15.0
99.999
99

5
0
0.0

stringy
3
```

So, this probably gave some surprises. The first one is pretty standard, giving 15.0. After that, we converted the string '99.999' to a float and to an integer. The float did what we expected; the integer was, as always, rounded down.

Next, we had some examples of some... unusual strings being converted into numbers. to_i ignores the first thing it doesn't understand, and the rest of the string from that point on. So the first one was converted to 5, but the others, since they started with letters, were ignored completely... so the computer just picks zero.

Finally, we saw that our last two conversions did nothing at all, just as we would expect.

Another Look at puts

There's something strange about our favorite method... Take a look at this:

```
puts 20
puts 20.to_s
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 one, the integer 20? For that matter, what does it even mean to write out the integer 20? When you write a 2 and then a 0 on a piece of paper, you are writing down 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 our friend, puts: 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 really means put string.

This may not seem too exciting now, but there are many, many kinds of objects in Ruby (you'll even learn how to make your own!), and it's nice to know what will happen if you try to puts a really weird object, like a picture of your grandmother, or a music file or something. But that will come later...

In the meantime, we have a few more methods for you, and they allow us to write all sorts of fun programs...

The Methods gets and chomp

If puts means put string, I'm sure you can guess what gets stands for. And just as puts always spits out strings, gets will only retrieve strings. And whence does it get them?

From you! Well, from your keyboard, anyway. Since your keyboard only makes strings, that works out beautifully. What actually happens is that gets just sits there, reading what you type until you press Enter. Let's try it out:

```
puts gets
```

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

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

Of course, whatever you type in will just get repeated back to you. Run it a few times and try typing in different things.

Now we can make interactive programs! In this one, type in your name and it will greet you:

```
puts 'Hello there, and what\'s your name?'
name = gets
puts 'Your name is ' + name + '?  What a lovely name!'
puts 'Pleased to meet you, ' + name + '.  :)'
```

Eek! I just ran it—I typed in my name, 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
.  :)
```

Hmmm... it looks like when I typed in the letters C, h, r, i, s, and then pressed Enter, gets got all of the letters in my name and the Enter! Fortunately, there's a method just for this sort of thing: chomp. It takes off any Enters hanging out at the end of your string. Let's try that program again, but with chomp to help us this time:

```
puts 'Hello there, and what\'s your name?'
name = gets.chomp
puts 'Your name is ' + name + '?  What a lovely name!'
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, we don't ever have to say name.chomp; name was already chomped.

A Few Things to Try

Write a program which asks for a person's first name, then middle, then last. Finally, it should greet the person using their full name.

Write a program which asks for a person's favorite number. Have your program add one to the number, then suggest the result as a bigger and better favorite number. (Do be tactful about it, though.)

Once you have finished those two programs (and any others you would like to try), let's learn some more (and some more about) methods.

More About Methods

Chapter 5

So far we've seen a number of different methods, puts and gets and so on (Pop Quiz: List all of the methods we have seen so far! There are ten of them; the answer is below.), but we haven't really talked about what methods are. We know what they do, but we don't know what they are.

But really, that is what they are: things that do stuff. If objects (like strings, integers, and floats) are the nouns in the Ruby language, then methods are like the verbs. And, just like in English, you can't have a verb without a noun to do the verb. For example, ticking isn't something that just happens; a clock (or a watch or something) has to do it. In English we would say, "The clock ticks." In Ruby we would say clock.tick (assuming that clock was a Ruby object, of course). Programmers might say we were "calling clock's tick method," or that we "called tick on clock."

So, did you take the quiz? Good. Well, I'm sure you remembered the methods puts, gets, and chomp, since we just covered those. You probably also got our conversion methods, to_i, to_f, and to_s. However, did you get the other four? Why, it's none other than our old arithmetic buddies +, -, *, and /!

So as I was saying, just as every verb needs a noun, so every method needs an object. It's usually easy to tell which object is performing the method: it's what comes right before the dot, like in our clock.tick example, or in 101.to_s. Sometimes, however, it's not quite as obvious; like with the arithmetic methods. As it turns out, 5 + 5 is really just a shortcut way of writing 5.+ 5. For example:

```
puts 'hello '.+ 'world'
puts (10.* 9).+ 9
```

```
hello world
99
```

It isn't very pretty, so we won't ever write it like that; however, it's important to understand what is really happening. (On older versions of Ruby, this code might also give a warning: warning: parenthesize argument(s) for future version. It would still run the code just fine, though.) This also gives us a deeper understanding of why we can do 'pig' * 5 but we can't do 5 * 'pig': 'pig' * 5 is telling 'pig' to do the multiplying, but 5 * 'pig' is telling 5 to do the multiplying. 'pig' knows how to make 5 copies of itself and add them all together; however, 5 will have a much more difficult time of making 'pig' copies of itself and adding them together.

And, of course, we still have puts and gets 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 whoever he is yelling at. In Ruby, if I say puts 'to be or not to be', what I am really saying is self.puts 'to be or not to be'. So what is self? It's a special variable which points to whatever object you are in. We don't even know how to be in an object yet, but until we find out, we are always going to be in a big object which is... the whole program! And lucky for us, the program has a few methods of its own, like puts and gets. Watch this:

```
iCantBelieveIMadeAVariableNameThisLongJustToPointToA3 = 3
puts iCantBelieveIMadeAVariableNameThisLongJustToPointToA3
self.puts iCantBelieveIMadeAVariableNameThisLongJustToPointToA3
```

```
3
3
```

If you didn't entirely follow all of that, that's OK. The important thing to take away 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.

Fancy String Methods

Let's learn a few fun string methods. You don't have to memorize them all; you can just look up this page again if you forget them. I just 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 there are great references on the internet with all of the string methods listed and explained. (I will show you where to find them at the end of this tutorial.) Really, 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... and isn't that really the whole point of the dictionary? So you don't have to know what's in it?

So, our first string method is reverse, which gives a backwards version of a string:

```
var1 = 'stop'
var2 = 'stressed'
var3 = 'Can you pronounce this sentence backwards?'

puts var1.reverse
puts var2.reverse
puts var3.reverse
puts var1
puts var2
puts var3
```

```
pots
desserts
?sdrawkcab ecnetnes siht ecnuonorp uoy naC
stop
stressed
Can you pronounce this sentence backwards?
```

As you can see, reverse doesn't reverse the original string; it just makes a new backwards version of it. That's why var1 is still 'stop' even after we called reverse on var1.

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

```
puts 'What is your full name?'
name = gets.chomp
puts 'Did you know there are ' + name.length +
     ' characters in your name, ' + name + '?'
```

```
What is your full name?
Christopher David Pine
#<TypeError: no implicit conversion of Fixnum into String>
```

Uh-oh! Something went wrong, and it looks like it happened sometime after the line name = gets.chomp... Do you see the problem? See if you can figure it out.

The problem is with length: it gives us a number, but we want a string. Easy enough, we'll just throw in a to_s (and cross our fingers):

```
puts 'What is your full name?'
name = gets.chomp
puts 'Did you know there are ' + name.length.to_s +
     ' characters in your name, ' + name + '?'
```

```
What is your full name?
Christopher David Pine
Did you know there are 22 characters in your name, Christopher David Pine?
```

No, I did not know that. Note: that's the number of characters in my name, not the number of letters (count 'em). I guess we could write a program which asks for your first, middle, 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? Good! It's nice to program, isn't it? After a few more chapters, though, you'll be amazed at what you can do.

So, there are also a number of string methods which change the case (uppercase and lowercase) of your 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 just like downcase, except that it switches the first character to uppercase (if it is a letter).

```
letters = 'aAbBcCdDeE'
puts letters.upcase
puts letters.downcase
puts letters.swapcase
puts letters.capitalize
puts ' a'.capitalize
puts letters
```

```
AABBCCDDEE
aabbccddee
AaBbCcDdEe
Aabbccddee
 a
aAbBcCdDeE
```

Pretty standard stuff. As you can see from the line puts ' a'.capitalize, the method capitalize only capitalizes the first character, not the first letter. Also, as we have 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. There are some methods which do change the associated object, but we haven't seen any yet, and we won't for some time.

The last of the fancy string methods we'll look at are for visual formatting. The first one, center, adds spaces to the beginning and end of the string to make it centered. However, just like you have to tell puts what you want it to print, and + what you want it to add, you have to tell center how wide you want your centered string to be. So if I wanted to center the lines of a poem, I would do it like this:

```
lineWidth = 50
puts(                'Old Mother Hubbard'.center(lineWidth))
puts(               'Sat in her cupboard'.center(lineWidth))
puts(         'Eating her curds an whey,'.center(lineWidth))
puts(          'When along came a spider'.center(lineWidth))
puts(         'Which sat down beside her'.center(lineWidth))
puts('And scared her poor shoe dog away.'.center(lineWidth))
```

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

Hmmm... I don't think that's how that nursery rhyme goes, but I'm too lazy to look it up. (Also, I wanted to line up the .center lineWidth part, so I put in those extra spaces before the strings. This is just because I think it is prettier that way. Programmers often have strong feelings about what is pretty in a program, and they often disagree about it. The more you program, the more you will come into your own style.) Speaking of being lazy, laziness isn't always a bad thing in programming. For example, see how I stored the width of the poem in the variable lineWidth? This was so that if I want to go back later and make the poem wider, I only have to change the very top line of the program, instead of every line which does centering. With a very long poem, this could save me a lot of time. That kind of laziness is really a virtue in programming.

So, about that centering... you may have noticed that it isn't quite as beautiful as what a word processor would have done. If you really want perfect centering (and maybe a nicer font), then you should just use a word processor! Ruby is a wonderful tool, but no tool is the right tool for every job.

The other two string formatting methods 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 take a look at all three in action:

```
lineWidth = 40
str = '--> text <--'
puts str.ljust  lineWidth
puts str.center lineWidth
puts str.rjust  lineWidth
puts str.ljust(lineWidth/2) + str.rjust(lineWidth/2)
```

```
--> text <--
              --> text <--
                            --> text <--
--> text <--                --> text <--
```

A Few Things to Try

Write an Angry Boss program. It should rudely ask 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., it should yell back WHADDAYA MEAN "I WANT A RAISE."?!? YOU'RE FIRED!! So here's something for you to do in order to play around more with center, ljust, and rjust: Write a program which will display a Table of Contents so that it looks like this:

```
                Table of Contents

Chapter 1:  Numbers                       page 1
Chapter 2:  Letters                      page 72
Chapter 3:  Variables                   page 118
```

Higher Math

(This section is totally optional. It assumes a fair degree of mathematical knowledge. If you aren't interested, you can go straight to Flow Control without any problems. However, a quick look at the section on Random Numbers might come in handy.)

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). Here, we'll look at the rest of the arithmetic methods, a random number generator, and the Math object, with its trigonometric and transcendental methods.

More Arithmetic

The other two arithmetic methods are ** (exponentiation) and % (modulus). So if you want to say "five squared" in Ruby, you would 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 division by a number. So, for example, if I divide 7 by 3, I get 2 with a remainder of 1. Let's see it working in a program:

```
puts 5 ** 2
puts 5 * 0.5
puts 7 / 3
puts 7 % 3
puts 365 % 7
```

```
25
2.23606797749979
2
1
1
```

From that last line, we learn that a (non-leap) year has some number of weeks, plus one day. So if your birthday was on a Tuesday this year, it will be on a Wednesday next year. You can also use floats with the modulus method. Basically, it works the only sensible way it could... but I'll let you play around with that.

There's one last method to mention before we check out the random number generator: abs. It just takes the absolute value of the number:

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

```
3
3
```

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 just like that, you'll get a float greater than or equal to 0.0 and less than 1.0. If you give rand an integer (5 for example), it will 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.

```
puts rand
puts rand
puts rand
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(1))
puts(rand(1))
puts(rand(1))
puts(rand(99999999999999999999999999999999999999999999999999999999999))
puts('The weatherman said there is a ' + rand(101).to_s + '% chance of rain,')
puts('but you can never trust a weatherman.')
```

```
0.17429261270690644
0.6038796470847551
0.5357456897902644
82
63
21
0
0
0
93717128025224525218060865578711367433735357080012377789690
The weatherman said there is a 9% chance of rain,
but you can never trust a weatherman.
```

Note that I used rand(101) to get back numbers from 0 to 100, and that rand(1) always gives back 0. Not understanding the range of possible return values is the biggest mistake I see people make with rand; even professional programmers; even in finished products you can buy at the store. I even had a CD player once which, if set on "Random Play," would play every song but the last one... (I wonder what would have happened if I had 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, once I was using randomly generated numbers to create a randomly generated world for a computer game. If I found a world that I really liked, perhaps I would want to play on it again, or send it to a friend.) In order to do this, you need to set the seed, which you can do with srand. Like this:

```
srand 1776
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts ''
srand 1776
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
```

```
24
35
36
58
```

```
70


24
35
36
58
70
```

It 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 never use srand), then just call srand 0. This seeds it with a really weird number, using (among other things) the current time on your computer, down to the millisecond.

The Math Object

Finally, let's look at the Math object. We might as well jump right in:

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

```
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 really beyond the, uh... scope of this tutorial. No pun intended. I swear. Suffice it to say, you can use Math::PI just like you would expect to.

As you can see, Math has all of the things you would expect a decent scientific calculator to have. And as always, the floats are really close to being the right answers.

So now let's flow!

Flow Control

Chapter 6

Ahhhh, flow control. This is where it all comes together. Even though this chapter is shorter and easier than the methods chapter, it will open up a whole world of programming possibilities. After this chapter, we'll be able to write truly interactive programs; in the past we have made programs which say different things depending on your keyboard input, but after this chapter they will actually do different things, too. But before we can do that, we need to be able to compare the objects in our programs. We need...

Comparison Methods

Let's rush through this part so we can get to the next section, Branching, where all the cool stuff happens. So, to see if one object is greater than or less than another, we use the methods > and <, like this:

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

```
false
true
```

No problem. Likewise, we can find out if an object is greater-than-or-equal-to another (or less-than-or-equal-to) with the methods >= and <=

```
puts 5 >= 5
puts 5 <= 4
```

```
true
false
```

And finally, we can see if two objects are equal or not using == (which means "are these equal?") and != (which means "are these different?"). It's important not to confuse = with ==. = is for telling a variable to point at an object (assignment), and == is for asking the question: "Are these two objects equal?"

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

```
true
true
```

Of course, we can compare strings, too. When strings get compared, they compare their lexicographical ordering, which basically means their dictionary ordering. cat comes before dog in the dictionary, so:

```
puts 'cat' < 'dog'
```

```
true
```

There's a catch, though: the way computers usually do things, they order capital letters as coming before lowercase letters. (That's how they store the letters in fonts, for example: all the capital letters first, then the lowercase ones.) This means that it will think 'Zoo' comes before 'ant', 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.

One last note before Branching: The comparison methods aren't giving us the strings 'true' and 'false'; they are giving us the special objects true and false. (Of course, true.to_s gives us 'true', which is why puts printed 'true'.) true and false are used all the time in...

Branching

Branching is a simple concept, but powerful. In fact, it's so simple that I bet I don't even have to explain it at all; I'll just show you:

```
puts 'Hello, what\'s your name?'
name = gets.chomp
puts 'Hello, ' + name + '.'
if name == 'Chris'
  puts 'What a lovely name!'
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 is branching. If what comes after the if is true, we run the code between the if and the end. If what comes after the if is false, we don't. Plain and simple.

I indented the code between the if and the end just because I think it's easier to keep track of the branching that way. Almost all programmers do this, regardless of what language they are programming in. It may not seem much help in this simple example, but when things get more complex, it makes a big difference.

Often, we would like a program to do one thing if an expression is true, and another if it is false. That's what else is for:

```
puts 'I am a fortune-teller.  Tell me your name:'
name = gets.chomp
if name == 'Chris'
  puts 'I see great things in your future.'
else
  puts 'Your future is... Oh my!  Look at the time!'
  puts 'I really have to go, sorry!'
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:
Ringo
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 we take the path for people whose name == 'Chris', or else do we take the other path?

And just like the branches of a tree, you can have branches which themselves have branches:

```
puts 'Hello, and welcome to 7th grade English.'
puts 'My name is Mrs. Gabbard.  And your name is...?'
name = gets.chomp

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?'
  reply = gets.chomp

  if reply.downcase == 'yes'
    puts 'Hmmph!  Well, sit down!'
  else
    puts 'GET OUT!!'
  end
end
```

```
Hello, and welcome to 7th 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 it...

```
Hello, and welcome to 7th grade English.
My name is Mrs. Gabbard.  And your name is...?
Chris
Please take a seat, Chris.
```

Sometimes it might get confusing trying to figure out where all of the ifs, elses, and ends go. What I do is write the end at the same time I write the if. So as I was writing the above program, this is how it looked first:

```
puts 'Hello, and welcome to 7th grade English.'
puts 'My name is Mrs. Gabbard.  And your name is...?'
name = gets.chomp

if name == name.capitalize
else
end
```

Then I filled it in with comments, stuff in the code the computer will ignore:

```
puts 'Hello, and welcome to 7th grade English.'
puts 'My name is Mrs. Gabbard.  And your name is...?'
name = gets.chomp

if name == name.capitalize
  # She's civil.
else
  # She gets mad.
end
```

Anything after a # is considered a comment (unless, of course, you are in a string). After that, I replaced the comments with working code. Some people like to leave the comments in; personally, I think well-written code usually speaks for itself. I used to use more comments, but the more "fluent" in Ruby I become, the less I use them. I actually find them distracting much of the time. It's a personal choice; you'll find your own (usually evolving) style. So my next step looked like this:

```
puts 'Hello, and welcome to 7th grade English.'
puts 'My name is Mrs. Gabbard.  And your name is...?'
name = gets.chomp

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??'
  reply = gets.chomp

  if reply.downcase == 'yes'
  else
  end
end
```

Again, I wrote down the if, else, and end all at the same time. It really helps me keep track of "where I am" in the code. It also makes the job seem easier because I can focus on one small part, like filling in the code between the if and the else. The other benefit of doing it this way is that the computer can understand the program at any stage. Every one of the unfinished versions of the program I showed you would run. They weren't finished, but they were working programs. That way I could test it as I wrote it, which helped to see how it was coming along and where it still needed work. When it passed all of the tests, that's how I knew I was done!

These tips will help you write programs with branching, but they also help with the other main type of flow control:

Looping

Often, you'll want your computer to do the same thing over and over again—after all, that's what computers are supposed to be so good at.

When you tell your computer to keep repeating something, you also need to tell it when to stop. Computers never get bored, so if you don't tell it to stop, it won't. We make sure this doesn't happen by telling the computer to repeat certain parts of a program while a certain condition is true. This works very similarly to how if works:

```
command = ''

while command != 'bye'
  puts command
  command = gets.chomp
end

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!
```

And that's a loop. (You may have noticed the blank line at the beginning of the output; it's from the first puts, before the first gets. How would you change the program to get rid of this first line. Test it! Did it work exactly like the program above, other than that first blank line?)

Loops allow you to do all kinds of interesting things, as I'm sure you can imagine. However, they can also cause problems if you make a mistake. What if your computer gets trapped in an infinite loop? If you think this may have happened, just hold down the Ctrl key and press C.

Before we start playing around with loops, though, let's learn a few things to make our job easier.

A Little Bit of Logic

Let's take a look at our first branching program again. What if my wife came home, saw the program, tried it out, and it didn't tell her what a lovely name she had? Well... she probably wouldn't care. But I'd care! So let's rewrite it:

```
puts 'Hello, what\'s your name?'
name = gets.chomp
puts 'Hello, ' + name + '.'
if name == 'Chris'
  puts 'What a lovely name!'
else
  if name == 'Katy'
    puts 'What a lovely name!'
  end
end
```

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

It works... but it isn't a very pretty program. Why not? Well, the best rule I ever learned in programming was the DRY rule: Don't Repeat Yourself. I could probably write a small book just on why that is such a good rule. In our case, we repeated the line puts 'What a lovely name!'. Why is this such a big deal? Well, what if I made a spelling mistake when I rewrote it? What if I wanted to change it from 'lovely' to 'beautiful' on both lines? I'm lazy, remember? Basically, if I want the program to do the same thing when it gets 'Chris' or 'Katy', then it should really do the same thing:

```
puts 'Hello, what\'s your name?'
name = gets.chomp
puts 'Hello, ' + name + '.'
if (name == 'Chris' or name == 'Katy')
  puts 'What a lovely name!'
end
```

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

Much better. In order to make it work, I used or. The other logical operators are and and not. It is always a good idea to use parentheses when working with these. Let's see how they work:

```
iAmChris  = true
iAmPurple = false
iLikeFood = true
iEatRocks = false

puts (iAmChris  and iLikeFood)
puts (iLikeFood and iEatRocks)
puts (iAmPurple and iLikeFood)
puts (iAmPurple and iEatRocks)
puts
puts (iAmChris  or iLikeFood)
puts (iLikeFood or iEatRocks)
puts (iAmPurple or iLikeFood)
puts (iAmPurple or iEatRocks)
puts
puts (not iAmPurple)
puts (not iAmChris )
```

```
true
false
false
false

true
true
true
false
```

```
true
false
```

The only one of these which might trick you is or. In English, we often use "or" to mean "one or the other, but not both." For example, your mom might say, "For dessert, you can have pie or cake." She did not mean you could have them both! A computer, on the other hand, uses or to mean "one or the other, or both." (Another way of saying it is, "at least one of these is true.") This is why computers are more fun than moms.

A Few Things to Try

```
"99 bottles of beer on the wall..."
```

Write a program which prints out the lyrics to that beloved classic, that field-trip favorite: "99 Bottles of Beer on the Wall."

Write a Deaf Grandma program. Whatever you say to grandma (whatever you type in), she should respond with 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 at the end of the methods chapter.) You can't stop talking to grandma until you shout BYE.

Hint: Don't forget about chomp! 'BYE'with an Enter is not 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.

Extend your Deaf Grandma program: 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 which will ask for a starting year and an ending year, and then puts all of the leap years between them (and including them, if they are also leap years). Leap years are years divisible by four (like 1984 and 2004). However, years divisible by 100 are not leap years (such as 1800 and 1900) unless they are divisible by 400 (like 1600 and 2000, which were in fact leap years).

(Yes, it's all pretty confusing, but not as confusing as having July in the middle of the winter, which is what would eventually happen.)

When you finish those, take a break! You've learned a lot already. Congratulations! Are you surprised at the number of things you can tell a computer to do? A few more chapters and you'll be able to program just about anything. Seriously! Just look at all the things you can do now that you couldn't do without looping and branching.

Now let's learn about a new kind of object, one which keeps track of lists of other objects: arrays.

Arrays and Iterators

Chapter 7

Let's write a program which asks us to type in as many words as we want (one word per line, continuing until we just press Enter on an empty line), and which then repeats the words back to us in alphabetical order. OK?

So... first we'll—uh... um... hmmm... Well, we could—er... um...

You know, I don't think we can do it. We need a way to store an unknown amount of words, and how to keep track of them all together, so they don't get mixed up with other variables. We need to put them in some sort of a list. We need arrays.

An array is just a list in 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 take a look at some arrays:

```
[]
[5]
['Hello', 'Goodbye']

flavor = 'vanilla'             # This is not an array, of course...
[89.9, flavor, [true, false]]  # ...but this is.
```

So first we have an empty array, then an array holding a single number, then an array holding two strings. Next, we have a simple assignment; then an array holding three objects, the last of which is the array [true, false]. Remember, variables aren't objects, so our last array is really pointing to float, a string, and an array. Even if we were to set flavor to point to something else, that wouldn't change the array.

To help us find a particular object in an array, each slot is given an index number. Programmers (and, incidentally, most mathematicians) start counting from zero, though, so the first slot in the array is slot zero. Here's how we would reference the objects in an array:

```
names = ['Ada', 'Belle', 'Chris']

puts names
puts names[0]
puts names[1]
puts names[2]
puts names[3]  # This is out of range.
```

```
Ada
Belle
Chris
Ada
```

```
Belle
Chris
```

So, we see that puts names prints each name in the array names. Then we use puts names[0] to print out the "first" name in the array, and puts names[1] to print the "second"... I'm sure this seems confusing, but you do get used to it. You just have to really start thinking that counting begins at zero, and stop using words like "first" and "second". If you go out to 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 are juggling clubs 0-5. Hopefully in the next few months, we'll be able to juggle club 6 (and thus be juggling seven clubs between us). You'll know you've got it when you start using the word "zeroth". 😃 Yes, it's a real word; ask any programmer or mathematician.

Finally, we tried puts names[3], just to see what would happen. Were you expecting an error? Sometimes when you ask a question, your question doesn't make sense (at least to your computer); 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 names[3]? nil: Ruby's way of saying "nothing". nil is a special object which basically means "not any other object." And when you puts nil, it prints out nothing. (Just a new line.)

If all this funny numbering of array slots is getting to you, fear not! Often, we can avoid them completely by using various array methods, like this one:

The Method each

each allows us to do something (whatever we want) to each object the array points to. So, if we want to say something nice about each language in the array below, we'd do this:

```ruby
languages = ['English', 'German', 'Ruby']

languages.each do |lang|
  puts 'I love ' + lang + '!'
  puts 'Don\'t you?'
end

puts 'And let\'s hear it for C++!'
puts '...'
```

```
I love English!
Don't you?
I love German!
Don't you?
I love Ruby!
Don't you?
And let's hear it for C++!
```

So what just happened? Well, we were able to go through every object in the array without using any numbers, so that's definitely nice. Translating into English, the above program reads something like: For each object in languages, point the variable lang to the object and then do everything I tell you to, until you come to the end. (Just so you know, C++ is another programming language. It's much harder to learn than Ruby; usually, a C++ program will be many times longer than a Ruby program which does the same thing.)

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

But not each; each is just another array method. Methods like each which "act like" loops are often called iterators.

One thing to notice about iterators is that they are always followed by do...end. while and if never had a do near them; we only use do with iterators.

Here's another cute little iterator, but it's not an array method... it's an integer method!

```
3.times do
  puts 'Hip-Hip-Hooray!'
end
```

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

More Array Methods

So we've learned each, but there are many other array methods... almost as many as there are string methods! In fact, some of them (like length, reverse, +, and *) work just like they do for strings, except that they operate on the slots of the array rather than the letters of the string. Others, like last and join, are specific to arrays. Still others, like push and pop, actually change the array. And just 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 (right here).

First, let's look at to_s and join. join works much like to_s does, except that it adds a string in between the array's objects. Let's take a look:

```
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
```

```
artichoke
brioche
caramel

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

artichoke, brioche, caramel

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

As you can see, puts treats arrays differently from other objects: it just calls puts on each of the objects in the array. That's why putsing an empty array 200 times doesn't do anything; the array doesn't point to anything, so there's nothing to puts. (Doing nothing 200 times is still doing nothing.) Try putsing an array containing other arrays; does it do what you expected?

Also, did you notice that I left out the empty strings when I wanted to puts a blank line? It does the same thing.

Now let's take a look at push, pop, and last. The methods push and pop are sort of opposites, like + and - are. push adds an object to the end of your array, and pop removes the last object from the array (and tell 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 leaves the array alone. 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
```

A Few Things to Try

Write the program we talked about at the very beginning of this chapter. Hint: There's a lovely array method which will give you a sorted version of an array: sort. Use it!

Try writing the above program without using the sort method. A large part of programming is solving problems, so get all the practice you can!

Rewrite your Table of Contents program (from the chapter on methods). Start the program with an array holding all of the information for your Table of Contents (chapter names, page numbers, etc.). Then print out the information from the array in a beautifully formatted Table of Contents.

So far we have learned quite a number of different methods. Now it's time to learn how to make our own.

Writing Your Own Methods

Chapter 8

As we've seen, loops and iterators allow us to do the same thing (run the same code) over and over again. However, sometimes we want to do the same thing a number of times, but from different places in the program. For example, let's say we were writing a questionnaire program for a psychology student. From the psychology students I have known and the questionnaires they have given me, it would probably go something like this:

```ruby
puts 'Hello, and thank you for taking the time to'
puts 'help me with this experiment.  My experiment'
puts 'has to do with the way people feel about'
puts 'Mexican food.  Just think about Mexican food'
puts 'and try to answer every question honestly,'
puts 'with either a "yes" or a "no".  My experiment'
puts 'has nothing to do with bed-wetting.'
puts

### We ask these questions, but we ignore their answers.

goodAnswer = false
while (not goodAnswer)
  puts 'Do you like eating tacos?'
  answer = gets.chomp.downcase
  if (answer == 'yes' or answer == 'no')
    goodAnswer = true
  else
    puts 'Please answer "yes" or "no".'
  end
end

goodAnswer = false
while (not goodAnswer)
  puts 'Do you like eating burritos?'
  answer = gets.chomp.downcase
  if (answer == 'yes' or answer == 'no')
    goodAnswer = true
  else
    puts 'Please answer "yes" or "no".'
  end
end

### We pay attention to *this* answer, though.
```

```ruby
goodAnswer = false
while (not goodAnswer)
  puts 'Do you wet the bed?'
  answer = gets.chomp.downcase
  if (answer == 'yes' or answer == 'no')
    goodAnswer = true
    if answer == 'yes'
      wetsBed = true
    else
      wetsBed = false
    end
  else
    puts 'Please answer "yes" or "no".'
  end
end

goodAnswer = false
while (not goodAnswer)
  puts 'Do you like eating chimichangas?'
  answer = gets.chomp.downcase
  if (answer == 'yes' or answer == 'no')
    goodAnswer = true
  else
    puts 'Please answer "yes" or "no".'
  end
end

puts 'Just a few more questions...'

goodAnswer = false
while (not goodAnswer)
  puts 'Do you like eating sopapillas?'
  answer = gets.chomp.downcase
  if (answer == 'yes' or answer == 'no')
    goodAnswer = true
  else
    puts 'Please answer "yes" or "no".'
  end
end

### Ask lots of other questions about Mexican food.

puts
puts 'DEBRIEFING:'
puts 'Thank you for taking the time to help with'
puts 'this experiment.  In fact, this experiment'
puts 'has nothing to do with Mexican food.  It is'
puts 'an experiment about bed-wetting.  The Mexican'
puts 'food was just there to catch you off guard'
puts 'in the hopes that you would answer more'
puts 'honestly.  Thanks again.'
puts
puts wetsBed
```

```
Hello, and thank you for taking the time to
help me with this experiment.  My experiment
has to do with the way people feel about
Mexican food.  Just think about Mexican food
and try to answer every question honestly,
with either a "yes" or a "no".  My experiment
has nothing to do with bed-wetting.

Do you like eating tacos?
yes
Do you like eating burritos?
yes
Do you wet the bed?
no way!
Please answer "yes" or "no".
Do you wet the bed?
NO
Do you like eating chimichangas?
yes
Just a few more questions...
Do you like eating sopapillas?
yes

DEBRIEFING:
Thank you for taking the time to help with
this experiment.  In fact, this experiment
has nothing to do with Mexican food.  It is
an experiment about bed-wetting.  The Mexican
food was just there to catch you off guard
in the hopes that you would answer more
honestly.  Thanks again.

false
```

That was a pretty long program, with lots of repetition. (All of the sections of code around the questions about

Mexican food were identical, and the bed-wetting question was only slightly different.) Repetition is a bad thing. Still, we can't make it into a big loop or iterator, because sometimes we have things we want to do between questions. In situations like these, it's best to write a method. Here's how:

```
def sayMoo
  puts 'mooooooo...'
end
```

Uh... our program didn't sayMoo. Why not? Because we didn't tell it to. We told it how to sayMoo, but we never actually said to do it. Let's give it another shot:

```
def sayMoo
  puts 'mooooooo...'
end

sayMoo
sayMoo
puts 'coin-coin'
sayMoo
sayMoo
```

```
mooooooo...
mooooooo...
coin-coin
mooooooo...
mooooooo...
```

Ahhh, much better. (Just in case you don't speak French, that was a French duck in the middle of the program. In France, ducks say "coin-coin".)

So we defined the method sayMoo. (Method names, like variable names, start with a lowercase letter. There are a few exceptions, though, like + 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 just associated with the object representing the whole program. In the next chapter we'll see how to add methods to other objects. But first...

Method Parameters

You may have noticed that some methods (like gets, to_s, reverse...) you can just call on an object. However, other methods (like +, -, puts...) take parameters 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 add a parameter to sayMoo (let's say, the number of moos), we would do this:

```
def sayMoo numberOfMoos
  puts 'mooooooo...'*numberOfMoos
end
```

```
sayMoo 3
puts 'oink-oink'
sayMoo  # This should give an error because the parameter is missing.
```

```
mooooooo...mooooooo...mooooooo...
oink-oink
#<ArgumentError: wrong number of arguments (given 0, expected 1)>
```

numberOfMoos is a variable which points to the parameter passed in. I'll say that again, but it's a little confusing: numberOfMoos is a variable which points to the parameter passed in. So if I type in sayMoo 3, then the parameter is 3, and the variable numberOfMoos points to 3.

As you can see, the parameter is now required. After all, what is sayMoo supposed to multiply 'mooooooo...' by if you don't give it a parameter? Your poor computer has no idea.

If objects in Ruby are like nouns in English, and methods are like verbs, then you can think of parameters as adverbs (like with sayMoo, where the parameter told us how to sayMoo) or sometimes as direct objects (like with puts, where the parameter is what gets putsed).

Local Variables

In the following program, there are two variables:

```
def doubleThis num
  numTimes2 = num*2
  puts num.to_s+' doubled is '+numTimes2.to_s
end

doubleThis 44
```

```
44 doubled is 88
```

The variables are num and numTimes2. They both sit inside the method doubleThis. These (and all of the variables you have seen so far) are local variables. This means that they live inside the method, and they cannot leave. If you try, you will get an error:

```
def doubleThis num
  numTimes2 = num*2
  puts num.to_s+' doubled is '+numTimes2.to_s
end

doubleThis 44
puts numTimes2.to_s
```

```
44 doubled is 88
#<NameError: undefined local variable or method `numTimes2' for #
<StringIO:0x00000002e78130>>
```

Undefined local variable... In fact, we did define that local variable, but it isn't local to where we tried to use it; it's local to the method.

This might seem inconvenient, but it actually quite nice. While it does mean that you have no access to variables inside methods, it also means that they have no access to your variables, and thus can't screw them up:

```
def littlePest var
  var = nil
  puts 'HAHA!  I ruined your variable!'
end

var = 'You can\'t even touch my variable!'
littlePest var
puts var
```

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

There are actually two variables in that little program named var: one inside littlePest, and one outside of it. When we called littlePest var, we really just passed the string from one var to the other, so that both were pointing to the same string. Then littlePest pointed its own local var to nil, but that did nothing to the var outside the method.

Return Values

You may have noticed that some methods give you something back when you call them. For example, gets returns a string (the string you typed in), and the + method in 5+3, (which is really 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 methods returning a value to where the method was called, and your program outputting information to your screen, like puts does. Notice that 5+3 returns 8; it does not output 8.

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

```
returnVal = puts 'This puts returned:'
puts returnVal
```

```
This puts returned:
```

The first puts didn't seem to return anything, and in a way it didn't; it returned nil. Though we didn't test it, the second puts did, too; puts always returns nil. Every method has to return something, even if it's just nil.

Take a quick break and write a program to find out what sayMoo returned.

Were you surprised? Well, here's how it works: the value returned from a method is simply the last line of the method. In the case of sayMoo, this means it returns puts 'mooooooo...'*numberOfMoos, which is just nil since puts always returns nil. If we wanted all of our methods to return the string 'yellow submarine', we would just need to put that at the end of them:

```
def sayMoo numberOfMoos
  puts 'mooooooo...'*numberOfMoos
  'yellow submarine'
end

x = sayMoo 2
puts x
```

```
mooooooo...mooooooo...
yellow submarine
```

So, let's try that psychology experiment again, but this time we'll write a method to ask the questions for us. It will need to take the question as a parameter, and return true if they answered yes and false if they answered no. (Even though most of the time we just ignore the answer, it's still a good idea for our method to return the answer. This way we can use it for the bed-wetting question, too.) I'm also going to shorten the greeting and the debriefing, just so this is easier to read:

```
def ask question
  goodAnswer = false
  while (not goodAnswer)
    puts question
    reply = gets.chomp.downcase

    if (reply == 'yes' or reply == 'no')
      goodAnswer = true
      if reply == 'yes'
        answer = true
      else
        answer = false
      end
    else
      puts 'Please answer "yes" or "no".'
    end
  end

  answer  # This is what we return (true or false).
```

```
end

puts 'Hello, and thank you for...'
puts

ask 'Do you like eating tacos?'      # We ignore this return value.
ask 'Do you like eating burritos?'
wetsBed = ask 'Do you wet the bed?'  # We save this return value.
ask 'Do you like eating chimichangas?'
ask 'Do you like eating sopapillas?'
ask 'Do you like eating tamales?'
puts 'Just a few more questions...'
ask 'Do you like drinking horchata?'
ask 'Do you like eating flautas?'

puts
puts 'DEBRIEFING:'
puts 'Thank you for...'
puts
puts wetsBed
```

```
Hello, and thank you for...

Do you like eating tacos?
yes
Do you like eating burritos?
yes
Do you wet the bed?
no way!
Please answer "yes" or "no".
Do you wet the bed?
NO
Do you like eating chimichangas?
yes
Do you like eating sopapillas?
yes
Do you like eating tamales?
yes
Just a few more questions...
Do you like drinking horchata?
yes
Do you like eating flautas?
yes
```

```
DEBRIEFING:
Thank you for...

false
```

Not bad, huh? We were able to add more questions (and adding questions is easy now), but our program is still quite a bit shorter! It's a big improvement — a lazy programmer's dream.

One More Big Example

I think another example method would be helpful here. We'll call this one englishNumber. It will take a number, like 22, and return the english version of it (in this case, the string 'twenty-two'). For now, let's have it only work on integers from 0 to 100.

(NOTE: This method uses a new trick to return from a method early using the return keyword, and introduces a new twist on branching: elsif. It should be clear in context how these work.)

```ruby
def englishNumber number
  # We only want numbers from 0-100.
  if number < 0
    return 'Please enter a number zero or greater.'
  end
  if number > 100
    return 'Please enter a number 100 or lesser.'
  end

  numString = ''  # This is the string we will return.

  # "left" is how much of the number we still have left to write out.
  # "write" is the part we are writing out right now.
  # write and left... get it?  :)
  left  = number
  write = left/100          # How many hundreds left to write out?
  left  = left - write*100  # Subtract off those hundreds.

  if write > 0
    return 'one hundred'
  end

  write = left/10          # How many tens left to write out?
  left  = left - write*10  # Subtract off those tens.

  if write > 0
    if write == 1  # Uh-oh...
      # Since we can't write "tenty-two" instead of "twelve",
      # we have to make a special exception for these.
    if    left == 0
      numString = numString + 'ten'
    elsif left == 1
      numString = numString + 'eleven'
    elsif left == 2
      numString = numString + 'twelve'
    elsif left == 3
      numString = numString + 'thirteen'
    elsif left == 4
      numString = numString + 'fourteen'
    elsif left == 5
      numString = numString + 'fifteen'
    elsif left == 6
      numString = numString + 'sixteen'
    elsif left == 7
      numString = numString + 'seventeen'
    elsif left == 8
      numString = numString + 'eighteen'
    elsif left == 9
      numString = numString + 'nineteen'
    end
    # Since we took care of the digit in the ones place already,
    # we have nothing left to write.
    left = 0
  elsif write == 2
    numString = numString + 'twenty'
  elsif write == 3
    numString = numString + 'thirty'
  elsif write == 4
    numString = numString + 'forty'
  elsif write == 5
    numString = numString + 'fifty'
  elsif write == 6
    numString = numString + 'sixty'
  elsif write == 7
    numString = numString + 'seventy'
  elsif write == 8
    numString = numString + 'eighty'
  elsif write == 9
    numString = numString + 'ninety'
  end

  if left > 0
    numString = numString + '-'
  end
  end

  write = left  # How many ones left to write out?
  left  = 0     # Subtract off those ones.

  if write > 0
    if    write == 1
      numString = numString + 'one'
    elsif write == 2
      numString = numString + 'two'
    elsif write == 3
      numString = numString + 'three'
    elsif write == 4
      numString = numString + 'four'
    elsif write == 5
      numString = numString + 'five'
    elsif write == 6
      numString = numString + 'six'
    elsif write == 7
      numString = numString + 'seven'
    elsif write == 8
      numString = numString + 'eight'
    elsif write == 9
      numString = numString + 'nine'
    end
  end

  if numString == ''
    # The only way "numString" could be empty is if
    # "number" is 0.
    return 'zero'
  end

  # If we got this far, then we had a number somewhere
  # in between 0 and 100, so we need to return "numString".
  numString
end

puts englishNumber(  0)
puts englishNumber(  9)
puts englishNumber( 10)
puts englishNumber( 11)
puts englishNumber( 17)
puts englishNumber( 32)
puts englishNumber( 88)
puts englishNumber( 99)
puts englishNumber(100)
```

```
zero
nine
ten
eleven
seventeen
thirty-two
eighty-eight
ninety-nine
one hundred
```

Well, there are certainly a few things about this program I don't like. First, it has too much repetition. Second, it doesn't handle numbers greater than 100. Third, there are too many special cases, too many returns. Let's use some arrays and try to clean it up a bit:

```
def englishNumber number
  if number < 0  # No negative numbers.
    return 'Please enter a number that isn\'t negative.'
  end
  if number == 0
    return 'zero'
  end

  # No more special cases! No more returns!

  numString = ''  # This is the string we will return.

  onesPlace = ['one',     'two',       'three',    'four',     'five',
               'six',     'seven',     'eight',    'nine']
  tensPlace = ['ten',     'twenty',    'thirty',   'forty',    'fifty',
               'sixty',   'seventy',   'eighty',   'ninety']
  teenagers = ['eleven',  'twelve',    'thirteen', 'fourteen', 'fifteen',
               'sixteen', 'seventeen', 'eighteen', 'nineteen']

  # "left" is how much of the number we still have left to write out.
  # "write" is the part we are writing out right now.
  # write and left... get it?  :)
  left  = number
  write = left/100          # How many hundreds left to write out?
  left  = left - write*100  # Subtract off those hundreds.

  if write > 0
```

```
    # Now here's a really sly trick:
    hundreds  = englishNumber write
    numString = numString + hundreds + ' hundred'
    # That's called "recursion". So what did I just do?
    # I told this method to call itself, but with "write" instead of
    # "number". Remember that "write" is (at the moment) the number of
    # hundreds we have to write out. After we add "hundreds" to
    # "numString", we add the string ' hundred' after it.
    # So, for example, if we originally called englishNumber with
    # 1999 (so "number" = 1999), then at this point "write" would
    # be 19, and "left" would be 99. The laziest thing to do at this
    # point is to have englishNumber write out the 'nineteen' for us,
    # then we write out ' hundred', and then the rest of
    # englishNumber writes out 'ninety-nine'.

    if left > 0
      # So we don't write 'two hundredfifty-one'...
      numString = numString + ' '
    end
  end

  write = left/10          # How many tens left to write out?
  left  = left - write*10  # Subtract off those tens.

  if write > 0
    if ((write == 1) and (left > 0))
      # Since we can't write "tenty-two" instead of "twelve",
      # we have to make a special exception for these.
      numString = numString + teenagers[left-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 left to write.
      left = 0
    else
      numString = numString + tensPlace[write-1]
      # The "-1" is because tensPlace[3] is 'forty', not 'thirty'.
    end

    if left > 0
      # So we don't write 'sixtyfour'...
      numString = numString + '-'
    end
  end
  write = left  # How many ones left to write out?
  left  = 0     # Subtract off those ones.

  if write > 0
    numString = numString + onesPlace[write-1]
    # The "-1" is because onesPlace[3] is 'four', not 'three'.
  end

  # Now we just return "numString"...
  numString
end

puts englishNumber(  0)
puts englishNumber(  9)
puts englishNumber( 10)
puts englishNumber( 11)
puts englishNumber( 17)
puts englishNumber( 32)
puts englishNumber( 88)
puts englishNumber( 99)
puts englishNumber(100)
puts englishNumber(101)
puts englishNumber(234)
puts englishNumber(3211)
puts englishNumber(999999)
puts englishNumber(1000000000000)
```

```
zero
nine
ten
eleven
seventeen
thirty-two
eighty-eight
ninety-nine
one hundred
one hundred one
two hundred thirty-four
thirty-two hundred eleven
ninety-nine hundred ninety-nine hundred ninety-nine
one hundred hundred hundred hundred hundred hundred
```

Ahhhh.... That's much, much better. The program is fairly dense, which is why I put in so many comments. It even works for large numbers... though not quite as nicely as one would hope. For example, I think 'one trillion' would be a nicer return value for that last number, or even 'one million million' (though all three are correct). In fact, you can do that right now...

A Few Things to Try

Expand upon englishNumber. First, put in thousands. So it should return 'one thousand' instead of 'ten hundred' and 'ten thousand' instead of 'one hundred hundred'.

Expand upon englishNumber some more. Now put in millions, so you get 'one million' instead of 'one thousand thousand'. Then try adding billions and trillions. How high can you go?

How about weddingNumber? It should work almost the same as englishNumber, except that it should insert the word "and" all over the place, returning things like 'nineteen hundred and seventy and two', or however wedding invitations are supposed to look. I'd give you more examples, but I don't fully understand it myself. You might need to contact a wedding coordinator to help you.

"Ninety-nine bottles of beer..." Using englishNumber and your old program, write out the lyrics to this song the right way this time. Punish your computer: have it start at 9999. (Don't pick a number too large, though, because writing all of that to the screen takes your computer quite a while. A hundred thousand bottles of beer takes some time; and if you pick a million, you'll be punishing yourself as well!

Congratulations! At this point, you are a true programmer! You have learned everything you need to build huge programs from scratch. If you have ideas for programs you would like to write for yourself, give them a shot!

Of course, building everything from scratch can be a pretty slow process. Why spend time writing code that someone else already wrote? Would you like your program to send some email? Would you like to save and load files on your computer? How about generating web pages for a tutorial where the code samples are all automatically tested? 😉 Ruby has many different kinds of objects we can use to help us write better programs faster.

Classes

Chapter 9

So far we've seen several different kinds, or classes, of objects: strings, integers, floats, arrays, and a few special objects (true, false, and nil) which we'll talk about later. In Ruby, these classes are always capitalized: String, Integer, Float, Array... etc. In general, if we want to create a new object of a certain class, we use new:

```
a = Array.new  + [12345]  # Array  addition.
b = String.new + 'hello'  # String addition.
c = Time.new

puts 'a = '+a.to_s
puts 'b = '+b.to_s
puts 'c = '+c.to_s
```

```
a = [12345]
b = hello
c = 2016-11-17 01:26:50 -0600
```

Because we can create arrays and strings using [...] and '...' respectively, we rarely create them using new. (Though it's not really obvious from the above example, String.new creates an empty string, and Array.new creates an empty array.) Also, numbers are special exceptions: you can't create an integer with Integer.new. You just have to write the integer.

The Time Class

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

```
time  = Time.new   # The moment I generated this web page.
time2 = time + 60  # One minute later.

puts time
puts time2
```

```
2016-11-17 01:26:50 -0600
2016-11-17 01:27:50 -0600
```

You can also make a time for a specific moment using Time.mktime:

```
puts Time.mktime(2000, 1, 1)          # Y2K.
puts Time.mktime(1976, 8, 3, 10, 11)  # When I was born.
```

```
2000-01-01 00:00:00 -0600
1976-08-03 10:11:00 -0500
```

Notice: that's when I was born in Pacific Daylight Savings Time (PDT). When Y2K struck, though, it was Pacific Standard Time (PST), at least to us West Coasters. The parentheses are to group the parameters to mktime together. The more parameters you add, the more accurate your time becomes.

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!

A Few Things to Try

One billion seconds... Find out the exact second you were born (if you can). Figure out when you will 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 in, then the month, then the day. Figure out how old they are and give them a big SPANK! for each birthday they have had. The Hash Class

Another useful class is the Hash class. Hashes are a lot like arrays: they have a bunch of slots which can point to various objects. However, 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 are just sort of jumbled together), and you can use any object to refer to a slot, not just 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 really fit into an ordered list. For example, the colors I use for different parts of the code which created this tutorial:

```
colorArray = []  # same as Array.new
colorHash  = {}  # same as Hash.new

colorArray[0]         = 'red'
colorArray[1]         = 'green'
colorArray[2]         = 'blue'
colorHash['strings']  = 'red'
colorHash['numbers']  = 'green'
colorHash['keywords'] = 'blue'

colorArray.each do |color|
  puts color
end
colorHash.each do |codeType, color|
  puts codeType + ':  ' + color
end
```

```
red
green
```

```
blue
strings:  red
numbers:  green
keywords:  blue
```

If I use an array, I have to remember that slot 0 is for strings, slot 1 is for numbers, etc. But if I use a hash, it's easy! Slot 'strings' holds the color of the strings, of course. Nothing to remember. You might have noticed that when we used each, the objects in the hash didn't come out in the same order we put them in. Arrays are for keeping things in order, not hashes.

Though people usually use strings to name the slots in a hash, you could use any kind of object, even arrays and other hashes (though I can't think of why you would want to do this...):

```
weirdHash = Hash.new

weirdHash[12] = 'monkeys'
weirdHash[[]] = 'emptiness'
weirdHash[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.

Extending Classes

At the end of the last chapter, you wrote a method to give the English phrase for a given integer. It wasn't an integer method, though; it was just a generic "program" method. Wouldn't it be nice if you could write something like 22.to_eng instead of englishNumber 22? Here's how you would do that:

```
class Integer
  def to_eng
    if self == 5
      english = 'five'
    else
      english = 'fifty-eight'
    end

    english
  end
end

# I'd better test on a couple of numbers...
puts 5.to_eng
puts 58.to_eng
```

```
five
```

```
fifty-eight
```

Well, I tested it; it seems to work. 😉

So we defined 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, if you didn't like the way a built-in method like to_s worked, you could just 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.

So... confused yet? Let me go over that last program some more. So far, whenever we executed any code or defined any methods, we did it in the default "program" object. In our last program, we left that object for the first time and went into the class Integer. We defined a method there (which makes it an integer method) and all integers can use it. Inside that method we use self to refer to the object (the integer) using the method.

Creating Classes

We've seen a number of different classes of objects. However, it's easy to come up with kinds of objects that Ruby doesn't have. Luckily, creating a new class is as easy as extending an old one. Let's say we wanted to make some dice in Ruby. Here's how we could make the Die class:

```ruby
class Die

  def roll
    1 + rand(6)
  end

end

# Let's make a couple of dice...
dice = [Die.new, Die.new]

# ...and roll them.
dice.each do |die|
  puts die.roll
end
```

```
6
2
```

(If you skipped the section on random numbers, rand(6) just gives a random number between 0 and 5.)

And that's it! Objects of our very own.

We can define all sorts of methods for our objects... but there's something missing. Working with these objects feels a lot like programming before we learned about variables. Look at our dice, for example. We can roll them, and each time we do they give us a different number. But if we wanted to hang on to that number, we would 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 the number. If we keep track of the die, we shouldn't also have to keep track of the number it is showing.

However, if we try to store the number we rolled in a (local) variable in roll, it will be gone as soon as roll is finished. We need to store the number in a different kind of variable:

Instance Variables

Normally when we want to talk about a string, we will just call it a string. However, we could also call it a string object. Sometimes programmers might call it an instance of the class String, but this is just a fancy (and rather long-winded) way of saying string. An instance of a class is just an object of that class.

So instance variables are just 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:

```
class Die

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

  def showing
    @numberShowing
  end

end

die = Die.new
die.roll
puts die.showing
puts die.showing
die.roll
puts die.showing
puts die.showing
```

```
4
4
6
6
```

Very nice! So roll rolls the die and showing tells us which number is showing. However, what if we try to look at what's showing before we've rolled the die (before we've set @numberShowing)?

```
class Die
```

```

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

  def showing
    @numberShowing
  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... well, at least it didn't give us an error. Still, it doesn't really make sense for a die to be "unrolled", or whatever nil is supposed to mean here. It would be nice if we could set up our new die object right when it's created. That's what initialize is for:

```
class Die

  def initialize
    # I'll just roll the die, though we
    # could do something else if we wanted
    # to, like setting the die with 6 showing.
    roll
  end

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

  def showing
    @numberShowing
  end

end

puts Die.new.showing
```

```
1
```

When an object is created, its initialize method (if it has one defined) is always called.

Our dice are just about perfect. The only thing that might be missing is a way to set which side of a die is showing... why don't you write a cheat method which does just 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!

So that's some pretty cool stuff we just covered. It's tricky, though, so let me give another, more interesting example. 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 we will need to be able to feed it, put it to bed, and take it on walks. Internally, our dragon will need to keep track of if it is hungry, tired, or needs to go, but we won't be able to see that when we interact with our dragon, just like you can't ask a human baby, "Are you hungry?". We'll also add a few other fun ways we can interact with our baby dragon, and when he is born we'll give him a name. (Whatever you pass into the new method is passed into the initialize method for you.) Alright, let's give it a shot:

```
class Dragon

  def initialize name
    @name = name
    @asleep = false
    @stuffInBelly     = 10  # He's full.
    @stuffInIntestine =  0  # He doesn't need to go.

    puts @name + ' is born.'
  end

  def feed
    puts 'You feed ' + @name + '.'
    @stuffInBelly = 10
    passageOfTime
  end

  def walk
    puts 'You walk ' + @name + '.'
    @stuffInIntestine = 0
    passageOfTime
  end

  def putToBed
    puts 'You put ' + @name + ' to bed.'
    @asleep = true
    3.times do
      if @asleep
        passageOfTime
      end
      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 'He giggles, which singes your eyebrows.'
    passageOfTime
  end

  def rock
    puts 'You rock ' + @name + ' gently.'
    @asleep = true
    puts 'He briefly dozes off...'
    passageOfTime
    if @asleep
      @asleep = false
      puts '...but wakes when you stop.'
    end
  end

  private

  # "private" means that the methods defined here are
  # methods internal to the object.  (You can feed
  # your dragon, but you can't ask him if he's hungry.)

  def hungry?
    # Method names can end with "?".
    # Usually, we only do this if the method
    # returns true or false, like this:
    @stuffInBelly <= 2
  end

  def poopy?
    @stuffInIntestine >= 8
  end

  def passageOfTime
    if @stuffInBelly > 0
      # Move food from belly to intestine.
      @stuffInBelly     = @stuffInBelly     - 1
      @stuffInIntestine = @stuffInIntestine + 1
```

```ruby
    else  # Our dragon is starving!
      if @asleep
        @asleep = false
        puts 'He wakes up suddenly!'
      end
      puts @name + ' is starving!  In desperation, he ate YOU!'
      exit  # This quits the program.
    end

    if @stuffInIntestine >= 10
      @stuffInIntestine = 0
      puts 'Whoops!  ' + @name + ' had an accident...'
    end

    if hungry?
      if @asleep
        @asleep = false
        puts 'He wakes up suddenly!'
      end
      puts @name + '\'s stomach grumbles...'
    end

    if poopy?
      if @asleep
        @asleep = false
        puts 'He wakes up suddenly!'
      end
      puts @name + ' does the potty dance...'
    end
  end

end

pet = Dragon.new 'Norbert'
pet.feed
pet.toss
pet.walk
pet.putToBed
pet.rock
pet.putToBed
pet.putToBed
pet.putToBed
pet.putToBed
```

```
Norbert is born.
You feed Norbert.
You toss Norbert up into the air.
He 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.
He briefly dozes off...
...but wakes when you stop.
You put Norbert to bed.
He wakes up suddenly!
Norbert's stomach grumbles...
You put Norbert to bed.
He wakes up suddenly!
Norbert's stomach grumbles...
You put Norbert to bed.
He wakes up suddenly!
Norbert's stomach grumbles...
Norbert does the potty dance...
You put Norbert to bed.
He wakes up suddenly!
Norbert is starving!  In desperation, he ate YOU!
```

Whew! Of course, it would be nicer if this was an interactive program, but you can do that part later. I was just trying to show the parts directly relating to creating a new dragon class.

We saw a few new things in that example. The first is simple: exit terminates the program right then and there. The second is the word private which we stuck right in the middle of our class definition. I could have left it out, but I wanted to enforce the idea of certain methods being things you can do to a dragon, and others which simply happen within the dragon. You can think of these as being "under the hood": unless you are an automobile mechanic, all you really need to know is the gas pedal, the brake pedal, and the steering wheel. A programmer might call those the public interface to your car. How your airbag knows when to deploy, however, is internal to the car; the typical user (driver) doesn't need to know about this.

Actually, for a bit more concrete example along those lines, let's talk about how you might represent a car in a video game (which happens to be my line of work). First, you would want to decide what you want your public interface to look like; in other words, which methods should people be able to call on one of your car objects? Well, they need to be able to push the gas pedal and the brake pedal, but they would also need to be able to specify how hard they are pushing the pedal. (There's a big difference between flooring it and tapping it.) They would also need to be able to steer, and again, they would need to be able to say how hard they are turning the wheel. I suppose you could go further and add a clutch, turn signals, rocket launcher, afterburner, flux capacitor, etc... it depends on what type of game you are making.

Internal to a car object, though, there would need to be much more going on; other things a car would need are a speed, a direction, and a position (at the most basic). These attributes would be modified by pressing on the gas or brake pedals and turning the wheel, of course, but the user would not be able to set the position directly (which would be like warping). You might also want to keep track of skidding or damage, if you have caught any air, and so on. These would all be internal to your car object.

A Few Things to Try too

Make an OrangeTree class. It should have a height method which returns its height, and a oneYearPasses method, which, 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 should not 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 most sense. And, of course, you should be able to countTheOranges (which returns the number of oranges on the tree), and pickAnOrange (which reduces the @orangeCount by one and returns a string telling you how delicious the orange was, or else it just tells you that there are no more oranges to pick this year). Make sure that any oranges you don't pick one year fall off before the next year.

Write a program so that you can interact with your baby dragon. You should be able to enter commands like feed and walk, and have those methods be called on your dragon. Of course, since what you are inputting are just strings, you will have to have some sort of method dispatch, where your program checks which string was entered, and then calls the appropriate method.

And that's just about all there is to it! But wait a second... I haven't told you about any of those classes for doing things like sending an email, or saving and loading files on your computer, or how to create windows and buttons, or 3D worlds, or anything! Well, there are just so many classes you can use that I can't possibly show you them all; I don't even know what most of them are! What I can tell you is where to find out more about them, so you can learn about the ones you want to program with. Before I send you off, though, there is just one more feature of Ruby you should know about, something most languages don't have, but which I simply could not live without: blocks and procs.

Blocks and Procs

Chapter 10

This is definitely one of the coolest features of Ruby. Some other languages have this feature, though they may call it something else (like closures), but most of the more popular ones don't, and it's a shame.

So what is this cool new thing? It's the ability to take a block of code (code in between do and end), wrap it up in an object (called a proc), store it in a variable or pass it to a method, and run the code in the block whenever you feel like (more than once, if you want). So it's kind of like a method itself, except that 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. I think it's example time:

```
toast = Proc.new do
  puts 'Cheers!'
end

toast.call
toast.call
toast.call
```

```
Cheers!
Cheers!
Cheers!
```

So I created a proc (which I think is supposed to be short for "procedure", but far more importantly, it rhymes with "block") which held the block of code, then I called the proc three times. As you can see, it's a lot like a method.

Actually, it's even more like a method than I have shown you, because blocks can take parameters:

```
doYouLike = Proc.new do |aGoodThing|
  puts 'I *really* like '+aGoodThing+'!'
end

doYouLike.call 'chocolate'
doYouLike.call 'ruby'
```

```
I *really* like chocolate!
I *really* like ruby!
```

Ok, so we see what blocks and procs are, and how to use them, but what's the point? Why not just use methods? Well, it's because there are some things you just 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 Which Take Procs

When we pass a proc into a method, we can control how, if, or how many times we call the proc. For example, let's say there's something we want to do before and after some code is run:

```
def doSelfImportantly someProc
  puts 'Everybody just HOLD ON!  I have something to do...'
  someProc.call
  puts 'Ok everyone, I\'m done.  Go on with what you were doing.'
end

sayHello = Proc.new do
  puts 'hello'
end

sayGoodbye = Proc.new do
  puts 'goodbye'
end

doSelfImportantly sayHello
doSelfImportantly sayGoodbye
```

```
Everybody just HOLD ON!  I have something to do...
hello
Ok everyone, I'm done.  Go on with what you were doing.
Everybody just HOLD ON!  I have something to do...
goodbye
Ok everyone, I'm done.  Go on with what you were doing.
```

Maybe that doesn't appear particulary fabulous... but it is. 😃 It's all too common in programming to have strict requirements about what must be done when. If you want to save a file, for example, you have to open the file, write out the information you want it to have, and then close the file. If you forget to close the file, Bad Things(tm) 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, then close the file. It's tedious and easy to forget. In Ruby, saving (or loading) files works similarly to the code above, so you don't have to worry about anything but what you actually want to save (or load). (In the next chapter I'll show you where to find out how to do things like save and load files.)

You can also write methods which will determine how many times, or even if to call a proc. Here's a method which will call the proc passed in about half of the time, and another which will call it twice:

```
def maybeDo someProc
```

```
  if rand(2) == 0
    someProc.call
  end
end

def twiceDo someProc
  someProc.call
  someProc.call
end

wink = Proc.new do
  puts '<wink>'
end

glance = Proc.new do
  puts '<glance>'
end

maybeDo wink
maybeDo glance
twiceDo wink
twiceDo glance
```

```
<wink>
<wink>
<glance>
<glance>
```

These are some of the more common uses of procs which enable us to do things we simply could not have done using methods alone. Sure, you could write a method to wink twice, but you couldn't write one to just do something twice!

Before we move on, let's look at one last example. So far the procs we have passed in have been fairly similar to each other. This time they will be quite different, so you can see how much such a method depends on the procs passed into it. Our method will take some object and a proc, and will call the proc on that object. If the proc returns false, we quit; otherwise we call the proc with the returned object. We keep doing this until the proc returns false (which it had better do eventually, or the program will crash). The method will return the last non-false value returned by the proc.

```
def doUntilFalse firstInput, someProc
  input  = firstInput
  output = firstInput

  while output
    input  = output
    output = someProc.call input
  end

  input
end

buildArrayOfSquares = Proc.new do |array|
  lastNumber = array.last
  if lastNumber <= 0
    false
  else
    array.pop                         # Take off the last number...
    array.push lastNumber*lastNumber  # ...and replace it with its square...
    array.push lastNumber-1           # ...followed by the next smaller number.
  end
end

alwaysFalse = Proc.new do |justIgnoreMe|
  false
end

puts doUntilFalse([5], buildArrayOfSquares).inspect
puts doUntilFalse('I\'m writing this at 3:00 am; someone knock me out!',
alwaysFalse)
```

```
[25, 16, 9, 4, 1, 0]
I'm writing this at 3:00 am; someone knock me out!
```

Ok, so that was a pretty weird example, I'll admit. But it shows how differently our method acts when given very different procs.

The inspect method is a lot like to_s, except that the string it returns tries to show you the ruby code for building the object you passed it. Here it shows us the whole array returned by our first call to doUntilFalse. Also, you might notice that we never actually squared that 0 on the end of that array, but since 0 squared is still just 0, we didn't have to. And since alwaysFalse was, you know, always false, doUntilFalse didn't do anything at all the second time we called it; it just returned what was passed in.

Methods Which Return Procs

One of the other cool things you can do with procs is to create them in methods and return them. This allows all sorts of crazy programming power (things with impressive names, like lazy evaluation, infinite data structures, and currying), but the fact is that I almost never do this in practice, nor can I remember seeing anyone else do this in their code. I think it's the kind of thing you don't usually end up having to do in Ruby, or maybe Ruby just encourages you to find other solutions; I don't know. In any case, I will only touch on this briefly.

In this example, compose takes two procs and returns a new proc which, when called, calls the first proc and passes its result into the second proc.

```
def compose proc1, proc2
  Proc.new do |x|
    proc2.call(proc1.call(x))
  end
end

squareIt = Proc.new do |x|
  x * x
end

doubleIt = Proc.new do |x|
  x + x
end

doubleThenSquare = compose doubleIt, squareIt
squareThenDouble = compose squareIt, doubleIt

puts doubleThenSquare.call(5)
puts squareThenDouble.call(5)
```

```
100
50
```

Notice that the call to proc1 had to be inside the parentheses for proc2 in order for it to be done first.

Passing Blocks (Not Procs) into Methods

Ok, so this has been sort of academically interesting, but also sort of a hassle to use. A lot of the problem is that there are three steps you have to go through (defining the method, making the proc, and calling the method with the proc), when it sort of feels like there should only be two (defining the method, and passing the block right into the method, without using a proc at all), since most of the time you don't want to use the proc/block after you pass it into the method. Well, wouldn't you know, Ruby has it all figured out for us! In fact, you've already been doing it every time you use iterators.

I'll show you a quick example first, then we'll talk about it.

```
class Array
  def eachEven(&wasABlock_nowAProc)
    # We start with "true" because arrays start with 0, which is even.
    isEven = true

    self.each do |object|
      if isEven
        wasABlock_nowAProc.call object
      end

      isEven = (not isEven)  # Toggle from even to odd, or odd to even.
    end
  end
end

['apple', 'bad apple', 'cherry', 'durian'].eachEven do |fruit|
  puts 'Yum!  I just love '+fruit+' pies, don\'t you?'
end

# Remember, we are getting the even-numbered elements
# of the array, all of which happen to be odd numbers,
# just because I like to cause problems like that.
[1, 2, 3, 4, 5].eachEven do |oddBall|
  puts oddBall.to_s+' 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!
```

So to pass in a block to eachEven, all we had to do was stick the block after the method. You can pass a block into any method this way, though many methods will just ignore the block. In order to make your method not ignore the block, but grab it and turn it into a proc, put the name of the proc 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 only have to do that once (when you define the method). Then you can use the method over and over again, just like the built-in methods which take blocks, like each and times. (Remember 5.times do...?)

If you get confused, just remember what eachEven 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 just use them.

I remember one time I wanted to be able to time how long different sections of a program were taking. (This is also known as profiling the code.) So I wrote a method which takes the time before running the code, then it runs it, then it takes the time again at the end and figures out the difference. I can't find the code right now, but I don't need it; it probably went something like this:

```
def profile descriptionOfBlock, &block
  startTime = Time.now

  block.call
  duration = Time.now - startTime

  puts descriptionOfBlock+':  '+duration.to_s+' seconds'
end

profile '25000 doublings' do
  number = 1

  25000.times do
    number = number + number
  end

  # Show the number of digits in this HUGE number.
  puts number.to_s.length.to_s+' digits'
end

profile 'count to a million' do
  number = 0

  1000000.times do
    number = number + 1
  end
end
```

```
7526 digits
25000 doublings:  0.109064 seconds
count to a million:  0.124997 seconds
```

How simple! How elegant! With that tiny method, I can now easily time any section of any program that I want to; I just throw the code in a block and send it to profile. What could be simpler? In most languages, I would have to explicitly add that timing code (the stuff in profile) around every section which I wanted to time. In Ruby, however, I get to keep it all in one place, and (more importantly) out of my way!

A Few Things to Try

Grandfather Clock. Write a method which 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 'DONG!' end, it would chime (sort of) like a grandfather clock. Test your method out with a few different blocks (including the one I just gave you). Hint: You can use Time.now.hour to get the current hour. However, this returns a number between 0 and 23, so you will have to alter those numbers in order to get ordinary clock-face numbers (1 to 12).

Program Logger. Write a method called log, which takes a string description of a block and, of course, a block. Similar to doSelfImportantly, it should puts a string telling that it has started the block, and another string at the end telling you that it has finished the block, and also telling you what the block returned. Test your method by sending it a code block. Inside the block, put another call to log, passing another block to it. (This is called nesting.) 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 Logger. The output from that last logger was kind of hard to read, and it would just get worse the more you used it. It would be so much easier to read if it indented the lines in the inner blocks. To do this, 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, a variable you can see from anywhere in your code. To make a global variable, just precede your variable name with $, like these: $global, $nestingDepth, and $bigTopPeeWee. 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
```

Well, that's about all you're going to learn from this tutorial. Congratulations! You've learned a lot! Maybe you don't feel like you remember everything, or you skipped over some parts... really, that's just fine. Programming isn't about what you know; it's about what you can figure out. As long as you know where to find out the things you forgot, you're doing just fine. I hope you don't think that I wrote all of this without looking things up every other minute! Because I did. I also got a lot of help with the code which runs all of the examples in this tutorial. But where was I looking stuff up, and who was I asking for help? Let me show you...Blocks and Procs

Beyond This Tutorial

Chapter 11

So where do we go now? If you have a question, who can you ask? What if you want your program to open a webpage, send an email, or resize a digital picture? Well, there are many, many places to find Ruby help. Unfortunately, that's sort of unhelpful, isn't it? 😃

For me, there are really only three places I look for Ruby help. 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. And if I just can't figure it out on my own, then I ask for help on ruby-talk.

IRB: Interactive Ruby

If you installed Ruby, then you installed irb. To use it, just go to your command prompt and type irb. When you are in irb, you can type in any ruby expression you want, and it will tell you the value of it. Type in 1 + 2, and it will tell you 3. (Note that you don't have to use puts.) It's kind of like a giant Ruby calculator. When you are done, just type in exit.

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, The Pragmatic Programmer's Guide", by David Thomas and Andrew Hunt (the Pragmatic Programmers). While I highly recommend picking up the 4th edition of this excellent book, with all of the latest Ruby covered, you can also get a slightly older (but still mostly relevant) version for free online.

You can find just about everything about Ruby, from the basic to the advanced, in this book. It's easy to read; it's comprehensive; it's just about 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. I just love this book!

There are a number of places you can get it (including the Pragmatic Programmers' own site), but my favorite place is at ruby-doc.org. That version has a nice table of contents on the side, as well as an index. (ruby-doc.org has lots of other great documentation as well, such as for the Core API and Standard Library... basically, it documents everything Ruby comes with right out of the box. Check it out.)

And why is it called "the pickaxe"? Well, there's a picture of a pickaxe on the cover of the book. It's a silly name, I guess, but it stuck.

Ruby-Talk: the Ruby Mailing List

Even with irb and the pickaxe, sometimes you still can't figure it out. Or perhaps you want to know if someone already did whatever it is you are working on, to see if you could use it instead. In these cases, the place to go is ruby-talk, the Ruby Mailing List. It's full of friendly, smart, helpful people. To learn more about it, or to subscribe, look here.

WARNING: There's a lot of mail on the mailing list every day. I have mine automatically sent to a different mail folder so that it doesn't get in my way. If you just don't want to deal with all that mail, though, you don't have to! The ruby-talk mailing list is mirrored to the newsgroup comp.lang.ruby, and vice versa, so you can see the same messages there. Either way, you see the same messages, just in a slightly different format.

Tim Toady

Something I have tried to shield you from, but which you will surely run in to soon, is the concept of TMTOWTDI (pronounced "Tim Toady"): There's More Than One Way To Do It.

Now some will tell you what a wonderful thing TMTOWTDI is, while others feel quite differently. I don't really have strong feelings about it in general, but I think it's a terrible way to teach someone how to program. (As if learning one way to do something wasn't challenging and confusing enough!)

However, now that you are moving beyond this tutorial, 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 single quotes), and each one works slightly differently. I only showed you the simplest 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:

```
# These words are from a program I wrote to generate
# English-like babble.  Cool, huh?
puts 'grobably combergearl thememberate' if 5 == 2**2 + 1**1
puts 'enlestrationshifter supposine' unless 'Chris'.length == 5
```

```
grobably combergearl thememberate
```

And finally, there is another way of writing methods which take blocks (not procs). We saw the thing where we grabbed the block and turned it into a proc using the &block trick in your parameter list when you define the function. Then, to call the block, you just use block.call. Well, there's a shorter way (though I personally find it more confusing). Instead of this:

```
def doItTwice(&block)
  block.call
  block.call
end

doItTwice do
  puts 'murditivent flavitemphan siresent litics'
end
```

```
murditivent flavitemphan siresent litics
murditivent flavitemphan siresent litics
```

...you do this:

```ruby
def doItTwice
  yield
  yield
end

doItTwice do
  puts 'buritiate mustripe lablic acticise'
end
```

```
buritiate mustripe lablic acticise
buritiate mustripe lablic acticise
```

I don't know... what do you think? Maybe it's just me, but... yield?! If it was something like call_the_hidden_block or something, that would make a lot more sense to me. A lot of people say yield makes sense to them. But I guess that's what TMTOWTDI is all about: they do it their way, and I'll do it my way.

THE END

Use it for good and not evil. 😃 And if you found this tutorial useful (or confusing, or if you found an error), let me know!Beyond This Tutorial